THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Congestion control mechanism for multimedia applications

Rosman, Cédric

Award date:
2001

Congestion Control Mechanism for Multimedia Applications

Cédric Rosman

Promoteur : Olivier Bonaventure

Année académique 2000-2001

Travail réalisé dans l'intention d'obtenir le grade de maître en informatique

Abstract

While *TCP* and its congestion control mechanism deals with the major share of the Internet traffic today, insuring stability and fairness for users, recently proposed real-time multimedia services (such as IP-telephony, group communication (video or phone conference), distant learning, ...) based on the *UDP* protocol arise. Offering neither reliability nor congestion control mechanism, deploying uncontrolled *UDP* traffic in a large scale might result in extreme unfairness towards competing *TCP* traffic. In this thesis, we will compare the two main used transport protocols (*TCP* and *UDP*), pointing out the advantages and drawbacks of each related with those new types of services. We will present a new scheme called *Rate Adaptive protocol* (*RAP*) for adapting the transmission rate of multimedia applications to the congestion level of the network. *RAP* was designed to mimic *TCP* in its behaviour, working in a TCP-Friendly way to avoid starving competing *TCP* flows. Relying on packets acknowledgment and feedback information, *RAP* estimates what should be the fair throughput and adapts the time between the sending of two consecutive packets in consequence. Afterwards, we will introduce other congestion control mechanisms, different in their ways of working and implementation, to be compared to *RAP*. Finally, simulations and measurements of the *RAP* algorithm will show its TCP-Friendliness related with its consumption of the network resources while competing with multiple *TCP* flows.

Alors que *TCP* et son mécanisme de contrôle de congestion est utilisé pour une large majorité du trafic Internet de nos jours, assurant la stabilité et l'équité entre les utilisateurs, de récents services multimédia (comme la téléphonie sur *IP*, les groupes de communication (par vidéo ou oralement), l'apprentissage à distance, ...) basés sur le protocole *UDP* ont émergés. N'offrant ni fiabilité ni mécanismes de contrôle de congestion, le déploiement de trafic *UDP* non-contrôlé *à* une large échelle pourrait mener *à* une importante iniquité envers les flux *TCP* en compétition. Dans ce mémoire, nous comparerons les deux principaux protocoles de transport utilisés (*TCP* et *UDP*), indiquant leurs avantages et défauts respectifs quant à ce genre de nouveaux services. Nous présenterons un nouveau protocole appelé *Rate Adaptive Protocol* (*RAP*) qui adapte le taux de transmission des applications multimédia au niveau de congestion du réseau. *RAP* a été réalisé dans le but d'imiter *TCP* dans son comportement, fonctionnant de façon â éviter la mort des flux *TCP* en compétition. Se basant sur les acquis de paquets et différentes mesures, *RAP* estime ce que devrait être le taux de transmission équitable et adapte le temps écoulé antre deux paquets transmis en conséquence. Ensuite, nous introduirons d'autres mécanismes de contrôle de congestion, différents dans leur fonctionnement et leur implémentation, pour être comparés *à* *RAP*. Enfin, des simulations et mesures de l'algorithme de *RAP* montreront son caractère "amicale" quant *à* sa consommation en ressources du réseau face *à* plusieurs flux *TCP*

Acknowledgments

I would like to thank first of all my promoter Mr the Teacher O. Bonvanenture for the time and the pieces of advice he gave me through the elaboration of this work.

I express my deep gratitude to Mr S. De Cnodder, my external promoter at Alcatel Antwerp, for the precious help he brought me, as well as to Mr G. H. Petit, the director of the Research Department at Alcatel Antwerp, for giving me the opportunity to make my training course within his department.

I also would like to thank the 3 assistants P. Reinbold, S. Uhlig and V. Letocar for their availability and their kindness.

At last, I thank all the persons, and in particular Cristel and Louis (during our stay in Antwerp), who, directly or indirectly, have helped me achieving this thesis.

Glossary

ACK	Acknowledgment
ADWIN	advertised WINdow
AIMD	Additive Increase / Multiplicative Decrease
BE	Best Effort
CWND	Congestion WiNDow
ECN	Explicit Congestion Notification
FG	Fine Grain
FIFO	First In / First Out
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISP	Internet Service Provider
LDA	Loss Delay Adjustment
MSS	Maximum Segment Size
MTU	Maximum Transmit Unit
PDF	Probability Density Function
QoS	Quality of Service
RAP	Rat Adaptive Protocol
RED	Random Early Drop
RTCP	Real-Time Transport Control Protocol
RTP	Real-Time Transport Protocol
RTT	Round Trip Time
SYN	Synchronization
TCP	Transmission Control Protocol
TEAR	TCP Emulation At Receiver
TFRC	TCP Friendly Rate Control
UDP	User Datagram Protocol

Contents

List of Tables

List of Figures

Chapter 1

Introduction

While *TCP* and its congestion control mechanism deals with the major share of the Internet traffic today, insuring stability and fairness for users, recently proposed real-time multimedia services, such as IP-telephony, group communication (video or phone conference), video on demand, distant learning, ... are based on *UDP* protocol. While it does not offer reliability or congestion control mechanism, *UDP* is well suited to that kind of applications: no additional delays, no acknowledgments (lighter traffic for multicast), ... But deploying uncontrolled *UDP* traffic in a large scale might result in extreme unfairness towards competing *TCP* traffic.

In the last few years, there has been considerable research toward extending the Internet architecture to provide Quality of Service (QoS) guarantees for the emerging real-time multimedia applications. On one hand researchers proposed QoS reservations and per-flow state in the routers, which could be considered as long term solutions but still have enormous drawbacks: the network heterogeneity (thus hard deployment), the complexity of the involved mechanisms and scalability problems. On the other hand to bet that an over-provisioned best effort network will solve all the problems is really an uncertain bet.

More control is clearly needed to avoid congestion collapse and also to insure fairness between users, to guarantee friendliness between *TCP* and non-TCP flows but this control also has to maintain the simplicity of a best effort network, to be easily deployed and to be as simple as possible.

1.1 Why this thesis?

1.1.1 General situation

From the beginning and still now, the Internet is almost exclusively based on the *Best Effort (BE)* transmission concept: all packets are treated the same without any discrimination or explicit delivery guarantees. This really simple concept consists of doing its "*best effort*" to deliver the injected packets from wherever they come to wherever they go. The achieved quality of treatment for users does not only depend on the network resources but also on the other users and the amount of information to transmit. This leads to a lack of isolation and protection between flows.

The first level of protection against an increase of traffic arrival rate stands in the buffer space of the routers traditionally following the *First In - First Out (FIFO)* buffer management policy [1] consisting in forwarding packets as they arrive or dropping arriving packets in case of buffer overflow. But this can only be a temporary solution. If the situation persists, the buffer runs out of space and routers begin to drop packets. However, an "infinite" space is not the solution. Offering the advantage of not discarding packets (because not undergoing buffer overflow), it has the absolute disadvantage of increasing the end-to-end delay.

These concepts (best effort and FIFO policy) played an important role in the Internet deployment and stability. Because of their flexibility and robustness, it can operate under a wide range of network conditions without requiring specific configuration or adaptations.

However, a completely uncontrolled network may suffer from congestion or worse, congestion collapse (cf. section 1.1.3 for definition). That kind of problem occurred in the past (mid '80s) several times and led to the implementation and deployment of a set of congestion control functions in *TCP* (located in hosts mainly to avoid the problem of deployment in updating all routers). The goals of these functions are:

- To protect the Internet from congestion collapse.
- To share the available resources (bandwidth) between all users in a "fair" way.

[1] This management associates simplicity of concept and implementation but is precisely too simple, doing no difference between flows. The first proposal of active queue management was *Random Early Detection (RED)* which is still studied and improved nowadays ([FJ93], [CP00] and [CE99]).

That kind of congestion control mechanism relies on congestion detection performed by hosts but also by routers. The simplest congestion indication in a best effort network, using FIFO buffer management, is the packet loss which is an implicit feedback information. Explicit feedback also exists and consists for *TCP* in the ICMP Source Quench messages (cf. [Pos81a])and the Explicit Congestion Notification (ECN) (cf. [RF99]). But the efficiency of these functions heavily depends on a correct implementation at the users side and the utilization of everybody. These mechanisms have also been parts of the key contributors to the success of the Internet (cf. Chapter 2 for *TCP* way of working).

1.1.2 Problems

Nowadays, with the increasing growth of non-responsive applications (non TCP-based transmission), congestion control has to be extended to non-TCP flows, i.e. TCP-Friendly flow (cf. section 1.1.3 for definition). As a matter of fact, users who misbehave (not following *TCP*'s rule) capture more bandwidth than their fair share, seriously degrading the service delivered to cooperating users, and in general threaten the stability and the operation of the entire system. This is why congestion control mechanism for non-TCP applications is really important.

One reason of not using *TCP* for those applications stands in the complete inadequate way of working of *TCP* (delays, retransmissions, ...) related with *UDP* which offers severe advantages for them (cf. Chapter 2 for more explanation). Unfortunately, *UDP* does not dispose of any congestion control mechanism. Some lack of service are performed by the *Real-Time Transport Protocol (RTP)* above *UDP* but it still remains too uncontrolled. Something has to be added.

1.1.3 Definition of major concepts

- **Congestion**: is the state of sustained network overload where the demand for network resources is close to or exceeds the available capacity. Network resources, namely link bandwidth and buffer space in the routers, are both finite and in many cases still expensive (traffic is increasing while memory price is decreasing). This congestion can cause high packet loss rates, increased delays and can lead to congestion collapse (or "Internet meltdown")

- **Congestion collapse**: is the state where any increase in the offered load leads to a decrease of the useful work done by the network (overloaded). It may be due to other different reasons:

- *Undelivered packets* waste bandwidth by transmitting packets that will be dropped before they reach their final destinations.
- *Fragmented packets* wasted bandwidth by delivering fragments of packets that will be discarded at the receivers since they cannot reassemble them into a valid packet.
- *Stale packets* waste bandwidth by carrying packets that are no longer wanted by the users (took too much time).

- **TCP-Friendly flow**: is not an easy notion to define.
  - Internet Engineering Task Force (IETF) mandates that a non-TCP flow does not send more than a *TCP* flow would do under similar network conditions
  - If a *TCP* connection and an adaptive flow with similar transmission behaviours have similar round trip delays and losses they should receive similar bandwidth shares.
  - Non-TCP flows are considered TCP-Friendly if their long-term throughput does not exceed the throughput of a conformant *TCP* under the same conditions.

- **Fairness**: Under conditions of low load, everybody's demands are satisfied (no trade-offs, no considerations). When there are unsatisfied demands and users have to compete for their fare share, the classical notion of fairness seems to be what is called the *max-min fairness.*

- **max-min Fairness**: "The greatest benefit for the least advantaged", an allocation of bandwidth which maximizes the allocation of bandwidth to the sources receiving the smallest allocation (to increase the bandwidth allocated to one source, you have to decrease the allocation of another source which already received a lower allocation). It consists of sharing the resources in an incremental way. It first start with an allocation of 0 Mbps. Then it equally increments the allocation to each source until one link becomes satured. (Sources using this satured link receive an equal share of the bandwidth) Then the allocation of all the sources not using the satured link are equally incremented until next satured link and so on ...

1.2 Structure of the thesis

The rest of the thesis will continue with the following structure.

Chapter 2 explains the two main transport protocols used on the Internet: Transmission Control Protocol (*TCP*) and User Datagram Protocol (*UDP*) both based on the *IP* layer. It also explains the needs of real-time streaming applications and the problems met with such applications requirements.

Chapter 3 describes in details the way of working of the Rate Adaptation Protocol (*RAP*), the implementation choices and the improving mechanisms.

Chapter 4 introduces different mechanisms of congestion control with different schemes of working.

Chapter 5 describes through multiple simulations various aspects of the *RAP* protocol and its TCP-Friendly behaviour.

Chapter 6 concludes the thesis: it reminds the main goals of the evaluated protocol, the results obtained through the simulations and gives some guidelines for further works.

Chapter 2

Transport protocols: *TCP* Vs *UDP*

In this chapter, we describe the two main transport protocols used nowadays on the Internet: *Transmission Control Protocol* (*TCP*) and *User Datagram Protocol* (*UDP*), giving their advantages and disadvantages. Based on the characteristics and requirements of real-time streaming applications, one of the protocols is preferred but some problems remain, problems that require one of the mechanisms introduced in Chapter 4 and 3.

2.1 Transmission Control Protocol (*TCP*)

The *Transmission Control Protocol* (*TCP*), specified in [Pos81b], [Bra89] and [Ste94], provides a reliable connection-oriented byte stream service over an unreliable packet-based *IP* service, characterized by a single packet format protected by a checksum.

Connection-oriented means that two applications using *TCP* have to establish a connection before beginning to exchange any data (exactly like phone calls). It also offers a full-duplex service to the application, which allows sending data in both directions of the connection.

Data delivered by the application to the *TCP* layer are introduced in a fragment, called *segment*. The estimation of the segment best size is an option of *TCP* and is done at the establishment of the connection.

A byte stream service means that the sender just puts inside this *segment* the data bytes given by the application without any markers to separate the different writings. *TCP* does not interpret the payload of the segment (*TCP* does not know which format is used); it is left to the application. *TCP* just incorporated this segment in an *IP* datagram (cf. Figure 2.1 for the general structure of an *IP* datagram).

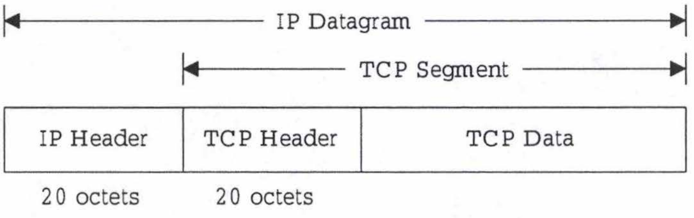


Figure 2.1: Structure of TCP/IP datagram

2.1.1 *TCP* segment structure

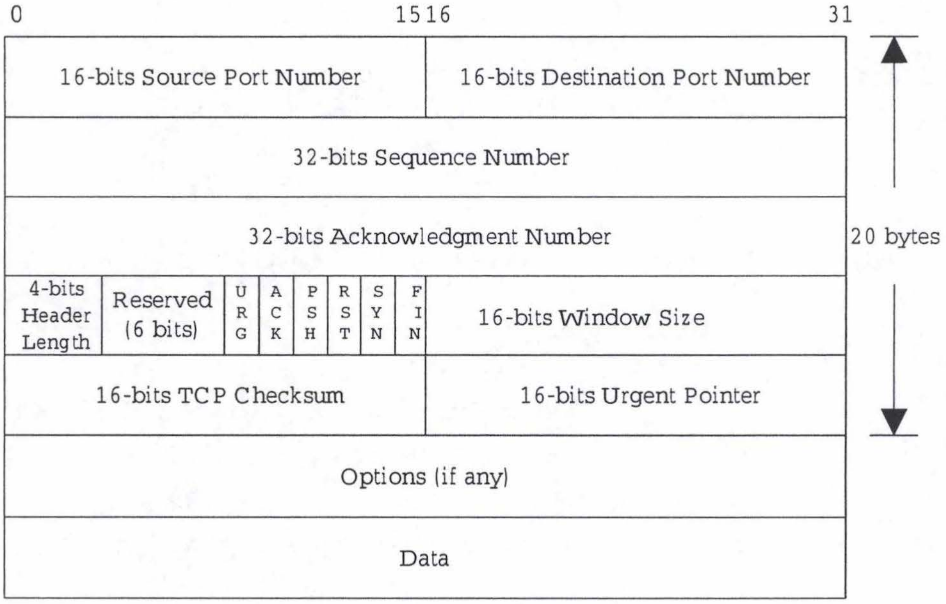


Figure 2.2: Structure of *TCP* segment

The fields:

- The *source and destination port numbers* identify the sending and receiving applications at the ends of the connection. Combined with the *IP* source and destination addresses and the protocol, it identifies a connection.

- The *sequence number* identifies each *TCP* segment in a message stream. It specifies the number of the first byte of each segment.

- The *acknowledgment number* contains the sequence number of the next byte the receiver is expecting to receive, this means the last sequence number received correctly + the segment size (also cf.ACK flag). In the original version of *TCP*, there is no mean to acknowledge specific segment.

- The *header length* contains the number of 32-bits words in the *TCP* header. 4 bits imply a maximum size of 60 octets (default $= 5 \equiv [0101] \equiv 20$ octets).

- The *reserved 6-bits* are for future and not yet specified use (expected for ECN option).

- The *flags*: if sets to 1, this means that

  URG: the urgent pointer field is valid (some data has to be processed immediately).

  ACK: the acknowledgment number field is valid.

  PSH: the receiver should forward all its data (segment + buffer) to the application immediately.

  RST: reset of the connection.

  SYN: synchronisation of the sequence numbers at the connection establishment.

  FIN: end of transmission for the sender.

- The *window size* is the central key of the flow control (cf. [Mog93]). It indicates the number of bytes that the receiver is able to receive, starting from the acknowledgment number field. 16 bits limits the window to a maximum of 65 535 bytes.

- The *checksum* is computed like *UDP*: complement of the sum of 16-bits length words.It takes into account the header and the payload. As for *UDP*, the checksum is computed with a pseudo-IP-header composed of the *IP* source and destination addresses, the protocol , the segment length and the padding (see 2.10). The *TCP* checksum is mandatory (unlike *UDP*).

- The *urgent pointer* is a value to add to the sequence number field to determine the end of the urgent data to be processed immediately (only taken into account when the URG flag is set to 1).

- The *option field* specifies option(s) between end systems. The most widely used option is the *maximum segment size (MSS)*: it specifies the maximum segment size the receiver agrees to receive and is determined by both sides at the establishment of the connection (SYN flag set).

Notes: the payload is optional. At the establishment and closing of a connection, only segments with header (and options if any) are exchanged. Empty segments are also used to acknowledge received segments when there is no data to send in the opposite direction.

2.1.2 Way of working

1) Offered services

As said before, *TCP* offers a reliability service. This means: loss detection and retransmission, segment integrity, detection and discard of duplicated segments, re-ordering of segments.

- *Loss detection and retransmission*: when the source sends its segment, it maintains a timer while waiting for the acknowledgement from the destination. In case of loss, the source retransmits the missing packet(s)
- *Integrity* of *TCP* segments: performed by a checksum in a header fields and checked at the end-points, a segment with an invalid checksum (data have been corrupted during the transfer) is rejected and not acknowledged to force the retransmission.
- *Duplication* of *IP* datagram: may occur in the network, so *TCP* must not take them into account and just have to discard them.
- *Re-ordering* of *TCP* segments: *IP* datagrams could follow different ways through the network, so they arrive not in sequence. *TCP* reorders the segments at the destination to correctly detect loss(es).

Further more, *TCP* also offers a flow control: it is a mechanism to prevent the source from over running the receiver's resources. By a dynamic allocation of buffer to receive data, the receiver warns the sender about the amount of data he is able to accept with (such that if he is slower than the sender, he will not run out of buffers).

2) Connection establishment and closing

a) **Connection establishment**: The connection establishment is made via a mechanism called *Three Way Handshake* and depicted on Figure 2.3.

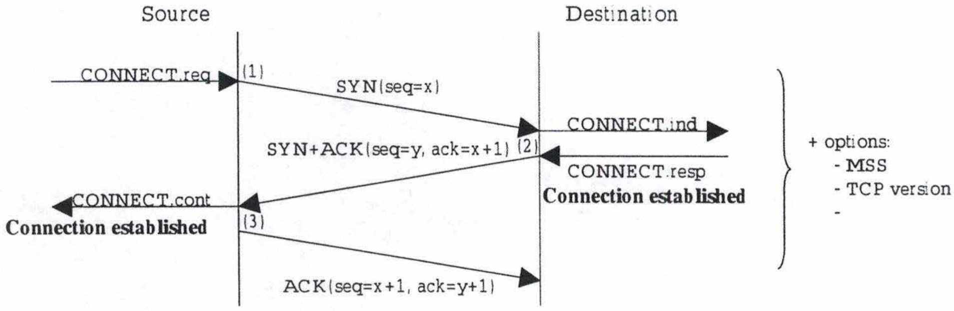


Figure 2.3: Establishment of a *TCP* connection

(1) The source sends a segment with an empty payload and the SYN flag sets to request for a connection. It may also try to negociate, in the option field, some options like MSS and *TCP* extensions (TCP-SACK for example). It also gives the initial sequence number of its first segment.

(2) The receiver acknowledges the connection request and confirms the connection by sending an (empty) segment with the ACK and the SYN flag set. It will also communicate its options.

(3) The sender confirms the connection establishment by sending a third empty segment with the ACK flag set, indicating the next segment he expects to received (just as for (2)).

Figure 2.4 exhibits the states machine of different connection establishments.

Path a: a typical source path (active opening).

Path b: a typical destination path (passive opening).

Path c: simultaneous opening path (both opening)

b) **Connection closing**: There are two kinds of closing: a symmetric one and an abrupt one. The symmetric closing is preferred because it guarantees that all segments have been received correctly. This connection closing is made via a mechanism called *Two Half-Close*. To close a connection, you need to close the two directions (as shown on Figure 2.5) because a *TCP* connection is a full-duplex connection.

3) **Data transfer**

The transfer mode is mainly based on three mechanisms: a congestion control, a flow control and timeouts mechanisms.

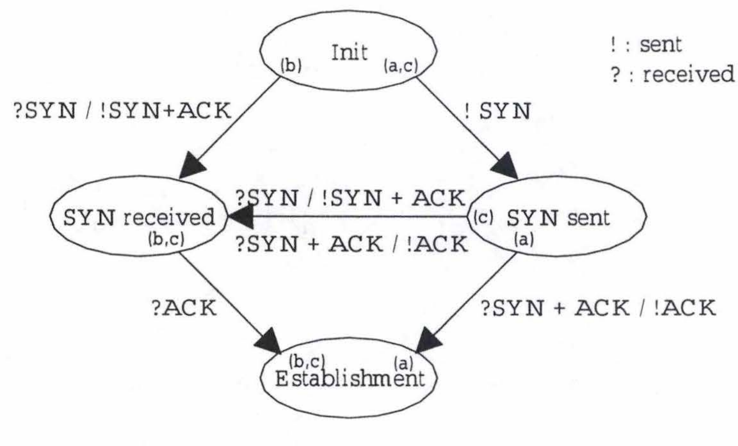


Figure 2.4: Establishment of a *TCP* connection

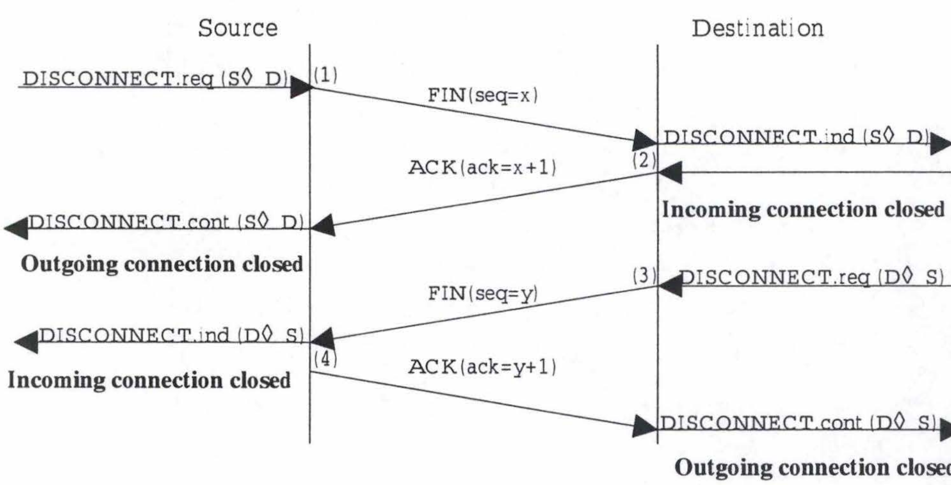


Figure 2.5: Closing of a *TCP* connection

The **congestion control mechanism** of *TCP* is based on the *Additive Increase / Multiplicative Decrease (AIMD)* algorithm, which is described in Figure 2.6 and can be expressed as follows:

- When no congestion is undergone, *TCP* additively increase its congestion window (CWND) to probe the network[1].
- When congestion is detected (packet loss[2]), *TCP* multiplicatively reduces its congestion window (by half).

[1]under some conditions: the window buffer size of the receiver, ...
[2]by triggered timer or duplicate acknowledgements

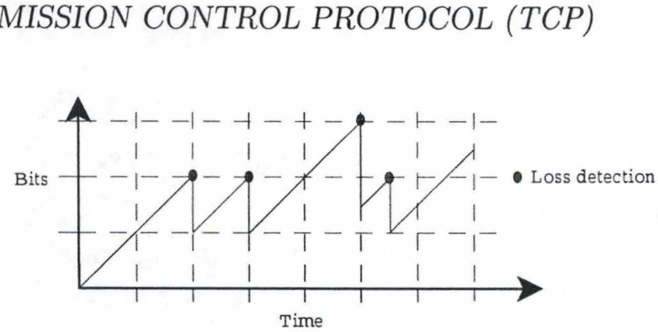


Figure 2.6: Additive increase / multiplicative decrease behaviour

The amount of sent packets on the network is determined by the **flow control** of *TCP*. It was at the beginning simply based on the sender's window buffer occupancy. It allowed *TCP* to transmit multiple packets without having to wait for an acknowledgement. This is at the origin of the bursty characteristic of the *TCP* transmission. At each time, the sender kept a list of sequence numbers that he used to send packets (not yet acknowledged by the receiver). In the same way, the receiver also had a list (*ADvertised WINdow (ADWIN)*) of packet sequence numbers that he already received or accepted to receive.

To estimate that a packet is lost, *TCP* relies on a set of **variables and timeout**. *TCP* computes the round trip time (the time for a sent packet to reach its destination and to be correctly acknowledged), uses it to have an estimate smooth RTT and combines this one with an estimation of the variation of the RTT to obtain the value of the retransmission timeout associated with the next packets to be sent. At the end of this timeout, if the associated packets are not yet acknowledged, *TCP* considers it as a loss and retransmits the packets.

All the mechanisms together constitute the *window-based rate flow control* that characterized *TCP*.

After this fast description of the first *TCP* ways of working, here are four algorithms developed by Van Jacobson ([Jac00]) and adopted by most operating systems to improve *TCP* in its adaptation scheme for the network.

(a) ***TCP* Slow Start algorithm**

Old *TCP* implementation would start a connection with the sender injecting multiple segments into the network, up to the window size advertised by the receiver. While this is OK when the two hosts are on the same LAN, if there are routers and slower links between the sender and the receiver, problems can arise:

- Some intermediate router must queue the packets,
- It's possible for that router to run out of space.

[Jac88] shows how this naive approach can reduce the throughput of a *TCP* connection drastically.

The algorithm proposed in [Jac88] to avoid this congestion collapse is called *slow start*. It operates by observing that the rate at which new packets should be injected into the network is the rate at which the acknowledgments are returned by the other end.

Slow start adds another window to the sender's *TCP*: *the congestion window (CWND)*. When a new connection is established with a host, the congestion window is initialised at one segment (i.e., the maximum segment size announced by the other end, MSS option at establishment, or the default, typically 536 or 512) and the SSThresh (threshold of slow transmission) at 65 535 Bytes (see Figure 2.7). Each time an ACK is received, the congestion window is increased by one segment. The sender can transmit up to the minimum of the congestion window and the advertised window.

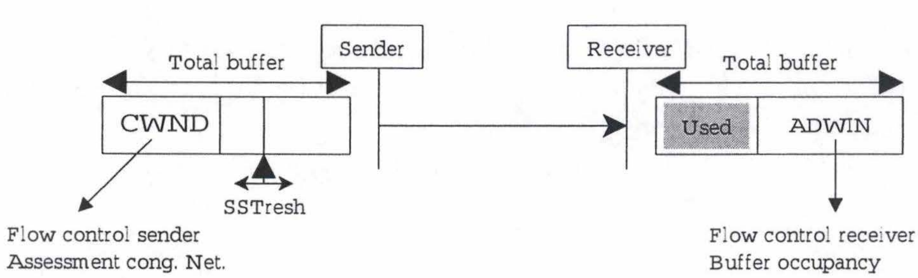


Figure 2.7: Congestion and advertised windows

The congestion window is flow control led by the sender, while the advertised window is flow control led by the receiver. The former is based on the sender's assessment of the perceived network congestion;. The latter is related to the amount of available buffer space at the receiver for this connection.

The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, the congestion window is incremented from one to two, and two segments can be sent. When each of those two segments is acknowledged, the congestion window is increased to four segments. This provides an exponential growth, although it is not exactly exponential because the receiver may delay its ACKs, typically sending one ACK for every two segment that it receives.

At some point the capacity of the Internet can be reached, and an intermediate router will start discarding packets. This informs the sender that its congestion window has gotten too large.

(b) **Congestion Avoidance algorithm**

Congestion can occur when data arrives from a big pipe (a fast LAN) and is sent out on a slower pipe (a slower WAN). Congestion can also occur when multiple input streams arrive at a router whose output capacity is less than the sum of its inputs. Congestion avoidance is a way to deal with lost packets.

The assumption of the algorithm is that packet loss caused by damage is very small (much less than 1

Congestion avoidance and slow start are independently implemented algorithms with different objectives but are highly correlated. But when congestion occurs *TCP* must slow down its transmission rate of packets into the network, and then invoke slow start to get things going again. In practice they are implemented together.

Congestion avoidance and slow start require that two variables be maintained for each connection: a congestion window, CWND, and a slow start threshold size, SSthresh. The combined algorithm operates as follows:

i. Initialisation for a given connection sets CWND to one segment and SSthresh to 65 535 bytes.
ii. The *TCP* output routine never sends more than the minimum of CWND and the receiver's advertised window.
iii. When congestion occurs (indicated by a timeout or the reception of duplicate ACKs), one-half of the current window size (the minimum of CWND and the receiver's advertised window, but at least two segments) is saved in SSthresh.
    - If the congestion is indicated by a timeout, CWND is set to one segment (i.e., slow start).
    - If not, this means it is duplicate ACK, then fast retransmit and fast recovery start.

iv. When new data is acknowledged by the other end, increase CWND, but the way it increases depends on whether *TCP* is performing slow start or congestion avoidance.

- If it was duplicate ACK, congestion avoidance carries on (after the 2 fast retransmit and recovery phases)
- If it was timeout indication, it goes as follows:

  If CWND is less than or equal to SSthresh, *TCP* is in slow start; slow start continues until *TCP* is halfway to where it was when congestion occurred (since it recorded half of the window size that caused the problem in step 2), and then congestion avoidance takes over. If not (CWND higher than the SSthresh) *TCP* performs the congestion avoidance phase.

Slow start has CWND begin at one segment, and be incremented by one segment every time an ACK is received. As mentioned earlier, this opens the window exponentially: send one segment, then two, then four, and so on. Congestion avoidance dictates that CWND be incremented by $MSS * \frac{MSS}{CWND}$ each time an ACK is received. This is a linear growth of CWND, compared to slow start's exponential growth. The increase in CWND should be at most one segment each round-trip time (regardless how many ACKs are received in that RTT), whereas slow start increments CWND by the number of ACKs received in a round-trip time.

(c) **Fast Retransmit**

Modifications to the congestion avoidance algorithm were proposed in 1990 ([Jac90]). Before describing the change, realize that *TCP* may generate an immediate acknowledgment (a duplicate ACK) when an out-of-order segment is received. This duplicate ACK should not be delayed. The purpose of this duplicate ACK is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected.

Since *TCP* does not know whether a duplicate ACK is caused by a lost segment or a reordering, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. *TCP* then performs a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire.

(d) **Fast Recovery**

After fast retransmit sends what appears to be the missing segment, congestion avoidance, and not slow start, is performed. This is the fast recovery algorithm. It is an improvement that allows high throughput under moderate congestion, especially for large windows.

The reason for not performing slow start in this case is that the receipt of the duplicate ACKs tells TCP more than just a packet has been lost. Since the receiver can only generate the duplicate ACK when another segment is received, that segment has left the network and is in the receiver's buffer. That is, there is still data flowing between the two ends, and *TCP* does not want to reduce the flow abruptly by going into slow start.

The fast retransmit and fast recovery algorithms are usually implemented together as follows:

i. When the third duplicate ACK in a row is received, set SSthresh to one-half of the current congestion window, CWND, but no less than two segments. Retransmit the missing segment. Set CWND to SSthresh plus 3 times the segment size. This inflates the congestion window by the number of segments that have left the network and which the other end has already received (3).

ii. Each time another duplicate ACK arrives, increment CWND by the segment size. This inflates the congestion window for the additional segment that has left the network. Transmit a packet, if allowed by the new value of CWND.

iii. When the ACK that acknowledges new data arrives, set CWND to SSthresh (the value set in step i). This ACK should be the acknowledgment of the retransmission from step 1, one round-trip time after the retransmission. Additionally, this ACK should acknowledge all the intermediate segments sent between the lost packet and the receipt of the first duplicate ACK. This step is congestion avoidance, since *TCP* is down to one-half the rate it was at when the packet was lost.

The fast retransmit algorithm first appeared in the 4.3BSD Tahoe release, and it was followed by slow start. The fast recovery algorithm appeared in the 4.3BSD Reno release.

2.2 User Datagram Protocol (*UDP*)

UDP (cf. [Pos80] and [Ste94]) is a really simple transport protocol that consists in sending as much data as needed for the using application without any timer or stuff used in *TCP*. It is a datagram oriented protocol, based like *TCP* on the *IP* layer. Every data delivered by the application generates a *UDP* datagram. This datagram is incorporated in an *IP* datagram as shown one Figure 2.8. If the *IP* datagram is too large for the network MTU, it will be fragmented by *IP*, multiple times if needed, through the whole network and reassembled only at the destination end system.

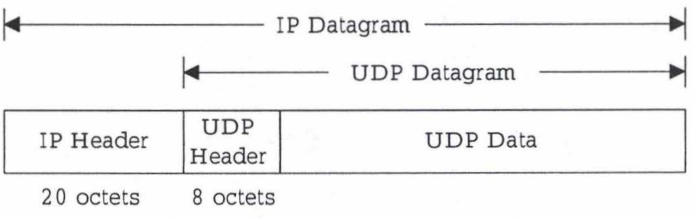


Figure 2.8: Structure of a UDP/IP datagram

UDP is not a reliable transport protocol: this means that it just sends the *UDP* datagram to the *IP* layer and does not manage any control or "following" concerning the sent data. This job is let to the application layer.

2.2.1 *UDP* packet structure

Figure 2.9 shows you the structure of a *UDP* datagram (header and data).

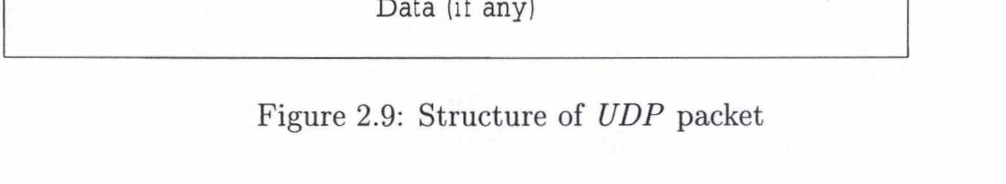


Figure 2.9: Structure of *UDP* packet

The fields:

- The *source and destination port numbers* are used to identify the corresponding processes.
- The *UDP length* cover the header length and the data length (redundant with the *IP* length field).
- The *checksum* includes its header and its data, but for *UDP*, the checksum is optional (the *IP* checksum just controls the *IP* header, so it does not cover *UDP*).

2.2.2 Characteristics

1) *UDP* checksum

Computation principle:

First the checksum field is set to 0. Then the *UDP* packet (header + data) is considered as a list of 16-bits length words. These words are summed and the complement is taken. The checksum field is then filled in with this complement.

Notes:

- The *UDP* datagram length may be an odd number of octets what is not allowed for the computation. *UDP* thus adds a "fake" octet (padding) at the end of the packet, an octet that will not be transmitted.
- Like TCP, *UDP* includes in its header a pseudo-header (12 octets) composed of certain fields of the *IP* header (depicted in Figure 2.10). They are used to compute the checksum and also to allow *UDP* to make a double control: to check if the data arrived at the good destination and also that *IP* did not give to *UDP* a datagram destined to a higher layer.
- As said before, the checksum is optional: if it is not used, the field is set to 0.
- If, during the check, the receiver detects a mistake with the checksum, the *UDP* datagram is destroyed silently (with no error message)

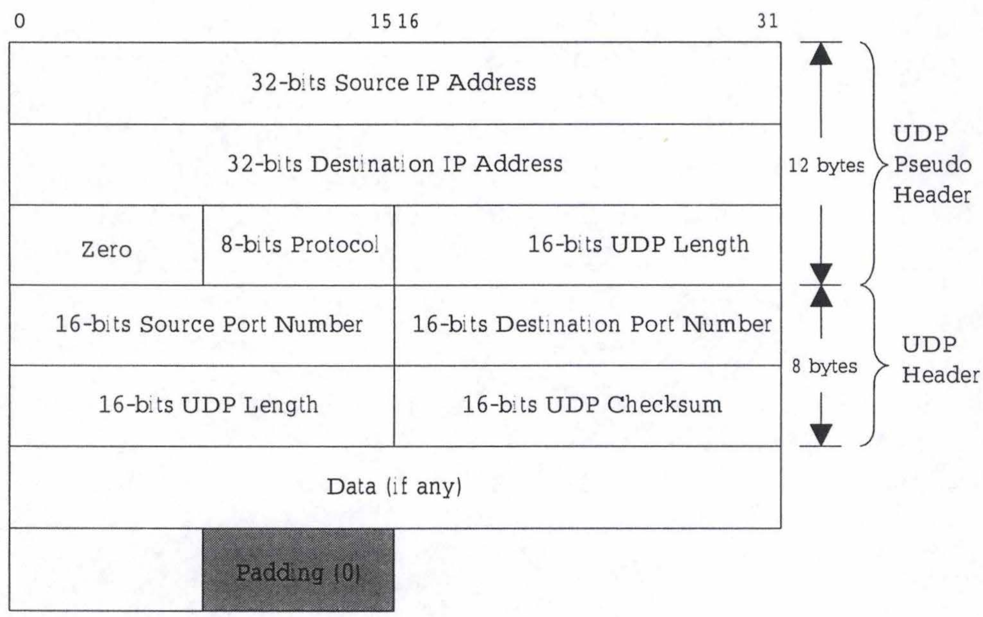


Figure 2.10: *UDP* complete packet for checksum

2) **Maximum size of *UDP* datagram**

Based on the *UDP* Length field of 16 bits, the maximum size should be 65 535 octets minus 20 octets for the *IP* header and minus 8 octets for the *UDP* header (left 65 507 octets of data). But in a practical way, this is not the case. One reason is that the applications may be limited in its accepted packet size. Nowadays, most systems offer a default maximum size of 8 192 octets. Another reason lies in the implementation of the TCP/IP kernel, which could limit *IP* packet lower than 65 535 octets to avoid fragmentation in the network.

2.3 Requirements for real-time streaming applications

Multimedia applications requirements are mainly based on three dimensions: the end-to-end delay, the packet loss ratio and the bandwidth (as represented on Figure 2.11).

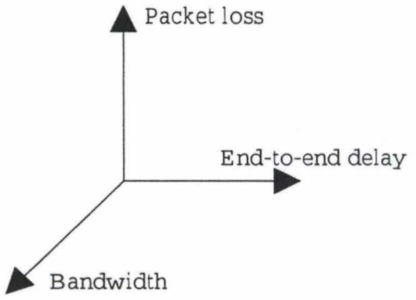


Figure 2.11: Multimedia applications requirements

Those three variables are highly correlated. The best (as for any applications but chiefly for multimedia ones) would be an infinite bandwidth with no packet loss and a zero end-to-end delay but in fact, this is never the case.

Usually multimedia applications try to minimize as much as possible the end-to-end delay, mainly when there are real-time interactions with human beings (*" for audio comfort"*). The required bandwidth can be really high, depending on the amount of data to be sent (video applications need far more bandwidth than audio ones) while they can easily survive to low packet loss ratio encountering a somewhat lower quality.

Let's take the example of the *Voice Over IP application*: two human beings are discussing through a network. This requires a really low end-to-end delay (about 150 to 200 ms because of the human tolerance) and a reasonable bandwidth but can deal with some losses.

2.4 Why *UDP* and not *TCP*

2.4.1 Why not *TCP* for multimedia applications

First of all, *TCP*'s usage of retransmissions mechanisms may cause large delays: when losses occur, not yet transmitted packets are delayed. Then the usually big size of *TCP* packets also introduces delays, waiting for the packet to be filled in (based on the Nagle algorithm [Nag84]). For a multimedia application, a late packet is a lost packet; you don't have the time to retransmit it. Finally, *TCP* does not support multicast what could limit the applications for the use they are designed for.

But the main reason is the type of transmission *TCP* is using which is, as mentioned before (cf. 2.1.2 Section 3), a *window-based transmission* (Figure 2.12). The transmission rate between the file containing the data and the *TCP* module can be considered as infinite related to the one between the *TCP* module and the outgoing link. So the *TCP* module receives packets at a high rate but may not send them over *b* immediately. The network or the destination determines when the packets have to be sent, what causes bufferisation and thus delay.

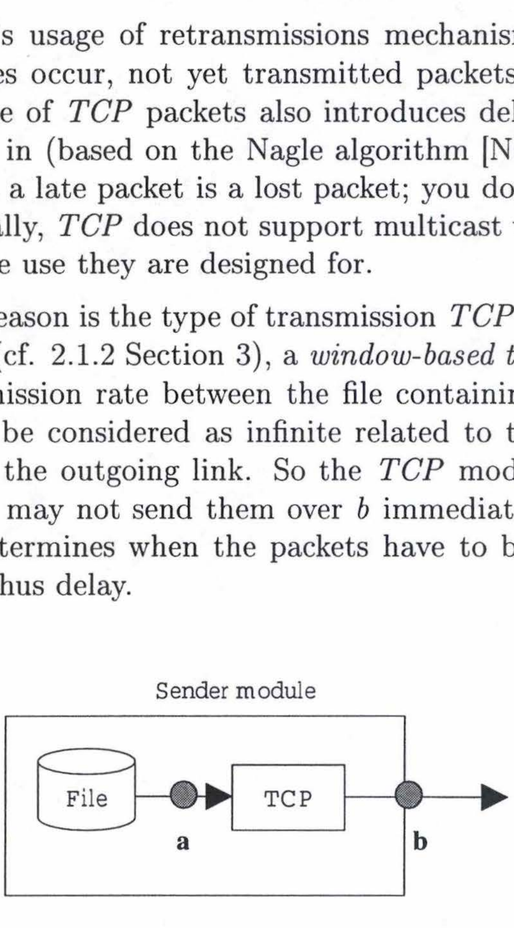


Figure 2.12: Window-based transmission

2.4.2 Why *UDP*

On the contrary, *UDP* does not use mechanisms like *TCP* (timeout and retransmission, ...). So it can provide minimum delay. Variable *UDP* packet size allows almost no delay before sending it (no need to wait it is full or a long timeout to expire). *UDP* also supports multicast, which is a major requirement for that kind of applications.

The main advantage of *UDP* is its *rate-based behaviour.* This type of transmission is not bursty, it is smoother than the window-based one, sending data as they arrive from the application. The data is sent as soon it is generated, without waiting for what ever, only based on the "application rate".

2.5 Remaining problems with *UDP*

Unfortunately *UDP* is not the ideal solution. Its simplicity, which was a powerful advantage, is turning into a serious drawback for multimedia applications. As mentioned before, *UDP* provides an unreliable connectionless service, based on the *IP* layer. This means that it cannot deal with packet loss (no guarantee about the correct data packet arrival), packet reordering (no guarantee about the packet sequence) and packet duplication. It also cannot recover from delay variations and furthermore, *UDP* is unable to distinguish medias and encoding.

To recover from almost all those limitations, the IETF decided to adopt a new protocol to work above the *UDP* layer in connection with the application layer: *Real-time Transport Protocol (RTP)* (cf. [ea96]). RTP alone is never used; its utility is only when "merged" with an application.

This protocol is composed of two sub-layers:

- ***RTP***, which deals with the flow of data packets. RTP provides the basic mechanisms needed by most multimedia applications (loss detection, reordering, duplication) and also offers some others functionalities.

- ***RTCP***, which controls the flow of data packets. The main goal concerns the quality of service and minimum of congestion control (far too weak): receivers send *RTCP* packets as low frequency acknowledgments to indicate the quality of reception and the sender to indicate the amount of information it has sent recently. *RTCP* is also used to provide more information about the sending application and to estimate, in case of multicast, the number of participants to limit the *RTCP* bandwidth.

RTP header

Figure 2.13 depicts the structure of *RTP* packets.

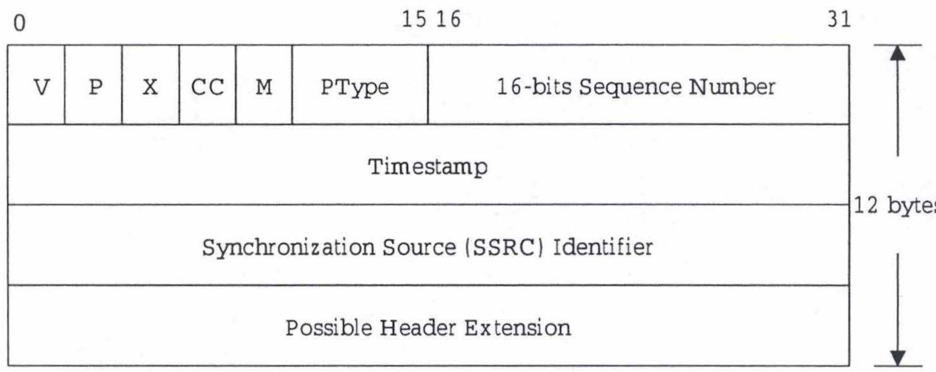


Figure 2.13: *RTP* header structure

The fields:

- The *sequence number* is used to reorder packets and to detect loss. It is also used to detect duplicated packets.
- The *timestamp* indicates when the packet has been generated, and is used combined with the sequence number to deal with the delay variations when silence suppression is in use.
- The *PType field* indicates the type (e.g. encoding) of audio/video data inside payload.
- The *SSRC field* identifies the source that created the packet.

But the main remaining problem stands in the absence of **congestion control mechanism**. Without this, no co-existence between *TCP* and non-TCP flows can be realized. So this is the domain where the next chapter takes place, introducing protocols trying to insure a fair sharing of the network resources between different kinds of flow.

Figure 2.14 shows how usually those mechanisms work together with the application, relying on the *UDP* layer combined with *RTP*. For multimedia applications, packets are sent over b when they arrive from a (almost the same rate).

1. The congestion control mechanism estimates b, sending rate of the output link based on the network load state,
2. b is then sent to the codec,
3. which adapts a such that a $\approx$ b so almost no packets are buffered $\Rightarrow$ no delay in the sender

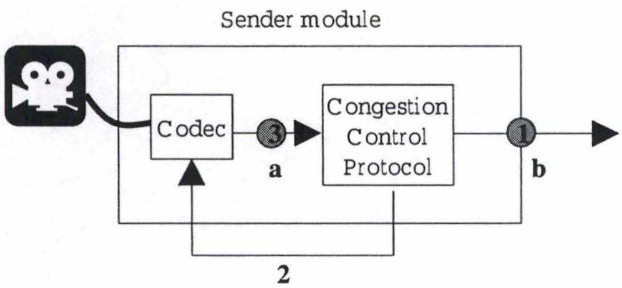


Figure 2.14: Rate-based transmission scheme

2.6 Conclusion

In this chapter, we have compared the way of working of the two mainly used transport protocols on the Internet nowadays: *TCP* and *UDP*.

Based on their transmission scheme and on the requirements of multimedia applications, *UDP*, enhanced with *RTP*, seemed to be the one that fit. *TCP*'s bursty transmission, abrupt and frequent wide rate fluctuations cause high delay and jiters, what is unacceptable for multimedia applications (audio and video can easily survive with limited losses but suffer from long delays).

But offering no congestion control mechanism, deploying uncontrolled traffic in a large scale might result in an extreme unfairness towards controlled flows like *TCP*. That's why mechanisms like we will see in the next two chapters are required.

Chapter 3

RAP: Rate Adaptive Protocol

This chapter describes the Rate Adaptive Protocol (*RAP*). We will try to see if this protocol is well behaved and TCP-Friendly when dealing with real-time streaming applications over best effort networks.

Designed to mimic *TCP*'s behaviour, it implements some mechanisms remembering the ones used in *TCP*. It first has to detect packet losses in different ways (based on timers or on duplicate ACK). Then, based on the kind of loss detection, it has to adapt its sending rate and "to re-start" the protocol in an appropriated way (cf. the four improving mechanisms for *TCP* in Section 3)

The goals of *RAP* are to ensure no starvation (*TCP* or *RAP*) by monopolizing the whole network resources and furthermore to guarantee a fair sharing of the bandwidth between all the sources.

Besides this, if the network uses features like Explicit Congestion Notification (ECN), we will mention how *RAP* could used those features to be more accurate in its adaptation scheme of the transmission rate.

In this chapter, *section 1* describes the way *RAP* is working. *Section 2* gives a complete description of both side of the *RAP* flow with their implementation and improving mechanisms. *Section 3* ends this chapter with the conclusions about *RAP*.

For the *RAP* source and destination modules implemented in OPNET simulator, see Appendix B. There you will find the complete structure of the used network for the simulations.

3.1 How does *RAP* work?

As an end-to-end congestion control mechanism, both sides of the "connection" have their own role. The most important part of the *RAP* mechanism lies at the source to keep the destination as simple as possible. The source sends packets with sequence numbers (identifier by flow) and keeps a table with records of information by sent packet. Each packet must be acknowledged by the destination. The destination first checks if there is a hole in the sequence number of the received packets using three static variables (note that a hole does not mean a loss, it may be a "slow" packet that still may come). It then updates its information and sends it back in the acknowledgment packet used by the source as feedback to detect losses. At the reception of such a packet, the source computes some variables to check in the records table the state of each sent packet. Based on this information, the *RAP* sender estimates the loss ratio and then the correct transmission rate.

By packet:

Step 1: *RAP* source sends a packet with a sequence number (identifier for that flow).

Step 2: *RAP* destination updates variables and sends feedback to the source.

Step 3: *RAP* source analyses feedback from destination and reacts appropriately.

3.2 Complete description of *RAP*

This section explains how the *RAP* protocol works. It describes the problems for the realisation of this protocol and the main mechanisms used to solve them.

Various options for the description are possible, the chosen approach is based on both sides of the "connection" instead of a sequential development of the protocol's working way, which would have been too abstract or confusing.

This section is structured as follows: first, a complete description of the *RAP* source including base concepts, the finite state machine representing the protocol, the behaviour when confronted or not to congestion and some particular points; then the description of the *RAP* destination. The implementation of each ends will follow and to conclude, some improving mechanisms.

3.2.1 The source

Concepts

1. **Inter-Packets Gap (IPG)**

For window-based protocols, like *TCP*, the transmission rate is a function of the sending window size. *RAP* does not perform a window-based rate control. It applies a rate-based rate control, which means that the transmission rate of the application is a function of the network's load but controlled by the amount of sent data and not by a window scheme of outgoing data's.

To control this transmission rate (depending specifically on the application), *RAP* manipulates the elapsed time between two consecutive sent packets. This is called the *Inter-Packets Gap,* IPG. By reducing the IPG, *RAP* increases the allowed sending rate for the application. Inversely, by increasing the time between two consecutive packets, *RAP* decreases the allowed transmission rate for the application. The application has to adapt its rate according to the information supplied by *RAP* about the network.

2. **Additive Increase / Multiplicative Decrease (AIMD)**

   The source performs an algorithm with working scheme of type Additive Increase / Multiplicative Decrease (AIMD) exactly like *TCP* does (cf. Section 2.6).

   - When there is no congestion indication, the source increases linearly the transmission rate periodically.
   - When congestion is detected (loss of packet), the source must decrease immediately the transmission rate by half.

   By *'Additive Increase'*, it means that while no congestion is undergone, the sending rate is increased by an amount X of bits per period. By *'Multiplicative Decrease'*, it means that when congestion is detected, the sending rate is divided by two.

   Note: terms 'increase', 'decrease' and 'periodically' still have to be explained. Be careful because *RAP* performs its control on the Inter-Packet Gap (IPG), when the algorithm says 'Additive Increase' (of the sending rate), this has to be translated in *RAP* by 'decrease' IPG. We have to do the same interpretation for Multiplicative 'Decrease' ($\rightarrow$ 'increase IPG').

Finite state machine:

Graph 3.1 depicts the finite state machine at the *RAP* source. In the first step, *RAP* initialises its general variables like the IPG, the SRTT, the first sequence number and different timers (in the **Init** state). Then, *RAP* enters in an idle state (**Idle**). There, multiple events can occur. First event, an ACK can be received ($\rightarrow$ **Ack** state). Based on the information held in the ACK, *RAP*updates its *history table*, detects if loss occurred and in case of it, adjusts its sending rate. It finally erases the now useless ACK. Second event, the IpgTimeout is triggered off ($\rightarrow$ **Ipg** state). So it's time to send a new packet but only after a negative loss check, otherwise *RAP* also has adjust its sending rate. Third event, the RttTimeout is triggered off ($\rightarrow$ **Rtt** state). One step of constant sending rate is over and it's time to start a new one with an higher sending rate (no loss during the last step, otherwise the RttTimeout would not have been triggered off). It then re-schedules the RttTimeout for the next step. Fourth event, a system message is received ($\rightarrow$ **End** state). Either it's the end simulation signal or an unknown event, both leading to the end of the simulation, closing the file and commenting the cause of this end.

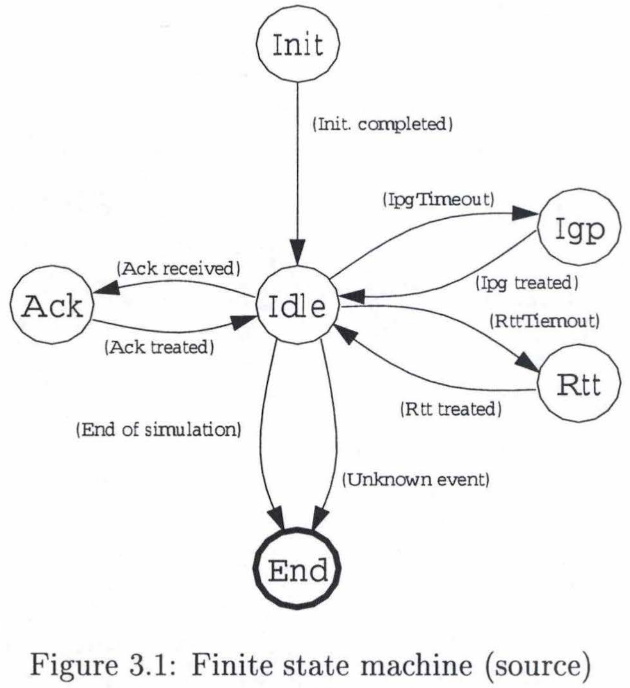


Figure 3.1: Finite state machine (source)

Note: The variables and procedures will be explained later.

For *RAP*, there is no real establishment of a connection (unlike for *TCP*) but let's call the flow between the sender and the receiver a connection. During a connection, congestion may be encountered or not. Depending on that, *RAP* has to react in an appropriate way. Let's examine the different situations.

First case: no congestion is detected.

In this case, the AIMD algorithm says that *RAP* has to periodically increase its sending rate.

The first question is: "How often do we have to increase the sending rate?", in other words: "How often do we have to change the IPG?" (the "periodically" term).

Ideally, if we had a perfect knowledge of the network capacity and its traffic load, we would be able to adjust the rate in a fair way and adopt a TCP-Friendly behaviour with the co-existing flows. Unfortunately, this is not possible.

For the end-to-end congestion control of *RAP* which is based on ACKs (without using features like ECN at the moment), all the information about the network and the destination is obtained at best after one round trip time. We will call the packets with this information "*feedback*". As mentioned in [MF97], it is suggested that adaptive schemes adjust their rate not more than one time per RTT. The reason is that RTT can be of random type. Using each RTT to change the rate could result in an inappropriate adaptation scheme (cf. [Bol92]). Indeed, if RTTs are consecutively high and low, the sending rate will have an unstable behaviour, which indicates that the adaptation scheme reacts hit by hit and not in response of the traffic load and the network in general (the required behaviour).

So, to have a stable frequency of the IPG re-computation, we have to smooth the gaps between consecutive RTT to get out the transient changes. Smooth RRT (SRTT) represents this stable frequency for re-computation and is called *a step* i.e. the period while the IPG stays unchanged. That's why we can say that *RAP* sends packets at a constant bit rate: fixed during a period. The SRTT is computed as follows to react smoothly to important variations of the RTT:

$$SRTT_{i+1} = \frac{7}{8}SRTT_i + \frac{1}{8}SampleRTT$$

Unless congestion is detected, when a step is over, *RAP* computes the new IPG. In this case (no congestion), it decreases the IPG to increase the transmission rate.

An advantage of using SRTT as a step for changing the IPG is that the packets sent during one step are likely to be acknowledged during the next step (SRTT sec after). It allows *RAP* to see how the network reacts to the previous adjustment of the rate before to compute what would be the best next rate.

The second question is: "In which way do we have to increase the sending rate?" (the "increase" term)

As said above, to increase the sending rate, *RAP* has to decrease its IPG. It is done based on this equation:

$$S_i = \frac{PkSize}{IPG_i}$$

$$\alpha = S_{i+1} - S_i = \frac{PkSize}{C}$$

where:

- S_i is the sending rate for the $step_i$,
- α is what we call the step height, the difference between two consecutive sending rates,
- C is a constant with the dimension of time,
- PkSize is the packet size.

The formula to compute the new IPG:

$S_{i+1} - S_i = \frac{PkSize}{C}$

$\Leftrightarrow$	$\frac{PkSize}{IPG_{i+1}} - \frac{PkSize}{IPG_i} = \frac{PkSize}{C}$	Replace S_i, S_{i+1}
$\Leftrightarrow$	$PkSize * \frac{IPG_i - IPG_{i+1}}{IPG_{i+1} * IPG_i} = \frac{PkSize}{C}$	Isolate PkSize
$\Leftrightarrow$	$\frac{IPG_i - IPG_{i+1}}{IPG_{i+1} * IPG_i} = \frac{1}{C}$	Divide by PkSize ($\neq 0$)
$\Leftrightarrow$	$C * (IPG_i - IPG_{i+1}) = IPG_{i+1} * IPG_i$	Multiply by C
$\Leftrightarrow$	$(C * IPG_i) - (C * IPG_{i+1}) = IPG_{i+1} * IPG_i$	Distribute C
$\Leftrightarrow$	$(IPG_{i+1} * IPG_i) + (C * IPG_{i+1}) = IPG_i * C$	Bring together IPG_{i+1}
$\Leftrightarrow$	$IPG_{i+1} * (IPG_i + C) = IPG_i * C$	Isolate IPG_{i+1}
$\Leftrightarrow$	$IPG_{i+1} = \frac{IPG_i * C}{IPG_i + C}$	IPG_{i+1} as a funct. of IPG_i

Now we have to assign the "good" value to C. The main goal of *RAP* is to mimic *TCP* (being TCP-Friendly), so let's try to do the same as *TCP*. In steady state, *TCP* increases its sending window by one packet every RTT seconds. Thus for *RAP*, we want one more packet to be sent each step (if no congestion) i.e. every SRTT seconds.

$$S_{i+1} - S_i = \frac{PkSize}{SRTT}$$

The sending rate will be increased by one packet every SRTT seconds (and thus C must be set equal to the step size i.e. SRTT). This gives:

$$IPG_{I+1} = \frac{IPG_I * SRTT}{IPG_I + SRTT}$$

Second case: congestion is detected.

In this case, the AIMD algorithm says that *RAP* has to immediately decrease its sending rate.

The first question is: "How to detect the congestion?"

RAP performs a loss-based rate control, which means that it relies on loss of packets to detect congestion and reacts appropriately. To achieve this, *RAP* source maintains a record for each sent packet. The set of records is called *transmission history* or *transmission table.* Each record contains the sequence number of the packet (identifier by flow), a flag that indicates the status of the packets (SENT, PURGED, INACTIVE) and the departure time. The sent flag means that the packet has been sent and that the source is waiting for the acknowledgement, the PURGED flag indicates that the corresponding packet has been acknowledged or recognised as lost and the INACTIVE flag will be explained more precisely in the improving mechanisms (cluster losses). In a few words, it is used to determine whether this packet was lost (SENT $\rightarrow$ PURGED) or this packet was in the transmission table while a loss occurred (SENT $\rightarrow$ INACTIVE), and thus is not considered as a lost packet.

The detection of packets loss can occur as a result of two events.

- The first one is the reception of an ACK. This situation will be explained in the section 'Improving mechanisms' (fast retransmit mechanism).

- The second one is when the IpgTimeout is triggered. The role of this interruption is either to allow *RAP* to transmit a new packet (no loss) or, if a loss has been detected, to react to this loss.

  Before sending a new packet (every IPG), the source computes the new timeout for the next step of transmission. This timeout is computed following the Jacobson/Karel's algorithm [1]. Based on this new timeout, the source goes through the whole *transmission table* to detect losses using the departure time of the packets. *RAP* compares the sum of departure time and timeout to the current time. In a single passage, it can detect multiple losses and reacts accurately according to it.

The second question is: "What does it have to do when congestion is detected?". In other words, "In which way do we have to decrease the sending rate?" (the 'decrease' term).

As said before, if congestion is detected, the source must immediately decrease its transmission rate. This is done by adjusting the Inter-Packets Gap (IPG). To multiplicatively decrease the rate in a *TCP* way, we just have to double the value of IPG. This has as effect that when loss occurs, the time between sending of two consecutive packets is doubled, so the amount of packets sent will be half of the amount before the detection of the congestion (just the way as *TCP*).

[1]Timeout = μ * SRTT + δ * VarSRTT where VarSRTT = variance of SRTT

$$S_{i+1} = \beta * S_i$$

where:

- S_i is the sending rate of the i^{th} period,
- $0 < \beta < 1$. (Default value: $\beta = 0.5$ to mimic *TCP*)

Some problems.

Problem 1: **Start-up phase.**

For long-term sessions, the start-up phase has no real importance; its influence is negligible which is not the case for short sessions. Anyway, they both have to probe the network to discover the available bandwidth and resources and to reach an equilibrium with the already existing sessions.

A slow probe (linear) of the bandwidth at the beginning of the session will have as effect a late use of the available resources of the network but a low loss of packets when the first congestion will be detected. In the opposite, a fast probe (exponential) will have as effect a fast acknowledgement of the available resources (and thus a fast utilisation of those resources) but a massive loss of packets (the way *TCP* is doing the probing).

The effect of the start-up phase is not studied in this document. It is assumed to be negligible compared to the length of the connections, which is typically in the order of minutes.

As default value, the start-up phase for every *RAP* flow consists of a sending rate of 40 kbps.

Problem 2: **Self-limiting Issues in *RAP*.**

In window-based rate control protocols, the source stops when the sending window is full of packets. It makes those protocols really stable, which is a researched characteristic, and easy to be implemented. It can be a little bit harder if, like *TCP*, the source allows the retransmission of lost packets but not too much. Unfortunately, for rate-based rate control protocols, it is not that easy because the sending rate is controlled by the computation of an appropriate Inter-Packets Gap. You never know exactly how many packets are outstanding (unless in the history table). There can be an arbitrary number of packets in the network with rate-based schemes, which is not the case with window-based schemes.

RAP's solution for the self-limiting problem is its timeout mechanism. In *RAP*, there are two timers: the IpgTimeout and the RttTimeout. With these two timers, *RAP* can deal with the limiting issue.

- The IpgTimeout represents the inter-packet gap. It is triggered when 'IPG seconds' have passed related to the last sent packet and thus indicates that the source may send another packet (unless loss has been detected). It is done by the function void IpgTimeout (void) (see Section 3.6). So every IPG seconds, *RAP* checks if loss occurred. If no loss occurred, *RAP* allows the sending of a new packet, but if loss(es) is (are) detected, *RAP* increases the IPG. This will have as effect to slow down the application's sending rate reacting to congestion.
- The RttTimeout represents the step while the IPG remains unchanged. When this timer is triggered, it is time to decrease the IPG thus to increase the transmission rate. It is done by the function void RttTimeout (void) (see Section 3.8). The decrease of the IPG is done unconditionally at each RttTimeout interruption and the function RttTimeout re-schedules a RTT interruption for IPG seconds after (starts the new step of constant IPG).

Note: if a loss is detected, the interruption scheduled by the RttTimeout function is cancelled because the IPG has been changed (cf. to AIMD), so a new step has started. Therefore, a new RTT interruption is scheduled.

Worst case: a link goes down. During the step of the crash, *RAP* will react at the loss of the outstanding packets. No ACKs are coming anymore but *RAP* sends one packet every IPG seconds. So, as explain before, *RAP*, based on the IPG timer, will check every IPG seconds if loss(es) occurred before trying to send any new packet. *RAP* will detect the loss(es) (timeout exceeded) and thus will decrease the transmission rate until the rate falls below the minimum rate tolerated by the application.

Common case: fair coexistence and TCP-Friendliness. Two timers configure the *RAP* protocol: one represents the time between two consecutive sent packets and the other the steps for re-computation of the IPG. In a normal way, the sending packet rate is in balance with the receiving ACK rate. If the traffic increases, the RTT will increase too. The SRTT will also increase and thus the step between re-computation of the IPG will be longer. If loss has been detected, the IPG will be doubled, decreasing thus the transmission rate and limiting the amount of outstanding packets. The balance is thus restored

3.2.2 The destination

The destination is the simplest side of a *RAP* link. First the finite state machine is described, then an explicit description of the goal for this side.

Finite state machine:

Grapf 3.2 depicts the finite state machine at the *RAP* destination. In the first step, *RAP* initialises the three variables used to detect packet loss (in the **Init** state). Then, *RAP* enters in an idle state (**Idle**), waiting for receiving a packet, the end signal of the simulation or an unknown event (also leading to the end of the simulation). In case of receiving a packet, *RAP* enters in the packet state (**Pack**) and performs its check algorithm to analyse the situation evolution with this new packets (*RAP* picks up the sequence number of the incoming packet, updates the three variables based on this number, generates and sends feedback to the sources then forwards the packet to the upper layer). The end state (**End**) just closes files and comments the cause of the end of the simulation.

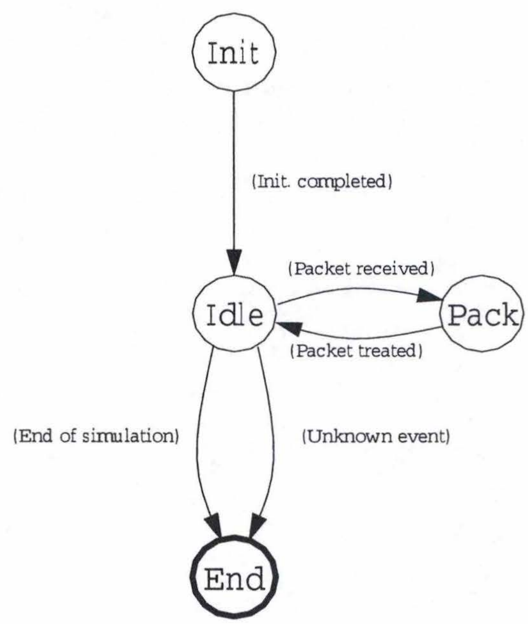


Figure 3.2: Finite state machine (destination)

Explicit description:

The destination has to deal with the sequence number of the arriving packet (seqNum) and three global variables per *RAP* connection:

- lastRecv (lr): sequence number of the latest packet received before seqNum,
- lastMiss (lm): sequence number of the latest packet not yet acknowledged before lastRecv (0 if no hole)

- precRecv (pr): sequence number of the latest packet received before lastMiss (0 if seqNum = 1)

These variables are used to inform the source about the received packet and possible holes. Only the arrival of the packets is important, not the order. Upon reception of a packet, the destination picks up the sequence number and then executes some comparisons to detect whether the packet creates a hole, fills in a hole, is in a hole but does not fill it in or is received in sequence (again the sequence is not important, an out-of-sequence packet only creates a temporary hole). All these information are then encapsulated in the feedback packets and sent back as an ACK for the received packet.

As you see, all the possibilities that could appear in a state are taken into account. It does not need anything else because the rest is done at the source side. Here are two examples of the feedback packet.

- The first one represents a common feedback packet (Graph 3.3),

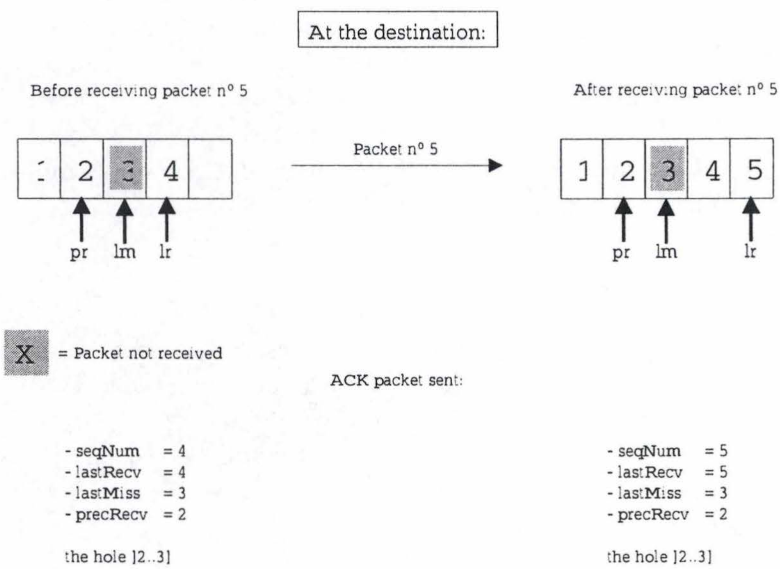


Figure 3.3: Feedback information

- The second one specifies an advantage of this feedback information (Graph3.4).

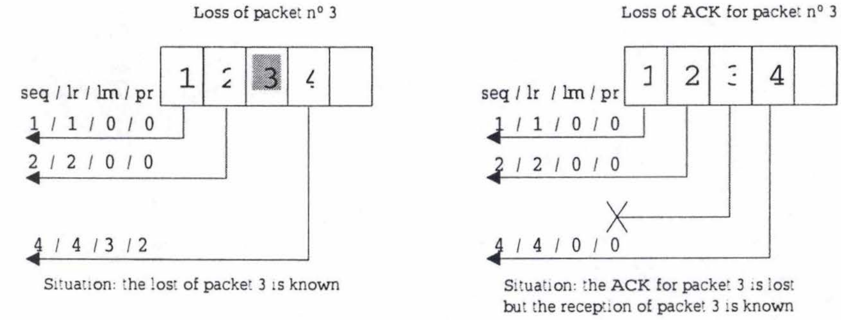


Figure 3.4: Feedback information advantage

These variables are used as feedback by the source and sent back in the ACK packet. It may seem that some information are redundant but they are all used depending on the case they represent. For example, seqNum is not always larger than lastRecv (receiving a late packet for instance). This kind of redundance has an advantage: the source can make the difference between the loss of an ACK and the loss of a packet. This is important because *RAP* performs a loss-based rate control so it has to know the difference: an ACK loss does not force the multiplicative decrease of the transmission rate like a packet loss would do.

3.2.3 The implementation

This section shows the made implementation choices to transcript the behaviour of this rate-based protocol into the C language and shows what was inevitable to make this code compatible and integrable in a modular way with the OPNET simulation tool. By "modular way", I mean that it could be reused afterwards by other people without having to modify anything (except the central parameters of the configuration that will be detailed in this text).

The source implementation:

The LossDetection function (Function 3.5) is triggered each time *RAP* has to check if a loss occurred (based on ACK information or on timer). It triggers the appropriate function and, in case of loss, indicates it to the calling function purges the useless packets of the history table (flag PURGED).

```
int LossDetection(int type, Packet* pkptr)
{  int numlosses;
   switch(type)
   {  case 0: // RAP_TIMER_BASED
              {numlosses = TimerLostPacket();}
              break;

      case 1: // RAP_ACK_BASED
              numlosses = AckLostPacket(pkptr);
              break;

      default: print{Wrong code used!!!};
   }
   Purge(1); // purge packets with PURGED flag
   return(numlosses);
}
```

Figure 3.5: LossDetection Function

The IpgTimeout function (Function 3.6) is triggered every IPG seconds. It checks if loss occurred. Then, either it allows a new packet to be sent (no loss), or it reacts to the loss calling the LossHandler function.

```
void IpgTimeout(void)
{
   double waitPeriod;
   if (LossDetection (timer-based))
   {
      LossHandler;
   }
   else
   {
      GenPacket();
   }
   if (finegrainused) {waitPeriod = ...;}
   // see improving mechanisms section
   // for the fine grain option
   else
   {
      waitPeriod = ipg;
   }
   op_intrpt_schedule_self(op_sim_time()+waitPeriod,0);
}
```

Figure 3.6: IpgTimeout function

The LossHandler function (Function 3.7) is either called from the IpgTimeout function for a timeout-based loss check or from the RecvAck function (at the reception of an ACK) for a loss check based on the ACK information.

```
void LossHandler (void)
{
    IncreaseIpg();
    for(int i = 0; i < eot; i++)
    {
        Flagi = INACTIVE;
    }
    op_ev_cancel(event);
    event=op_intrpt_schedule_self(op_sim_time()+srtt,1);
}
where eot is the end of the transmission table
    (the INACTIVE flag will be explained in the
    improvement section).
```

Figure 3.7: LossHandler function

Because the LossHandler function changes the IPG, a new step has to start so that *RAP* has to cancel the previous interruption and re-schedule a new one.

The RttTimeout function (Function 3.8) is scheduled every SRTT seconds to change the value of IPG from the LossHandler function or from the RttTimeout function itself. Note that the second argument of the schedule procedure is the code passed to know which function has to be called after a self-interrupt: Code 0 is for IpgTimeout, code 1 is for RttTimeout.

```
void RttTimeout (void)
{
    DecreaseIpg();
    event=op_intrpt_schedule_self(op_sim_time()+srtt,1);
}
```

Figure 3.8: RttTimeout function

Note: this interruption is stored in a global variable (event) to be able to cancel it if congestion is detected and IPG has been changed (see LossHandler function).

The UpdateTimeValues function (Function 3.9) is called at every received ACK. It computes the SRTT variable which determines the step length and the timeout variable used for the timeout-based loss check.

```
    void UpdateTimeValues(double sample)
    {
       double diff;
       if(initial)
       {
          frtt = xrtt = srtt = sample;
          variance = 0;
          initial = FALSE;
       }
       diff = sample - srtt;
       srtt = delta * srtt + (1 - delta) * sample;
       diff = (diff < 0) ? diff * -1: diff;
       variance += delta * (diff - variance);
       timeout = mu*srtt + phi*variance;
       if(finegrainused)
       {
          frtt=((1-KFRTT)*frtt)+(KFRTT*samplertt);
          xrtt=((1-KXRTT)*xrtt)+(KXRTT*samplertt);
          //cf. to fine grain improvement
       }
    }
Where:
    - xrtt and frtt are used in case of fine grain adaptation,
    - delta is usually set to 0.875% to limit the influence
      of the sample RTT on the srtt,
    - timeout used to detect loss in LossDetection function,
    - mu = 1.2 and phi = 4.0 in general to compute timeout.
```

Figure 3.9: UpdateTimeValues function

The DecreaseIpg function (Function 3.10) is called every SRTT seconds if no loss has been detected. It applies the formula described in Section 3.2.4.

```
    void DecreaseIpg (void)
    {
       ipg = (ipg * srtt) / (ipg + srtt);
    }
where srtt is Smooth \rtt{} and computed in the function
void UpdateTimeValue(double samplertt)
```

Figure 3.10: DecreaseIpg function

The IncreaseIpg function (Function 3.11) is called when loss has been detected. It doubles the IPG to cut in half the sending rate, the same way as *TCP* does.

```
    void IncreaseIpg (void)
    {
       ipg = ipg / beta;
    }
Where beta is set at 0.5.
```

Figure 3.11: IncreaseIpg function

The TimerLostPacket function (Function 3.12) is called every IPG seconds. It is the function used to detect loss before trying to send a new packet. It compares the sending time of every packets plus the newly computed timeout with the current time to estimate the state of the packets. If lost packet flag is set at PURGED, it indicates that the packet is no more needed in the transmission table (received or lost) and may be pulled out.

```
    int TimerLostPacket (void)
    {
       int numlosses = 0;
       for (i = 0; i < eot; i++)
       {
          if((departureTimei+timeout)<=currentTime)
          {
             if (flagi == SENT)
             {
                numlosses++; //Packet seqNumi is lost
             }
             flagi = PURGED;
          }
       }
       return(numlosses);
    }
where:
    - eot is end-of-table,
    - numlosses indicates to the calling function
       if loss(es) occurred (the number of lost packets),
       used as a boolean.
```

Figure 3.12: TimerLostPacket function

The destination implementation

The UpdateLastHole function (Function 3.13) is the only function at the destination side. It checks the sequence number of the incoming packet and uses it to compute the variables destined to be sent back in the feedback ACK.

```
Void UpdateLastHole (int seqNum)
{  if(seqNum==lastRecv+1) //Packet in sequence
   {  lastRecv=seqNum;
      return();
   }
   if(seqNum>lastRecv+1) //Loss(es) occurred
   {  prevRecv=lastRecv; //or re-ordered packets
      lastRecv=seqNum;
      lastMiss=seqNum-1;
      return();
   }
   if((lastMiss<seqNum)&&(seqNum<=lastRecv)) //Dup. pkt
   {  return();
   }
   if (seqNum==lastMiss)
   {  if (precRecv+1==seqNum) //Hole of one pkt filled in
      {  prevRecv=0;
         lastMiss=0;
      }
      else // Hole [n .. n+m]  (m>1)  to [n .. n+m-1]
      {  lastMiss--;
      }
      return();
   }
   if((prevRecv<seqNum)&&(seqNum<lastMiss)) //Pkt in hole
   {  prevRecv=seqNum;
      return();
   }
}
```

Figure 3.13: UpdateLastHole function

3.2.4 Improving mechanisms

To further mimic *TCP*, some mechanisms may be added to *RAP*. Let's introduce four of them.

First mechanism: *TCP*'s fast retransmit mechanism: duplicate acknowledgement.

As a loss-based rate controller, *RAP* needs to detect as soon as possible a packet loss. To achieve in that goal, we have already seen the timeout detection at the source. We also have seen the advantage of the information in the ACK packets from destination (ACK or packet loss). In addition, *RAP* may carry out an algorithm like the fast recovery mechanism of *TCP*. At each received ACK, *RAP* checks each record of the transmission table, searching for some packets too far behind from the lastRecv packet (in fact at most three sequence number behind). If it is the case and the status flag of those packets is SENT, *RAP* estimates that their ACK would arrived too late and considers the packets as lost. Function 3.14 depicts the function to be applied at each entry.

```
     int AckLostPacket (Packet* pkptr)
     {  int numlosses = 0;
        for("each entry seqi of the table")
        {  if(seqi <= lr)
           {  if((seqi > lm)&&(seqi <= pr))
              {  flagi = PURGED
              }
              else
              {  if((lr - seqi) >= 3)
                 {  if(flagi == SENT)
                    {  numlosses++;
                    }
                    flagi == PURGED;
                 }
              }
           }
        }
        return(numlosses);
     }
where:
     - pkptr is a pointer to the ACK packet,
     - Lr = lastRecv,
     - lm = lastMiss,
     - pr = prevRecv, each based on feedback packet,
     - seq_i is the seqNum of the entry checked in the table,
     - numlosses indicates if losses occurred.
```

Figure 3.14: AckLostPacket function

Second mechanism: Cluster losses

As described above, when there is no congestion indication, *RAP* periodically increases its transmission rate. "Periodically" has been defined as one time per SRTT (the most recent value of SRTT). So the IPG is updated only once per step (SRTT). When congestion is detected, *RAP* must immediately decrease its sending rate (it doubles the IPG). Congestion means loss of at least one packet of the outstanding packets but end systems should only react at a congestion situation and not at a single packet loss.

If a packet is lost during one step, *RAP* will react immediately but we will see the effect only at the next step. Thus it takes two steps to know if the reaction was appropriate. This shows that a good way to react to loss would be to only take into account the first detected loss during one step and to consider the others in the same step to be due to the same congestion event. *RAP* would only decrease one time per step the IPG in case of loss.

We already talked about the INACTIVE flag (see section source, second case). There are two ways to detect losses: when an ACK arrives (information carried inside) and when an IPG timeout happens. In these two cases, *RAP* will trigger off the LossHandler function (Function 3.7). This function increases the IPG (because of loss), re-schedules an RTT interruption but before it, puts to INACTIVE all the outstanding packets. This will have as effect that if another ACK arrives during the step indicating another packet loss, this loss will not be taken into account because the flag for the 'missing' packet has been set to INACTIVE and thus the IPG will not be increase. That was the goal for the cluster losses: to react only one time per step at loss.

Third mechanism: Fine grain

The fine grain adaptation scheme tries to mimic further the ACK-clocking based congestion avoidance while the coarse grain scheme still performs an AIMD algorithm. The goal of this new feature is to make *RAP* more responsive to transient congestion events (a short-term exponential moving average of the RTT captures short-term trends congestion). FRTTi is the short-term exponential moving average of RTT sample and XRTTi the long term one.

There are two ways to perform the fine grain mechanism: per step or per ACK adaptation.

- The first way gives a higher importance to the more recent RTT sample because it is supposed to be the most representative of the congestion situation of the network. At the beginning of the i^{th} step, first the new IPG is computed (eq.: ipg' = (ipg * rtt)/(ipg + rtt)), and then the fine grain feedback is used like:

$$ipg'' = ipg' * feedback_i$$

where $feedback_i = \frac{FRTT_i}{XRTT_i}$

- The second way performs a finer granularity mechanism for rate adjustment. At each ACK, FRTT and XRTT are update. In UpdateTimeValues function, they are computed as follow:

  - FRTT = ((1 - Kfrtt) * FRTT) + (KFRTT * samplertt)
  - XRTT = ((1 - Kxrtt) * XRTT) + (KXRTT * samplertt)

  where Kxrtt = 0.01 and Kfrtt = 0.9 to be able to capture the short-term congestion state since the last ACK. (FRTT, short-term, gives a higher weight to the samplertt, which represents the more recent computed RTT while XRTT, long-term, gives a higher weight to its last value). In the IpgTimeout function, the IPG for the next step is computed like in the first way.

- $IPG = \frac{FRTT}{XRTT} * IPG$

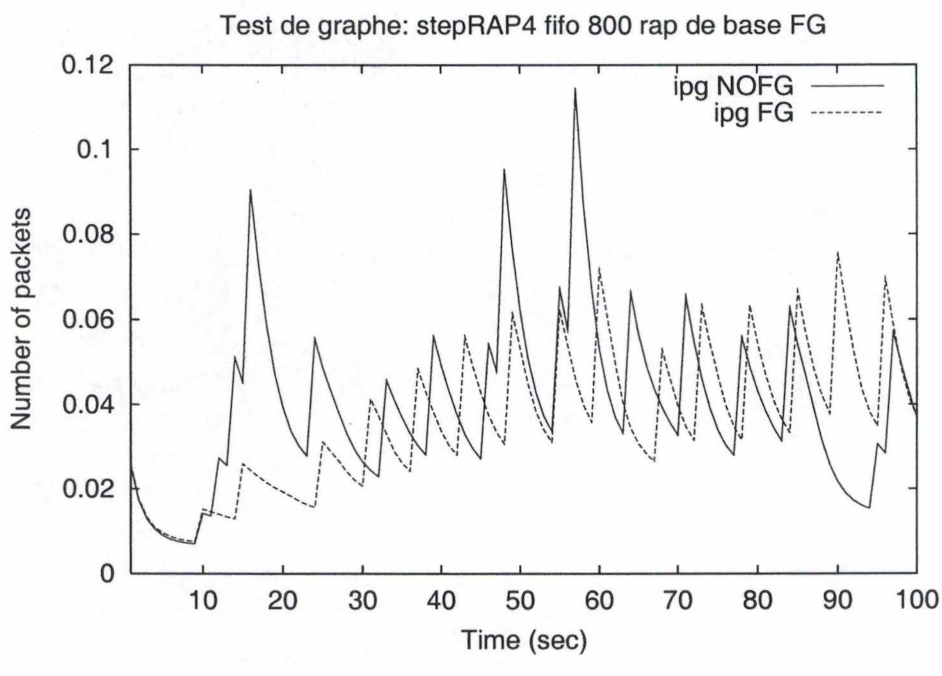


Figure 3.15: Fine grain smoothing effect

For the rest of the thesis, each simulation will be using the fine grain option. In case not, it will be clearly indicated.

Fourth mechanism: Explicit Congestion Notification: ECN

This is an option available on some networks, marking overflow packets instead of dropping them. It would be like cluster losses. This mechanism avoids waiting for the retransmission timeout and behave like the three duplicate ACK mechanism. It is the third way to detect loss and the reaction based on ECN is in the same way: just puts INACTIVE to all the outstanding packets to react only one time per step.

3.3 Conclusion

In this chapter, we fully describe a congestion control mechanism called *RAP* (Rate Adaptive protocol). Dedicated to real-time multimedia applications, it is designed to overpass the *UDP* aggressiveness, responsible of *TCP* flows starvation due to its lack of congestion control mechanism. Furthermore, *RAP* is supposed to be TCP-Friendly, i.e. to ensure a fair sharing of the network resources (cf. Chapter 5 for confirmation).

RAP performs a sending rate policy based on the AIMD algorithm (increasing linearly and decreasing multiplicatively its sending rate based on the met congestion). The working time of *RAP* is partitioned in steps during the ones the sending rate stays unchanged. The steps are computed based on a moving average of the RTT and the sending rate is performed by increasing or decreasing the time between two consecutive packets. The packet loss is detected either by a timer or a mechanism of hole detection; imitating the way *TCP* does it with its timer and three duplicate ACK scheme. It also implements some improving mechanisms: the clustered losses mechanism (reacting one time per step to loss) just like *TCP*, a sending rate variation smoother (fine grain) and may use network features like ECN.

Chapter 4

Other mechanisms

In this chapter, four mechanisms of congestion control will be introduced with some having very different ways of working. Notice that the first mechanism, *TFRC*, was the subject of an IEEE draft on the 17/11/2000 (end of the training course). This shows how important the subject is, very relevant at the moment. The next two are mainly informational and provided for comparison purposes only. Let's just mention here *RAP*, which is the center of my thesis, that will be fully describe in Chapter 3.

All of the following mechanisms are congestion control mechanisms based on adapting the sender transmission rate in accordance with the network congestion state. Based on feedback and complementary information, the sender would increase its transmission rate during underload situations and reduce it otherwise. Such way of working does not guarantee any QoS but the quality for the users is improved thanks to the loss reduction and to the increasing used bandwidth when available. Designed in a *TCP* similar fashion, they prevent the starvation of *TCP* connections and allow a stable transmission behaviour.

4.1 *TFRC*: TCP-Friendly Rate Control

TFRC (cf. [SFW00]), from Sally Floyd, Mark Handley and Jitendra Padhye, is a congestion control mechanism for unicast flows (which can be extended to support multicast) over a best effort Internet network. Its way of working is similar to *RAP* Indeed, *TFRC* is also based on the throughput estimation equation of *TCP*, related to the round trip time, the loss-event rate, ... it mimics the congestion control mechanism of *TCP* and adapts its sending rate to maintain a fair concurrence between co-existing flows.

4.1.1 General way of working

Principle:

Step 1: The receiver measures the loss event rate and sends it back to the sender (in feedback packets).

Step 2: The sender uses these feedback packets to measure the round trip time.

Step 3: Using the computed loss event rate, the round trip time and based on the *TCP* throughput equation, the sender identifies the acceptable transmission rate and matches its sending rate on it.

At the sender side, during a "period", the source sends packets at a fixed rate (initialised at one packet per second). When receiving a feedback packet, the source analyses the information carried inside, computes a new estimation of the round trip time and computes the new appropriated rate based on this new round trip time (increasing or decreasing the sending rate). If no feedback packet is received during a period of two round trip times or before the NoFeedBack timer (initialised at 2 seconds), the sending rate is cut in half.

At the receiver side, feedback packets are periodically sent to the sender, at least once per round trip time. If the sender has a really low sending rate (less than one packet per round trip time), a feedback packet should be sent for each data packet received. A feedback packet is also sent every new loss event. When receiving a data packet, the receiver introduces it in a data structure, computes the loss event rate, and if a new loss event is detected, a feedback packet is sent.

4.1.2 Major concept

Throughput computation equation

The *TCP* throughput equation (cf. [JPK98]), on which the *TFRC* algorithm is based, is characterized as a function of loss rate and round trip time for a bulk transfer *TCP* flow (i.e. with an unlimited amount of data to send) taking into account the fast retransmit mechanism and also the timeout effect on the throughput.

$$X = \frac{S}{R * \sqrt{\frac{2P}{3}} + (t\_RTO * min(1, 3\sqrt{\frac{3P}{8}}) * P * (1 + 32P^2))}$$

where

- X is the sending rate in bytes / second,
- S is the packet size in bytes,
- R is the round trip time in seconds,
- P is the packet loss ratio ([0 .. 1.0], i.e. the fraction of transmitted packets that are dropped in the network),
- t_RTO is the *TCP* retransmission timeout value in seconds.

The loss detection

Using the *TCP* throughput equation, *TFRC* uses a more sophisticated method to gather the necessary parameters.

The computation of the loss rate is performed at the receiver based on the detection of lost packets from the sequence numbers of arriving packets. Each packet has its own sequence number, which is incremented by one for every packet sent. It means that if a packet has to be retransmitted, its sequence number will not be the same as the first time (unless the transport protocol requires the retransmitted packet to have its first number).

Keeping track of the arrived and missing packets, a packet is considered as missing if at least three packets with a higher sequence number have arrived (almost in the same way as *TCP*). This scheme has the advantage to leave some flexibility for reordering packets. More of that, the late packet can fill the hole in the data structure and the receiver can re-compute the loss ratio.

To be robust to several consecutive packets lost, we have to point out a difference between *loss event* and *lost packet*: a loss event may include several lost packets but each lost packet does not mean a loss event. Each lost packet detected during one RTT is considered to belong to the same loss event (like *TCP* reacting once per RTT). The measurement of the RTT is done by the sender and is piggy-backed onto a data packet. Based on it, the receiver knows if a lost packet starts a new loss event or still belongs to the previous one.

To compute the loss event ratio P: first we have to compute the average loss interval, using the n more recent loss event interval weighted such that the recent ones influence more than the old ones:

```
if (i < n/2)
   then w_i = 1.0;
   else w_i = 1 - (i - (n/2 -1)) / (n/2 + 1;)
```

Number n is the key parameter to the accuracy and the speed of reaction of *TFRC*. Based on this, the average loss interval (I_mean) is computed (cf. [SFP00] Section 5.4) to finally obtain the loss event ratio P: $P = \frac{1}{I\_mean}$.

4.1.3 Structure of exchanged packets

Data packets

Figure 4.1 depicts the structure of *TFRC* packets sent by the sender.

- Seq.num. is the sequence number of the sent packet,

- Dep. time is the departure time of the packets in milliseconds,

Seq. num.	Dep. time	ERTT	Trans. rate	data

Figure 4.1: structure of *TFRC* data packet

- ERTT is the current estimation of the round trip time in milliseconds, used to know when feedback packets have to be sent (combined with the Trans. rate field)

- Trans. rate is the current transmission rate,

- Data is the packet coming from the upper layer.

Feedback packets

Figure 4.2 depicts the structure of the acknowledgement packet received by the sender.

Last recv.	Delay	Recv. rate	Estim. Loss rate

Figure 4.2: structure of *TFRC* ACK packet

- Last recv. is the departure time of the last received packet,

- Delay is elapsed time between the last received packet and the generation of this feedback report,

- Recv. rate is the estimated rate for the receiver of the data since the last feedback report was sent,

- Estim. loss rate is the receiver's current estimation of loss events.

4.2 *LDA+*: Loss Delay Adjustment +

LDA (cf. [SS99]) and its latter version *LDA+* (cf. [SW00b]) are end-to-end rate adaptation algorithm achieving AIMD algorithm and relying on the Real-Time Transport Protocol (*RTP*) for feedback information. Furthermore, some added functionalities of *RTP* are used to determine the bottleneck bandwidth and then, according to this bottleneck bandwidth, *LDA+* dynamically determines the adaptation parameters (mainly based on losses, delays and capacity observed on the used path).

LDA+ is a "QoS" control mechanism based on adapting the sender transmission rate in accordance to the network congestion state. Based on the feedback from the receiver (*RTP*), the sender would increase its transmission rate during underload situations and reduce it otherwise. This way of working does not guarantee any QoS but the quality for the users is improved thanks to the loss reduction. Designed in a *TCP* similar fashion, *LDA+* prevents the starvation of *TCP* connections but still allows a stable transmission behaviour. Made first for unicast flows, a new version, *MLDA* (cf. [SW00a]) has been made to support the multicast transmission.

4.2.1 General way of working

Principle:

Step 1: The sender initiates the probe phase to discover the bottleneck bandwidth.

Step 2: The receiver computes the bottleneck bandwidth and sends feedback about the received data and the charge of the network.

Step 3: The sender, based on the feedback information, computes the new appropriate rate.

At the sender side, the sender initiates the probe phase to estimate the bottleneck bandwidth. Based on the information of the feedback packets, notably the estimate bottleneck bandwidth, the sender calculates the RTT with the arrival time (t) of the packets: $RTT = t - t_{DLSR} - t_{LSR}$ where t_LSR is the timestamp of the last received sender report and t_DLSR, the time elapsed between receiving the last sender report and sending the receiver report. The round trip time propagation delay (τ) is the smallest RTT. Adding this RTT, the sender computes the new appropriate transmission rate.

At the receiver side, enhanced RTP offers the ability to estimate the bottleneck bandwidth of a connection based on the packet pair approach described by Bolot (cf. [Bol92]). The essential idea behind this approach is: "If two packets can be caused to travel together such that they are queued as a pair at the bottleneck, with no packets intervening between them, then the inter-packet spacing will be proportional to the time required for the bottleneck router to process the second packet of the pair". The bottleneck bandwidth (b) is calculated as:

$$b = \frac{probepacketsize}{gapbetween2probepackets}$$

To estimate the average bottleneck bandwidth, *LDA+* rely on the BPROBE tool ([CC96]), clustering similar estimates into intervals, and choosing the average of the interval with the highest number of estimates. The estimated value is then sent back to the sender with the next receiver report.

4.2.2 Major concept

Dynamic determination of the Additive Increase Rate (AIR)

The increase and decrease factors for AIMD scheme are dynamically adjusted to the network conditions:

- The amount of additive increase (AIR) is determined to ensure that:

  1) flows with a low bandwidth can increase their rate faster than flows with a higher bandwidth,
  2) flows do not exceed the estimated bottleneck bandwidth,
  3) flows do not increase their bandwidth faster than a *TCP* connection.

  AIR is set first to a small value (often 10 Kb/s) and is then increased during periods of no losses. So if no loss is detected, the sender computes the AIR' for the next period as follow: $AIR' = AIR + AIR * B_f$ with $B_f = 1 - \frac{r}{b}$ where r is the current rate and b the estimated bottleneck bandwidth. The B_f factor is used to allow connections with low bandwidth share to use larger AIR values and thereby converge faster to their fair bandwidth share. The new rate r' is then set to: $r' = r + AIR'$

- In case of loss detection, the transmission rate r in decreased based on the decrease factor (R_f), proportional to the indicated losses (l) as follow: $r' = r * (1 - (l * R_f))$ but never under the value given by the *TCP* throughput equation. R_f (usually set between 2 and 5) represents the degree of reaction due to losses. A high value results in a fast reduction of the transmission rate but a more oscillatory behaviour. A low value, on the other hand, leads to a more stable rate but a longer convergence time.

4.2.3 Structure of exchanged packet

Data packets

Figure 4.3 depicts the structure of *RTCP* packets enhanced for *LDA+* sent by the sender.

Src. seq. num.	SEQ	n	

Figure 4.3: Structure of *LDA+* data packet

where

- Src. seq. num. is the source sequence number of the sent packet,
- SEQ is the sequence number of the packet that starts the stream of probe packets,
- n is amount of probe packets that will be sent,
- ... is the typical *RTCP* packet information.

Feedback packets

Figure 4.4 depicts the structure of feedback packets for *RTCP* enhanced for *LDA+* received by the sender.

Frac. loss.	t_{LSR}	t_{DLSR}	Estim. Bandwidth	

Figure 4.4: structure of *LDA+* ACK packet

where

- Frac. loss is the fraction of lost data,
- t_{LSR} is the timestamp of the last received sender report,
- t_{DLSR} is the time elapsed in between receiving the last sender report and sending the receiver report,
- Estim. bandwidth is the estimated bottleneck bandwidth by the receiver,
- ... is the typical *RTCP* packet information.

4.3 *TEAR*: TCP-Friendly Emulation At Receiver

TEAR (cf. [IRY00]), from Injong Rhee, Volkan Ozdemir and Yung Yi, is a new flow control approach for congestion control mechanism for unicast flow. Indeed, *TEAR* shifts most of flow control mechanisms to receivers. The receiver does not send to the sender the congestion signals (packet arrivals, packet losses, timeout, ...) detected in the forward path but rather processes them immediately to determine the appropriate transmission rate. Using the network congestion signals and using a congestion window (just as *TCP*), the receiver can emulate the *TCP* sender's flow control functions. It estimates thus the TCP-Friendly rate (reactions of *TCP*) for the congestion conditions observed in the forward path, smoothes the estimated values of steady-state *TCP*'s throughput by filtering the noise and finally reflects it to the sender.

The big advantage of this mechanism is that for asymmetric networks, such as wireless networks, cable modems and satellite networks, transmitting feedback for (almost) every packet received (as it is "done" in *TCP*) is not very attractive because of the lack of bandwidth on the reverse links. Thus packet losses and delays occurring in the reverse paths severely degrade the performance of round trip based protocol (*TCP*), resulting in reduced bandwidth utilization, ...

4.3.1 General way of working

Principle:

Step 1: The receiver measures losses, delays and keeps track of the arrived packets. It computes the "*TCP* fair throughput" then sends it back to the sender (in periodic feedback packets).

Step 2: The sender uses this feedback to adjusts its transmission rate.

At the sender side, the sender just adjusts its sending rate to the rate forwarded by the receiver.

At the receiver side, the *TEAR* protocol behaves almost like *TCP*: slow start, congestion avoidance, fast recovery, timeout phases correspond to *TCP*'s features (+ window emulation of *TCP*).

The difference lies in the management of the CWND at the receiver. CWND is initialised to 1 packet and the SSthresh is set to a default value (larger than 2). When a packet is received in sequence, CWND is incremented by 1 if in slow start phase, by $\frac{1}{lastCWND}$ if in congestion avoidance phase (just like *TCP*). At the beginning of each round (see next point), last CWND is updated and used to compute the next round's increment. When the protocol is in slow start phase and the CWND is larger than the SSthresh, the protocol skips to congestion avoidance phase.

4.3.2 Major concepts

Rate independence

The probability of having a packet loss within a window of x consecutively transmitted packets does not depend on their transmission rate. In today's Internet, packets are dropped from routers indiscriminately of the transmission rate of the flows when routers lack of buffer but because of the prevailing of tail-drop queuing management, packet losses are highly correlated.

To decrease this correlation, *TEAR* treats the losses likely correlated as a loss event, in the same way as *TCP* with its congestion window. Under such operating conditions, rate independence can be generally assumed.

Round

As *TCP* with its congestion window (CWND) that indicates the number of packets in transit, *TEAR* maintains also a variable but at the receiver this time and updates it according to the same algorithm based on the arrival of packets.

A transmission session is partitioned into non-overlapping time period, called *round*. A round contains roughly an arrival of packets from the congestion window. In *TCP*, a "round" is recognized at the sender when an acknowledgment packet is received for the reception of packets in the current congestion window whereas in *TEAR* the receiver can recognize a round when receiving packets.

As you can see, the *TEAR* rounds depend on the transmission rate unlike *TCP*. This difference may cause CWND to be updated at different times: every round for *TEAR* instead of every RTT for *TCP*. To account for this discrepancy, *TEAR* estimates the *TCP* throughput by assigning a fictitious RTT to each round. When estimating the transmission rate during one round, *TEAR* divides the current value of CWND by the current estimate of *TCP* instead of the real-time duration of the round. The *TEAR* receiver estimates the *TCP* throughput by taking a long-term weighted average of these per-round rates and reports it to the sender. The sender adjusts its rate to the reported rate.

Rate computation

TEAR follows the typical behaviour: Additive Increase / Multiplicative Decrease, characteristic sawtooth pattern of the transmission rate. Although instantaneous rates would be highly oscillating, long-term throughput would be fairly stable. So the idea is to set the *TEAR* transmission rate to an average rate over some long-term period T (called *epoch*).

At the end of each round, the receiver divides the sum of all the CWND samples recorded in the current epoch by the sum of the RTT recorded in that epoch. The result is called *rate sample* of this epoch. At the end of each epoch, the rate is set to the most recent rate sample, which gives a smoother rate adjustment. But because of the noise, the algorithm includes more than just the current epoch. Introducing some weighted average over rate samples, the previous computation are taken into account to try to consider only reliable samples, large enough epochs, ...

Feedback: the sender sets its transmission rate to the most recently received rate from the receiver. If the most recent computed transmission rate is lower than the previous reported one, the receiver reports it immediately to the sender. Other way, the receiver will send its rate at the end of a feedback round.

4.3.3 Structure exchanged of packet

The packets structure is not different of *TCP* ones; the only difference stands in the feedback packet indicating the computed "fair" transmission rate.

	RAP	*TFRC*	*LDA+*	*TEAR*
Communication	Unicast	Unicast*	Unicast*	Multicast
Adaptation	States	*TCP* equation	Bottleneck	Window emul.
Complexity	Low	Medium	High	Low
feedback	Each packet	Own periodic	Enhanced *RTP*	Own periodic
Rate	Sawtooth	smooth	Sawtooth	smooth

Table 4.1: Characteristics of the presented mechanisms

4.4 Conclusion

In this chapter, we introduced three other TCP-Friendly congestion control mechanism working with different ways. The choice of a congestion control mechanism depends on the task to do, the network characteristics and the traffic requirements of the sending application. On controlled or closed environments, like a company's intranet, we can use the one we want even if we have to change the network infrastructure. But for a global deployment on the Internet, the task is high time consuming and very costly. The choice is not easy. Indeed, such solutions are likely to be used only if they offer vastly improved performance over solutions that can be used with today's Internet infrastructure. The deal is to find a good mix between difficulties and benefits.

All the introduced mechanisms are end-to-end protocols, being completely implemented in the end system without any additional features in the routers. To mimic *TCP* furthermore, they have to suffer from high RTT variations. They all perform a rate-based congestion control but compute differently their adaptation. Table 4.1 shows the main characteristics of those protocols.

All performing he AIMD scheme, *TFRC* computes the increase of its sending rate based on the *TCP* throughput equation and cut in half when losses are detected; *LDA+* does not cut in half its sending rate in case of congestion, it computes an value (positive or negative) to add to the sending rate based on the bottleneck bandwidth and the proportion of loss. *TEAR* uses a window to emulate the *TCP* reactions and sends back to the source the computed rate based on it. *RAP* follows the AIMD scheme: cutting in half its ending rate when congestion and increasing its rate depending on its current state (like *TCP*).

For the kind of communication, *TEAR*, *TFRC* and *LDA+* can be extended to multicast communication (*). *RAP*, acknowledging every received packets, could not deal with the amount of generated packets in response from all the destinations.

For he feedback information, *TFRC* and *TEAR* are working by themselves; they rely on periodic feedback reports generated by the receiver based on their current RTT and transmission rate. *LDA+* relies on an enhanced *RTP* to ensure periodic feedback information over the network. *RAP*, like *TCP*, is based on the explicit acknowledgements of the received packets (with some options for *TCP*).

In fact, *RAP* looked to be a good mix between complexity and improvement of *UDP*, trying to conciliate the smoothness of adaptation scheme preferred for multimedia applications and a competing but fair aggressiveness towards the other kinds of flows for the network resources.

Chapter 5

Simulations

The goals of *RAP* is to ensure neither *TCP* nor *RAP* to be able to monopolize the whole bandwidth and furthermore to guarantee a fair sharing of the network resources between all the sources.

Over the single bottleneck configuration scenario, by modification of central parameters, we will try to see if the *RAP* protocol is a well behaved and TCP-Friendly protocol dealing with real-time multimedia applications over best effort networks.

In this chapter, we focus on the behaviour of *RAP*, compared with *TCP* (as the base case), first by considering only *RAP* flows, then confronted with *TCP* flows over a best effort network with routers performing FIFO or RED queuing management. We will end this chapter by some comparisons between simulations. One of the goals of these comparisons is to show the influence of different kinds of queuing discipline (in fact FIFO and RED) on the *TCP* (base case) and *RAP* transmission scheme, achieving fairness or not. Another goal is to determine the variability of the protocols confronted to different modifications (packets size and increased RTT).

5.1 Single bottleneck topology

The topology used for the single bottleneck scenario is depicted in Figure 5.1. It consists in a single shared link between five greedy sources, sending an infinite amount of data while trying to avoid collapse and starvation. The parameters used for *RAP* and *TCP* simulations are summarized in the table 5.1. Specific values will be indicated in case of changes. It should be noted that:

1. The buffer sizes in the routers are chosen based on the data packet size to congestion both *RAP* and *TCP* sources approximately at the same level when evaluated separately. Too large buffers could have led to manipulate enormous data, useless for the simulations. The chosen values allowed each

flows to enqueued at least few packets (7 or 8) before entering in congestion phase. The link bandwidth ensures the 10 msec of transmission time for a packet on the bottleneck link.

2. The RED version implemented and used for the simulation is described as RED_4 in [CE99], and takes into account the packet size. Uniformly dropping packets, long packets will be more likely dropped than small ones.

3. Related to the small number of sources, each flow entering in congestion and reducing its sending rate frees a quite large part of the network resources, immediately used by the other flows. This is one reason of some oscillating behaviour we will see.

4. The represented data on the graphs correspond to the sent volume of KBytes computed at the sources (curves) and the behaviour of the queuing discipline (histogram), thus including for *TCP* flows the retransmitted lost packets and the control packets (SYN, ACK, ...) which are smaller than data packets. The values in the tables are more accurate and take into account the different packets sizes of *TCP* for the throughput and the standard deviation computations.

5. The sequential start of the flows means that the simulator starts each flow with a random elapsed time between them to avoid phase effect at the beginning. The order is also randomly chosen.

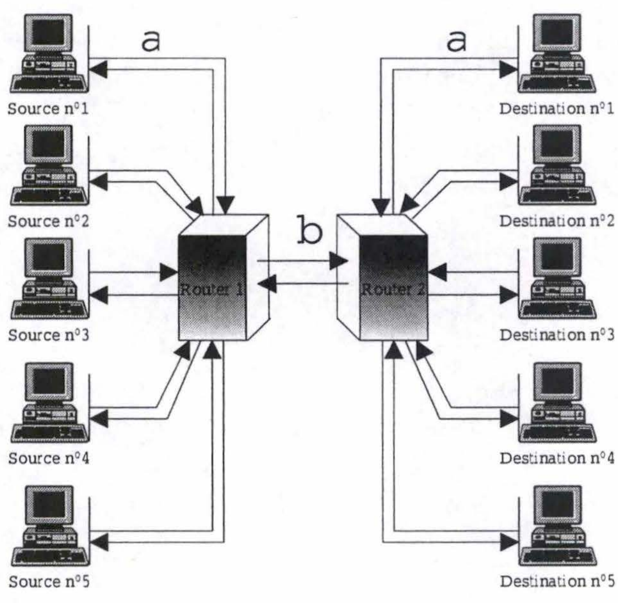


Figure 5.1: Single bottleneck topology

Object	Parameters	Values
Sources	*TCP* data packet size	1500 Bytes
	RAP data packet size	100 Bytes
	RAP ACK packet size	40 Bytes
	Sidelink delay (a)	2.5 msec
	Fine grain option	Active
Backbone		
	Queuing discipline	FIFO / RED (specified)
	Bottleneck link delay (b)	10 msec
Only *RAP*	Buffer size	30 Kbits
	Link bandwidth	80 Kbps
TCP and mix	Buffer size	500 Kbits
	Link bandwidth	1 200 Kbps
RED queue	Minimum threshold	30% of buffer size
	Maximum threshold	60% of buffer size
	Max. drop probability	10%
Simulation	Simulation length	105 sec
	Start-up phase	sequential starts

Table 5.1: Single bottleneck scenario parameters (SBN)

5.2 Simulations results

These simulations first illustrate the behaviour of *TCP* (base case for the rest of the simulations) with FIFO and RED queuing policy in the routers.

Afterwards, we will observe *RAP* confronted with itself, its intra-protocol fairness. The goal is to determine whether *RAP* is fair with itself or not. By fairness, we will observe the shared bandwidth along the *RAP* flows, the amount of transmitted packets and the influence of the different queuing disciplines.

We will then illustrate the RTT bias of *TCP* and the *RAP* behaviour confronted to it.

5.2.1 *TCP* base case simulations

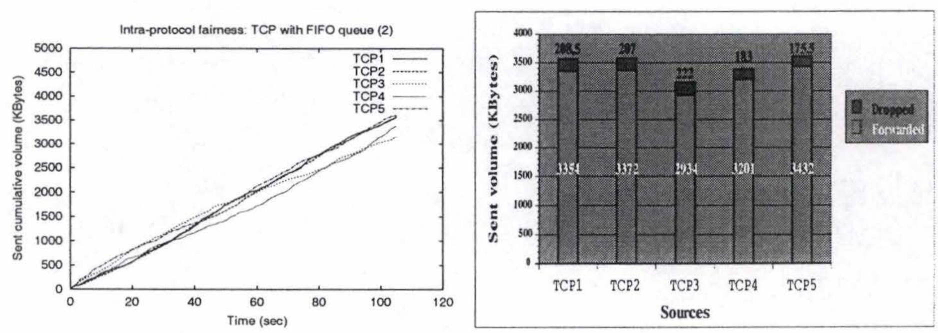
TCP with FIFO policy

The first simulation represents 5 co-existing *TCP* flows sharing the bottleneck bandwidth, FIFO as queuing discipline in the routers and no RTT modification.

We expect that *TCP* shares fairly the bandwidth between all the 5 flows (almost 5 confounded lines for graph (2)), sending the same volume of KBytes (3150 Kbytes are in average expected to be sent). The FIFO queuing discipline could interfere but with minor effect.

We can observe on Figure 5.2 that the 5 sources transmit almost the same amount of data at the same rate along the simulation (same slope for each curve). We can notice that *TCP* undergoes in average 5% of loss, quite a high loss but resulting of the small amount of sources. Small variations between the flows can be seen, the throughput oscillates a little bit but stays in average the same as show the small standard deviations really close to their average (12,14 KBytes/sec). We can say that the long-term fairness is good despite a short-term oscillation.

This could be explained by multiple causes: the bursty characteristic of the *TCP* transmission scheme combined with the FIFO policy, dropping in one time more packets of the same flow. It may also be due to a too short simulation length (not enough to converge), to some inner random parameters of the simulation or to some precision problems in computing values.



Flows	**Throughput** (KBytes/sec)	**St_dev** (KBytes/sec)
TCP1	30,97	12,26
TCP2	31,16	10,83
TCP3	26,53	12,02
TCP4	29,57	12,35
TCP5	32,03	13,21
Ideal	*30*	—

Figure 5.2: 5 *TCP* flows with FIFO queue: base case (FIFO)

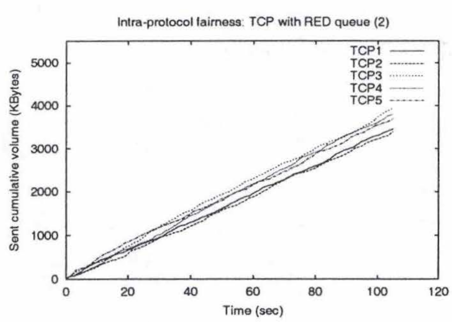
TCP with RED policy

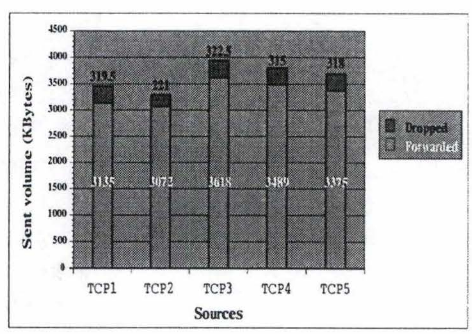
The second simulation is almost the same as the first one but this time with RED as the queuing discipline in the routers.

We expect that the network resources sharing will be almost perfect between all the 5 flows, due to the RED management (dropping randomly the same amount of packets of each flow in an homogeneous way).

We can observe on Graph 5.3 that the 5 sources do not transmit the same amount of packets for the simulation. The sending rate is almost equal for flows 3, 4 and 5 (slope of the curves are parallel) while a bit less for flows 1 and 2. Each flows undergoes the same number of drops. Small variations between the flows can be seen, the throughput oscillates a little bit but stays in average the same (st_dev close to its average of 11,67 KBytes/sec).

This could again be explained by the same causes as in the previous simulation: the bursty characteristic of the *TCP* transmission scheme. Even if the drops are more homogeneous, flows 1 and 2 suffer from it more than the other during the first 15 seconds, probably due to the start-up phase of *TCP*. Catching less bandwidth at the beginning, we may expect that the fairness will be reached at long-term even if the short-term is quite oscillating. It may also be due to some inner statistic bias.





Flows	Throughput (KBytes/sec)	St_dev (KBytes/sec)
TCP1	27,94	10,71
TCP2	27,20	11,43
TCP3	32,70	12,35
TCP4	31,50	11,01
TCP5	30,39	12,85
Ideal	*30*	—

Figure 5.3: 5 *TCP* flows with RED queue: base case (RED)

TCP-fifo Vs TCP-red

Comparing the TCP-fifo and the TCP-red simulation, using RED as queuing discipline generates more drops but it uniformly spreads them through the 5 flows along the simulation, ensuring drop fairness and thus smoothing the throughput fluctuation. The standard deviations, smaller in the second one, confirm it (flows reacting more slowly) even between the flows.

5.2.2 *RAP* simulations

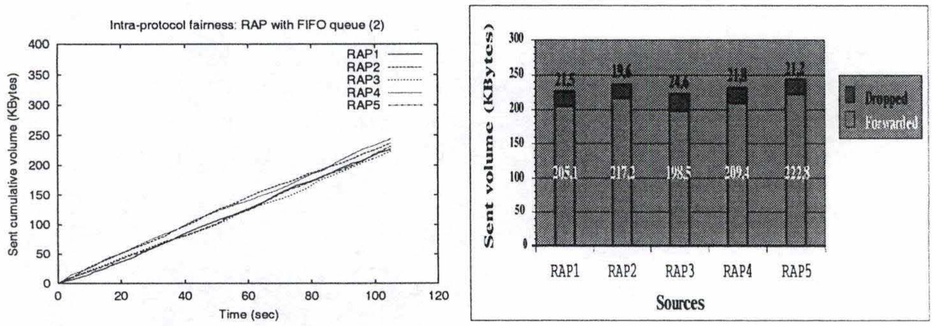
RAP with FIFO policy

This simulation represents 5 co-existing *RAP* flows sharing the bottleneck bandwidth, FIFO as queuing discipline in the routers and all sources have the same RTT.

Designed to adapt its sending rate smoothly, we expect that *RAP* will share fairly the bandwidth between all the 5 flows (almost 5 confounded lines for graph (2)), sending the same volume (210 KBytes are in average expected to be sent based on the router parameters). The FIFO queuing discipline should not interfere too much because of the smooth transmission scheme of *RAP*.

We can observe on Figure 5.4 that the 5 sources transmit quite the same amount of packets at the same rate along the simulation (surline slopes for each curve). The FIFO policy maintained this state, forwarding packets in a "blind" way. The sending rate is smooth with really light variations (low standard deviations and all close to the average of 0,56 KBytes/sec), the light variations coming from the FIFO policy. The curves are almost straight, indicating the quasi linearity of the transmission.

So, *RAP* flows adapt themselves to each other in a smooth way, without dominating flows, what could lead to flow starvation. We can say that the short-term and long-term fairness are good. The small variations are due to the FIFO policy, dropping consecutive packets of the same flow because of buffer overflow. But even with it, the rate still stays smooth.



Flows	Throughput (KBytes/sec)	St_dev (KBytes/sec)
RAP1	1,95	0,54
RAP2	2,07	0,60
RAP3	1,89	0,56
RAP4	1,99	0,50
RAP5	2,12	0,59
Ideal	*2*	—

Figure 5.4: 5 *RAP* flows with FIFO queue: base case (FIFO)

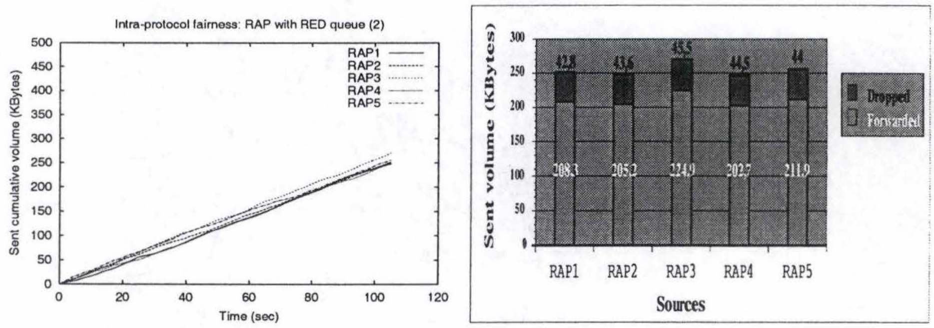
RAP with RED policy

This simulation is almost the same than the last one but this time with RED as queuing discipline in the routers.

We expect that the network resources sharing will be better than before between each the 5 flows, due to the RED management (randomly dropping the same amount of packets of each flow in an homogeneous way). This is supposed to represent the ideal scenario for *RAP*.

We can observe on Figure 5.5 that the 5 sources transmit in average the same amount of KBytes (210,6 KBytes) for the simulation. The sending rate is almost equal and quasi-linear (slope of the curves are parallel and almost straight). The loss ratios are the same (17.5 % in average).

This could again be explained by the smooth transmission scheme of *RAP* combined with the RED policy. The drops are more homogeneous; the flows do not suffer from consecutive losses and thus react with small variations, all together in a smooth way (low st_devs and close to the average of 0.49 KBytes/sec).



Flows	Throughput (KBytes/sec)	St_dev (KBytes/sec)
RAP1	1,98	0,53
RAP2	1,95	0,43
RAP3	2,14	0,51
RAP4	1,93	0,48
RAP5	2,02	0,49
Ideal	*2*	—

Figure 5.5: 5 *RAP* flows with RED queue: base case (RED)

Protocol	**FIFO st_dev** (Kbytes/sec)	**RED st_dev** (Kbytes/sec)	**Ratio** %
TCP	12,14	11,67	4
RAP	0,56	0,49	14

Table 5.2: Impact of RED on *TCP* and *RAP* flows

RAP-fifo Vs RAP-red

Comparing the RAP-fifo and the RAP-red simulation, it is obvious that using RED as queuing discipline generates more drops but it uniformly spreads them through the 5 flows along the simulation, ensuring drop fairness and thus smoothing the rate fluctuation. The standard deviations, smaller in the second one, confirm it (flows reacting more slowly). The amount of transmitted KBytes is fairer using RED; the slopes of the curves of the second simulation are almost straight and confounded, indicating that the sending rate adaptation is quasi-linear and the same for all the flows (not true for the first simulation).

FIFO queue Vs RED queue

Mixing the four first simulations is interesting to determine if applying the RED queue policy on *TCP* or *RAP* flows has a different impact. Based on the average standard deviation of the four simulation (cf. Table 5.2), RED seems to react better on *RAP* than on *TCP*. The ratio $\frac{AverageFIFOst\_dev}{AverageREDst\_dev}$ indicates an influence of 10% higher for *RAP* than for *TCP*. The reason comes from the transmission scheme of *RAP*, smoother than *TCP* (lower standard deviations). Indeed, RED used with *RAP*, try to homogeneously spread the losses of already homogeneously mixed flows. A contrary, *TCP* and its bursty characteristic does not help RED. So, the optimal working of *RAP* should be obtained with RED as queuing discipline.

5.2.3 Mixed flows simulations

FIFO policy

This simulation shows how *RAP* and *TCP* adapt themselves to each other, how they share the bandwidth, how they suffer from competition, from losses, ...

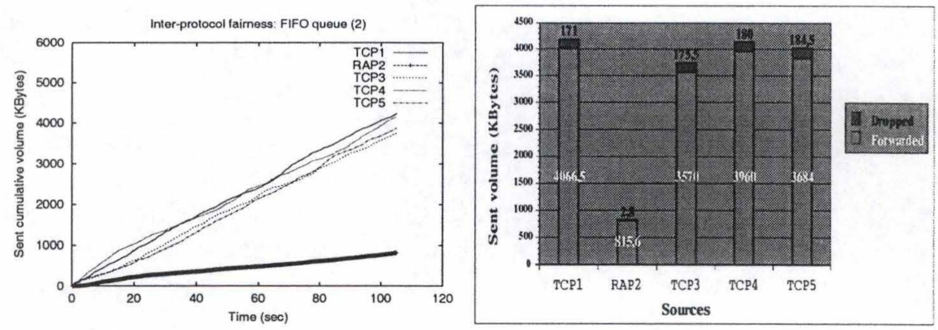
Designed to mimic *TCP*, *RAP* is supposed to adjust its sending rate to avoid any *TCP* starvation by using all network resources. Here the concept of fairness is an equilibrium between the number of transmitted packets and the obtained throughput. Combining the packet size of each protocol and the FIFO policy, the drop probability of *TCP* will be higher than the one for *RAP* (between 7 and 12 times bigger).

Examining the Figures in Figure 5.6 separately, we could conclude that *RAP* does not achieve its goal. First, it seems that *RAP* has far less throughput (4,5 times less) than *TCP*. Then if we compute the number of sent packets, it looks that *RAP* sends far more packets than *TCP* (3 times more).

In fact, this is exactly how *RAP* is supposed to react and it can be seen by examining its standard deviation: it is far lower than the one of *TCP*, indicating that it reacts in a smoother way (goal for multimedia). The major point is the *RAP* packets size (100 Bytes for *RAP* and 1500 Bytes for *TCP*, so 15 times longer). Due to the FIFO policy, small packets are more easily enqueued in the router's buffer than big ones (which are dropped) even if they are more numerous. That's why the average loss probability wont be correlated with a factor 15 but with a smaller one. *TCP* is thus undergoing more drops while *RAP* is able to sent more. If *RAP* does well mimic *TCP*, it should be checked by the *TCP* throughput equation (cf. [MSMO97]):

$$Throughput = \frac{PacketSize * C}{RTT * \sqrt{LossProb.}}$$

The average loss probability for *TCP* is 3,39 and 0.34 for *RAP*. The C constant is equal for the same simulation and the RTT (may be somewhat smaller for *RAP*) does not play a major role. If we introduce those values in the equation, *TCP* obtains indeed in average 4,5 times more throughput than *RAP*.



Flows	Throughput (KBytes/sec)	St_dev (KBytes/sec)
TCP1	38,17	13,10
RAP2	7,77	3,14
TCP3	33,21	13,23
TCP4	36,79	13,90
TCP5	34,30	12,15
Ideal	*30*	—

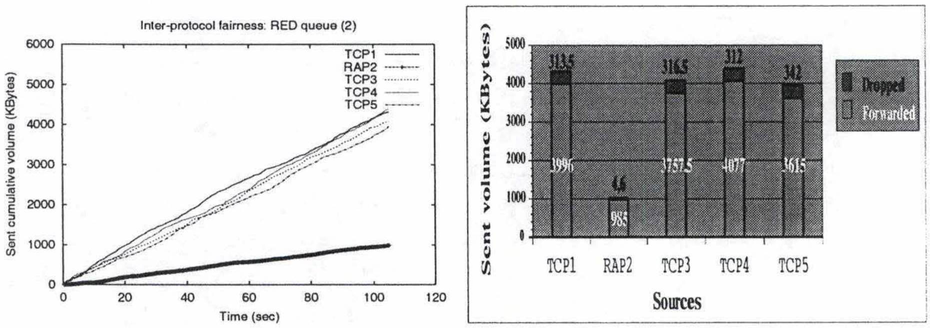
Figure 5.6: Inter-protocol fairness (FIFO queue)

RED policy

This simulation is almost the same than the last one but this time with RED as queuing discipline in the routers.

We expect that the network resources sharing will be better then before between all the 5 flows, due to the RED management (randomly dropping the same amount of packets of each flow in an homogeneous way) to reach the fairness.

We can observe on Figure 5.7 that the 5 sources transmit in average the same amount of KBytes for the simulation (with the same comments for *RAP* in the last simulation). The sending rate is quasi-linear (slope of the curves almost straight). We can thus deduct that at short-term or long-term, the fairness, based on the *TCP* throughput equation, is achieved.



Flows	Throughput (KBytes/sec)	St_dev (KBytes/sec)
TCP1	36,60	13,06
RAP2	9,38	3,78
TCP3	34,19	11,37
TCP4	37,21	12,77
TCP5	32,46	12,65
Ideal	*30*	—

Figure 5.7: Inter-protocol fairness (RED queue)

5.2.4 Mixed flows simulations with equal packets size

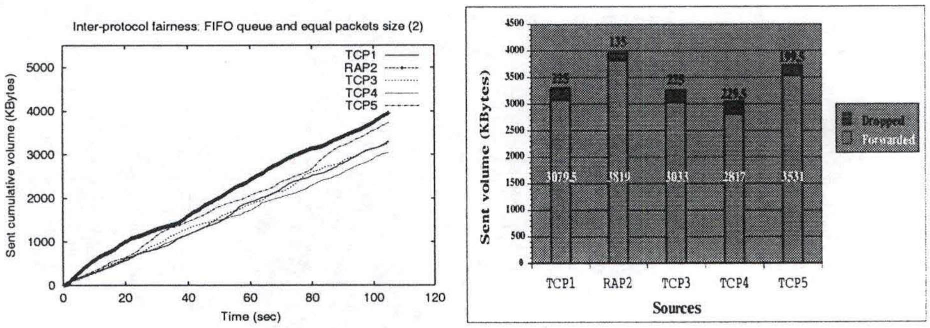
FIFO policy

This simulation shows how *RAP* and *TCP* adapt themselves to each other, how they share the bandwidth, how they suffer from competition, from losses but this time, the *RAP* packets size is equal to *TCP* packets size (1500 Bytes), which is not a too realistic packets size for multimedia applications (usually smaller to be able to minimize delays and jiters) but useful to give an overview of *RAP*'s behaviour without the packets size bias.

RAP is still supposed to smoother adjust its sending rate to avoid any *TCP* starvation by using the whole network resources but here the concept of fairness is not an equilibrium between the number of transmitted packets and the obtained throughput anymore. *RAP* should obtain the same network resources than *TCP*.

Examining the graphs in Figure 5.8, we can observe that *RAP* sent a bit more KBytes than *TCP* while undergoing less drops. *TCP* flows look quite oscillating. Each standard deviation is small and close to their average (12,99 KBytes/sec), indicating that the flows reacted in the same way. The slopes of the curves stay quite parallel (almost equal sending rate) what indicates that the differences between the flows just appear at the beginning of the simulation. With a longer simulation, we could confirm that the fairness will be achieved at long-term.

The fewer drops, the more sent data of *RAP* and the oscillating character of the *TCP* flows could be explained by the bursty transmission scheme of *TCP* compared to the smooth scheme of *RAP*, combined with the FIFO policy, dropping more often burst of *TCP* packets than isolated *RAP* ones.



Flows	**Throughput** (KBytes/sec)	**St_dev** (KBytes/sec)
TCP1	28,26	12,33
RAP2	36,37	13,53
TCP3	27,36	13,63
TCP4	25,53	11,86
TCP5	32,67	13,60
Ideal	*30*	——

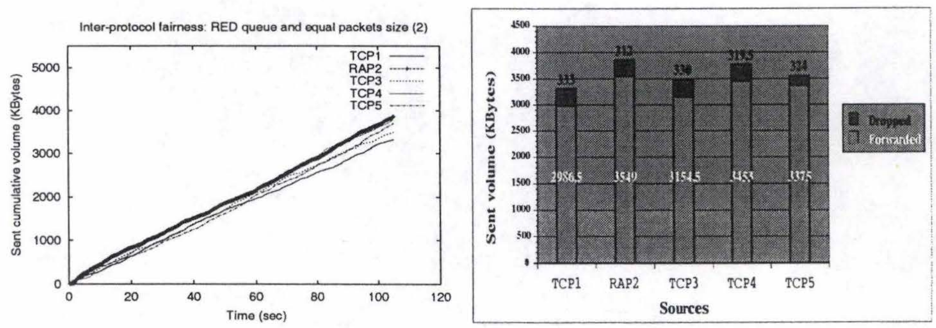
Figure 5.8: Inter-protocol fairness (FIFO queue and equal packets size)

RED policy

This simulation is the same than the last one but with RED as queuing discipline in the routers. *RAP* is still using long packets (1500 Bytes). Now that the packets size is the same, RED will not drop more likely *TCP*'s packets than *RAP*'s ones (cf. 5.1).

We expect the flows to adjust their rate in the same way, to undergo the same loss probability and to share the network resources almost perfectly.

We can observe on Figure 5.9 that the 5 sources transmit almost the same amount of data for the simulation. The sending rate is quasi-linear (slope of the curves almost straight) and each standard deviation is small and close to their average (11,22 KBytes/sec), indicating that the flows reacted in the same way. We can thus deduct that short-term and long-term fairness are achieved.



Flows	Throughput (KBytes/sec)	St_dev (KBytes/sec)
TCP1	26,40	10,20
RAP2	33,80	10,90
TCP3	28,19	11,88
TCP4	31,23	11,88
TCP5	30,27	11,26
Ideal	*30*	—

Figure 5.9: Inter-protocol fairness (RED queue and equal packets size)

Flows	Sidelink (a)	Bottleneck link (b)	Fixed RTT
1, 3, 5	2,5 msec	10 msec	30 msec
2	20 msec	10 msec	100 msec
4	60 msec	10 msec	260 msec

Table 5.3: RTT modification

5.2.5 Simulations with different RTT

From now on, we will observe the reactions (*biases*) of the different sources related with variations of their Round Trip Time (RTT). We will examine those biases still on the single bottleneck scenario depicted at the beginning of this chapter, with first only *TCP* sources (base case), then only *RAP* sources. We will finally observe the reaction of co-existing flows (one *RAP* and four *TCP*) in case of *TCP*'s RTT modifications.

To modify the RTT, we have significantly increased their sidelink access time (cf. "a" links on Figure 5.1) and to generate easily observable reactions, increases led to almost multiple by 3 and 9 the default fixed RTT, going from 30 msec to 100 msec and 260 msec (cf. Table 5.3).

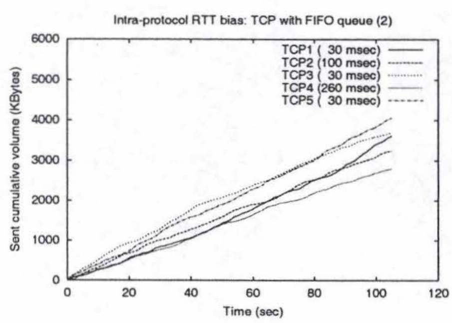
TCP simulation

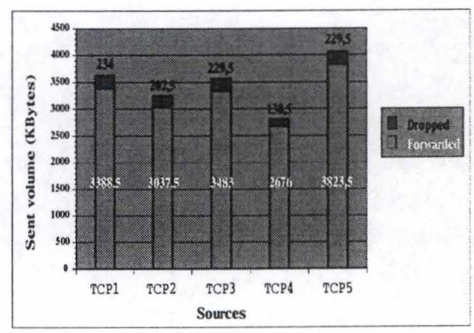
The first simulation represents 5 co-existing *TCP* flows sharing the bottleneck bandwidth with FIFO as queuing discipline in the routers and with the fixed RTT of flows 2 and 4 modified (100 msec for flow 2, 260 msec for flow 4).

We know that *TCP* suffers from RTT variation, adapting its sending rate slower then usual, obtaining thus less resources in congestion case. We expect that flows 2 and 4 will receive less bandwidth than the others in proportion to their increased RTT. The three remaining flows should fairly share the "available" bandwidth left by the two tested flows.

The observed behaviour on Figures 5.10 looks to what we expected. The standard deviations are smaller for flows 2 and 4 (average st_dev = 12,39 KBytes), corresponding to the higher RTT, indicating that *TCP* reacts more slowly due to modified RTT.

We can see that flow 1 exhibits a strange behaviour; it looks to suffer from the modification of flows 2 and 4 but at the end of the simulation, we can notice that it reached the two others unmodified flows. With a longer simulation, this should be converging to what we expect.





Flows	**Throughput** (KBytes/sec)	**St_dev** (KBytes/sec)
TCP1	30,93	14,01
TCP2	27,81	11,06
TCP3	31,66	13,22
TCP4	24,67	9,98
TCP5	35,13	13,68
Ideal	*30*	—

Figure 5.10: Intra-protocol RTT bias: *TCP* with FIFO queue

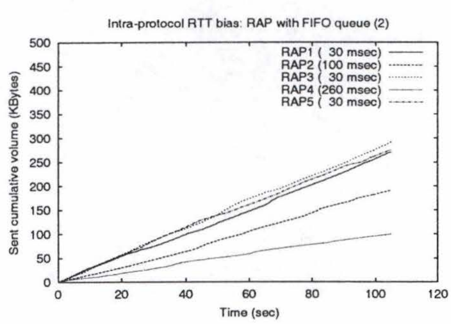
RAP simulation

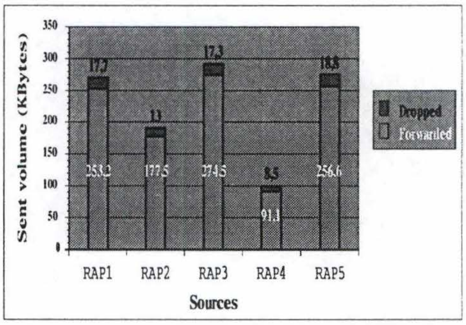
This simulation represents 5 co-existing *RAP* flows sharing the bottleneck bandwidth with FIFO as queuing discipline in the routers and with increased RTT for flows 2 and 4 (tripled for flow 2, times 9 for flow 4). Note that the *RAP* packets size is now again equal to 100 Bytes.

We would like to know, and expect that *RAP*, like *TCP*, suffers from RTT variation, adapting its sending rate slower then usual thus obtaining less resources in congestion case confronted with smaller RTT flows. We expect that flows 2 and 4 will receive less bandwidth than the others in proportion of their increased RTT. The three left flows should fairly share the "available" bandwidth not used anymore by the two tested flows.

The observed behaviour on Figures 5.11 looks to what we expected. *RAP* also suffers from the RTT modifications of flows 2 and 4. The proportion of lost throughput corresponds to the different RTT increases. The standard deviations are smaller for flows 2 and 4, corresponding to the higher RTT, indicating that *RAP* reacts more slowly due to bigger RTT (average st_dev = 0,56 KBytes).

We can still see small oscillations between flows with the same RTT (for the same reason as in 5.2.2).





Flows	**Throughput** (KBytes/sec)	**St_dev** (KBytes/sec)
RAP1	2,41	0,63
RAP2	1,69	0,48
RAP3	2,61	0,72
RAP4	0,87	0,34
RAP5	2,44	0,65
Ideal	*2*	—

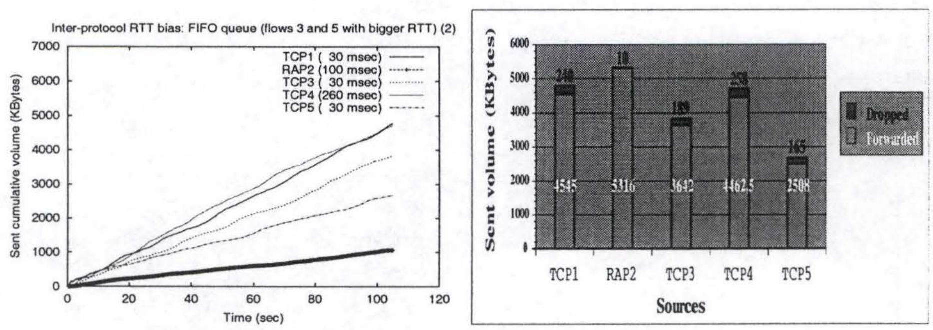
Figure 5.11: Intra-protocol RTT bias: *RAP* with FIFO queue

Mixed flows simulation

This simulation represents 4 *TCP* flows co-existing with 1 *RAP* flow, sharing the bottleneck bandwidth of the single bottleneck scenario with FIFO as queuing discipline in the routers and with increased fixed RTT for flows 3 and 5 (100 msec for flow 3, 260 msec for flow 5), two *TCP* flows (different from the Table 5.3).

Based on the former simulations, we expect that flows 3 and 5 lose some bandwidth, in the same way as in simulation 5.2.5 (proportionate to the RTT increases), bandwidth which should be fairly shared between the three remaining unchanged flows. Knowing that the *RAP* packets size is equal to 100 Bytes (15 times smaller than *TCP*), we will probably observe the same phenomena as in simulation 5.2.2

The observed behaviour on Figures 5.12 is what we expected. *TCP*, suffering from bigger RTT, lose some bandwidth for flow 3 and more for flow 5, adjusting their sending rate some what more slowly. The standard deviations for those two flows confirm it, smaller than the other.



Flows	**Throughput** (KBytes/sec)	**St_dev** (KBytes/sec)
TCP1	42,21	16,09
RAP2	10,13	3,32
TCP3	33,67	12,43
TCP4	41,07	17,77
TCP5	22,90	12,45
Ideal	*30*	—

Figure 5.12: Inter-protocol RTT bias: FIFO queue (flows 3 and 5 with bigger RTT)

TCP-fifo Forwarded vol. (KBytes)	TCP-fifo-rtt Forwarded vol. (KBytes)	TCP-fifo-rtt - TCP-fifo Difference (KBytes)	Variation %
3354	3388,5	34,5	+1,03 %
3372	3037,5	-334,5	-9,92 %
2934	3483	549	+11,37 %
3201	2676	-525	-16,40 %
3432	3823,5	391,5	+11,41 %

Table 5.4: Impact of RTT on *TCP* flows

5.3 Simulations comparisons

5.3.1 TCP-fifo Vs Mixed-fifo(1500)

Those two simulations show that, with equal packets size, *RAP* quite well mimics *TCP*. If we compare the four *TCP* common flows from the first and the second simulation, we can notice that both *TCP* sources transmit in average the same amount of packets (in fact a little bit less for the second). *RAP* benefits from the FIFO queuing discipline in the routers combined with the bursty characteristic of *TCP* to undergo less drops and catching a bit more of bandwidth.

5.3.2 TCP-fifo Vs TCP-fifo-rtt

Those two simulations indicate the bias of *TCP* confronted to longer RTT with FIFO queuing policy. The different behaviour are depicted in Table 5.4

The bandwidth reduction of flows 2 and 4 are correlated with a factor 2 (9% and 18%), representing the proportional increase of their respective RTT. One thing to mention, the small increase of throughput of flow 1 (1,3%) results from its strange behaviour during almost the whole simulation, reaching its expect place only at the end. A longer simulation would confirm the intuition of the end of the simulation.

5.3.3 TCP-red Vs Mixed-red(1500)

Those two simulations show that, with equal packets size, but this time with RED as queuing policy, *RAP* mimics *TCP* quite perfectly. If we compare the four *TCP* common flows from the first and the second simulation, we can notice that both *TCP* sources transmit in average the same amount of packets and almost the same as *RAP*. Equally spread drops between *TCP* and *RAP* based on the greediness of the sources, *RAP* behave quite like *TCP*

RAP-fifo Forwarded vol. (KBytes)	RAP-fifo-rtt Forwarded vol. (KBytes)	RAP-fifo-rtt - RAP-fifo Difference (KBytes)	Variation %
205,1	253,2	48,1	+23,45 %
217,2	177,5	-39,7	-18,28 %
198,5	274,5	76	+38,29 %
209,4	91,1	-118,3	-56,49 %
222,8	256,6	33,8	+14,77 %

Table 5.5: Impact of RTT on *RAP* flows

5.3.4 RAP-fifo Vs RAP-fifo-rtt

Those two simulations indicate the bias of *RAP* confronted to longer RTT with FIFO queuing policy. The different behaviour are depicted in Table 5.5

The bandwidth decrease of flows 2 and 4 are correlated with a factor bigger than 2 (20% and 57%), representing more than the proportional increase of their respective RTT. The repartition of the *available* bandwidth is well shared between the three remaining flows. We observe that *RAP* suffer more of the RTT variation than *TCP*. This is due to the way *RAP* uses RTT to both determine the duration step for constant bit sending rate and the sending rate itself. With longer RTT, *RAP* reacts twice, adapting its sending rate less often and less rapidly.

5.3.5 Mixed-fifo Vs Mixed-red

Comparing the Mixed-fifo and the Mixed-red simulation, it is obvious that using RED as queuing discipline generates more drops but it uniformly spreads them through the 5 flows along the simulation, ensuring drop fairness and thus smoothing the rate fluctuation but mainly for *TCP* because here, *RAP* benefits from its small packet size, undergoing far less drops and thus having a high sending rate.

5.4 Conclusions

These simulations tried to evaluate the TCP-Friendly behaviour of the *RAP* protocol according to some main parameters (different round trip times, the fine grain option, ...).

We first evaluated *TCP* with FIFO and RED queuing discipline to obtain the base cases for further comparisons. We then tested the *RAP* protocol confronted with itself, to see in which way it reacts to different queuing disciplines and packets size while being compared with *TCP*. Afterwards we confronted *TCP* and *RAP* again with the FIFO and the RED policies. We finally confronted both protocols independently and then mixed with variations of some RTT flows. Knowing that *TCP* suffers from that, it was interesting to know if *RAP*, designed to mimic *TCP*, also suffered from RTT variations.

We observed that *RAP* adapts its sending rate really smoothly (usual low standard deviation, less than *TCP*'s ones), which is the main goal we are working towards for multimedia applications.

It also appears that *RAP* behaves pretty well in competition with *TCP* flows, adapting its transmission rate based on the network charge without generating any collapse on the network. It avoids *TCP* starvation while offering a "quite fair" bandwidth sharing.

A general observation indicates that using RED as queuing discipline in the routers uniformly spreads the drops through all the flows along the simulation, smoothing the adaptive character of the flows.

We can then notice that *RAP*, like *TCP*, undergoes the effects of bigger RTT but far more than *TCP*. So, *RAP* seems to be more sensitive to RTT variations.

Thus, in a general way and based on those simulations, we can conclude that *RAP* has the researched transmission scheme (smooth) for multimedia applications, that *RAP* is TCP-Friendly, sharing the network resources based on its load and that *RAP*, like *TCP*, also suffers, in a worse way, from the RTT bias.

Chapter 6

Conclusions

In the *first chapter*, we introduced the today situation of the Internet (wide heterogeneous best effort network using mainly FIFO as queuing policy). We then introduced the problem of the emerging multimedia applications based on *UDP*, generating large amount of non-responsive traffic . Suffering from a lack of congestion control mechanism, those applications required the addition over *UDP* of mechanisms to avoid any collapse or *TCP* starvation and to ensure a fair sharing of the network resources.

In *chapter 2*, we described the two main transport protocols used nowadays: *TCP* and *UDP*. Based on the requirements of multimedia applications and the transmission scheme of *TCP* and *UDP*, *UDP* has been chosen and enhanced by *RTP*, in a first step, to provide flow control and some more services. And now, in a second step, a congestion control mechanism is envisaged.

In *chapter 3*, we fully described the Rate Adaptive Protocol (*RAP*) as a congestion control mechanism. Designed to mimic *TCP*, it performs a compatible transmission scheme with *TCP*, ensuring TCP-Friendliness and fair sharing of the network resources with responsive flows.

In *chapter 4*, we introduced 3 other congestion control mechanism, working in different ways than *RAP* to show that multiple solutions can be followed. Depending on the users requirements, they have specific features characterizing their utilisation choice.

In *chapter 5*, we confirmed the TCP-Friendliness and fair sharing of *RAP*, when competing with *TCP* flows. We pointed out some specific behaviours of *RAP* encountering typical problems (RTT variations, different sources, different packets sizes).

6.1 Evaluation

The main goal designing *RAP* was to avoid new congestion collapse of the Internet due to enormous uncontrolled traffics. Being too aggressive, *UDP*, even if less used on the net than *TCP*, may lead to extreme unfairness related with controlled traffics, monopolizing the resources.

Applications that can sustain a certain amount of loss may find *RAP* interesting for its ability to adapt to the network load while, at the same time, acting in a TCP-Friendly way. Nevertheless, *RAP* should not be used in the context of applications requiring no loss or multicast communication.

For what kind of applications is *RAP* useful. Only applications able to adapt their throughput and having to do it.

The scenarios of the simulations were supposed to be the best ones to observe differences between FIFO and RED policy (because undergoing high losses cf. [CJOS01]). In fact, the RED effect was almost insignificant. Furthermore, those differences only appear above 90% of load, what is almost never reach in reality.

We observed that the throughput of *RAP* behaves in same way as *TCP* but more smoothly, reaching a "fair" state in the sharing of the network resources.

6.2 Further work

One main characteristic pointed out through the simulations is the high sensitivity of *RAP* when confronted with RTT variation. It should be of high interest to modify the way the RTT comes into play in the transmission rate adaptation scheme. Another non-trivial challenge would be to modify *RAP* and its implementation to support multicast communications.

It would also be interesting to confront *TCP*, *RAP* and *UDP* flows in one simulation to observe the competition and estimate the resistance of *TCP* and *RAP* faces the aggressiveness of *UDP*, (also adding the introduced congestion control mechanisms of Chapter 4).

The simulations done are a bit too theorical. To estimate the *RAP* behaviour related with real conditions, more realistic simulations would be of greater interest. For example, simulations with longer duration time to observe the long-term results (avoiding start-up effects and strange behaviour due to not well appropriated initialisation of variable or too short convergence time). Adding more sources (especially *TCP* sources) would trend to represent daily configuration of the Internet (and minimizing the oscillating behaviour observe for *TCP*).

Finally, an good improvement would be to allow to parameter the adaptivity scheme of *RAP* based on the applications output rate. This could lead to a mechanism able to adapt itself to every kind of streaming flows, avoiding the drawbacks of both *UDP* and *TCP*: less aggressive than *UDP* and reacting smoother than *TCP*.

Bibliography

[Bol92] J.-C. Bolot. End-to-end packet delay and loss behaviour in the internet. *Computer Communication Review 23*, n^o4, oct 1992.

[Bra89] R.T. Braden. Requirements for internet hosts-communication layers. Internet Engineering Task Force, RFC 1122, oct 1989.

[CC96] R. L. Carter and M. E. Crovella. Measuring bottlenck link speed in packet-swichted networks. Technical Report BUCS-96006, Computer Science Departement, Boston University, mar 1996.

[CE99] S. De Cnodder and O. Elloumi. RED behaviour in the presence of different packet sizes. Technical Report TTD-816, Alcatel Bell, Network Architecture, Traffic Technologies, 15 September 1999.

[CJOS01] M. Christiansen, K. Jeffay, D. Ott, and F. Smith. Tuning red for web traffic. *Transaction on Networking*, 9(3), jun 2001.

[Cno99] S. De Cnodder. TTD_*813:* OPNET TCP *Simulator*, November 1999.

[CP00] S. De Cnodder and K. Pauwels. Rate based n-RED: The final frontier? Technical Report TTD-819, Alcatel Bell, Network Architecture, QoS, Traffic and Routing Technologies, 9 February 2000.

[ea96] H. Schulzrinne et al. Rtp: A transport protocol for real-time applications. Internet Engineering Task Force, RFC 1889, January 1996.

[FJ93] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. In *IEEE/ACM Transactions on networking*, aug 1993.

[IRY00] V. Ozdemir I. Rhee and Y. Yi. TEAR: TCP emulation at receiver - flow control for multimedia streaming. Technical report, Dept. of Computer Science, New-York University, Apr 2000.

[Jac88] V. Jacobson. Congestion avoidance and control. *Computer Communication Review*, n^o18(4):314–329, Aug 1988.

[Jac90] V. Jacobson. Modified TCP congestion avoidance algorithm. end2end-interest mailing list, 30 April 1990.

[Jac00] V. Jacobson. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. Internet Engineering Task Force, RFC 2001, January 2000.

[JPK98] D. Towsley J. Padye, V. Fioriu and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *ACM SIGCOMM'98*, oct 1998.

[MF97] J. Mahadavi and S. Floyd. TCP-friendly unicast rate-based flow control. end2end-interest mailing list, jan 1997.

[MIL] Inc MIL 3. OPNET *Modeler*. Release 7.0.B, Online documentation.

[Mog93] J.C. Mogul. IP network performance. In D.C. Lynch and M.T. Rose, editors, *Internet System Handbook*, pages 575–675. Addison-Wesley, Reading, Mass, 1993.

[MSMO97] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. Computer Communications Review, 27(3), jul 1997.

[Nag84] J. Nagle. Congestion control in IP/TCP internetworks. Internet Engineering Task Force, RFC 896, 6 January 1984.

[Pos80] J.B. Postel. User datagram protocol. Internet Engineering Task Force, RFC 768, nov 1980.

[Pos81a] J.B. Postel. ICMP: Internet control message protocol. Internet Engineering Task Force, RFC 792, sep 1981.

[Pos81b] J.B. Postel. Transmission control protocol. Internet Engineering Task Force, RFC 793, sep 1981.

[RF99] K. Ramakrishnan and S. Floyd. ECN: Explicit congestion notification. Internet Engineering Task Force, RFC 2481, jan 1999.

[SFP00] M. Handley S. Floyd and J. Padhye. TCP friendly rate control (tfrc): Protocol specification. Internet Engineering Task Force, INTERNET-DRAFT, November 2000.

[SFW00] J. Padhye S. Floyd, M. Handley and J. Winder. Equation-based congestion control for unicast applications: the extended version. Technical report, International Computer Science Institut, mar 2000.

[SS99] D. Sisalem and H. Schulzrinne. The loss-delay adjustment algorithm: A TCP-friendly adaptation scheme, 1999.

[Ste94] W.R. Stevens. TCP/IP *illustré: Les Protocoles*, volume n^o1. Addison-Wesley Publishing company, Inc., 1994.

[SW00a] D. Sisalem and A. Wolisz. Mdla: A TCP-friendly congestion control framework for heterogeneous multicast environments. 8th Int'l Wksp. QoS, June 2000.

[SW00b] D. Sisalem and A. Wolisz. TCP-friendly adaptation: A comparison and measurement study. Int'l Wksp. Network and OS Support for Digital Audio and Video, June 2000.

Appendix A

Simulation tool

In this chapter, we introduce the simulation tool we have enhanced to evaluate the congestion control mechanism (*RAP*) and also modules implemented to simulate this mechanism.

For the simulation tool, a brief introduction will be given (overview of how it works and what is possible). Based on this, we will see the way *RAP* has been implemented in the main layers. For the simulation topology, please refer to Section 5.

A.1 OPNET introduction: way of working

OPNET Modeler 7.0 b ([MIL]) is a vast software package with an extensive set of features designed to support general network modelling and to provide specific support for particular types of network simulation projects.

OPNET is a graphical tool for developing networks with different topologies and based on the C programming language. Its module architecture makes it really simple to use, new mechanisms can be "easily" implemented and added, tested and their results analysed.

To be able to simulate *TCP* traffic with every available upgrades (TCP-Tahoe, Reno, Sack, Fack, ...) or stuff like ECN, the module STCP has been added. For more information about how STCP and OPNET are linked to each other, we refer the reader to [Cno99].

A.1.1 Some keywords:

- **Object orientation**: systems specified in OPNET consist of objects, each with configurable set of attributes. For example, the PDF editor let you create, edit, and view Probability Density Functions (PDFs). PDFs can be used to control certain events, such as the frequency of packet generation in a source module, called *ideal generators* (i.e. for *UDP* transport protocol). Objects belong to *classes* which provide them with their characteristics in terms of behaviour and capability. Definition of new classes are supported in order to address as wide a scope of systems as possible. Classes can also be derived from other classes, or *specialized* in order to provide more specific support for particular applications.

- **Specialized in communication networks and information systems**: OPNET provides many constructs relating to communications and information processing, providing high leverage for modelling of networks and distributed systems.

- **Graphical specification**: wherever possible, models are entered via graphical editors. These editors provide an intuitive mapping from the modelled system to the OPNET model specification.

- **Flexibility to develop detailed custom models**: OPNET provides a flexible, high-level programming language with extensive support for communications and distributed systems. This environment allows realistic modelling of all communications protocols, algorithms, and transmission technologies.

- **Automatic generation of simulations**: model specifications are automatically compiled into executable, efficient, discrete-event simulations implemented in the C programming language. Advanced simulation construction and configuration techniques minimize compilation requirements.

- **Application-specific statistics**: OPNET provides numerous built-in performance statistics that can be automatically collected during simulations. In addition, modellers can augment this set with new application-specific statistics that are computed by user-defined processes.

- **Integrated post-simulation analysis tools**: performance evaluation, and trade-off analysis require large volumes of simulation results to be interpreted. OPNET includes a sophisticated tool for graphical presentation and processing of simulation output.

- **Interactive analysis**: all OPNET simulations automatically incorporate support for analysis via a sophisticated interactive debugger.

- **Animation**: simulation runs can be configured to automatically generate animations of the modelled system at various levels of detail and can include animation of statistics as they change over time. Extensive support for developing customized animations is also provided.

- **Application Program Interface (API)**: as an alternative to graphical specification, OPNET models and data files may be specified via a programmatic interface. This is useful for automatic generation of models or to allow OPNET to be tightly integrated with other tools.

A.1.2 Graphical editors of OPNET: the layers sub-division.

OPNET supports model specification with a number of tools or editors that capture the characteristics of a modelled system's behaviour. Because it is based on a suite of editors that address different aspects of a model, OPNET is able to offer specific capabilities to address the diverse issues encountered in networks and distributed systems.

The model-specification editors are organized in an essentially hierarchical fashion. Model specifications performed in the Project Editor rely on elements specified in the Node Editor; in turn, when working in the Node Editor, the developer makes use of models defined in the Process Editor. The remaining editors are used to define various data models, typically tables of values, packet formats, that will be later referenced by process - or node - level models.

Organization:

- **The project editor**: it is the main area to create a network model using standard objects from the library. There you can collect statistics about the network, run the simulations and view the results. You also may access to sub-layer constructors to create specific objects you need for your experimentations, object like packet format, specific links,... (Cf. Figure A.1)

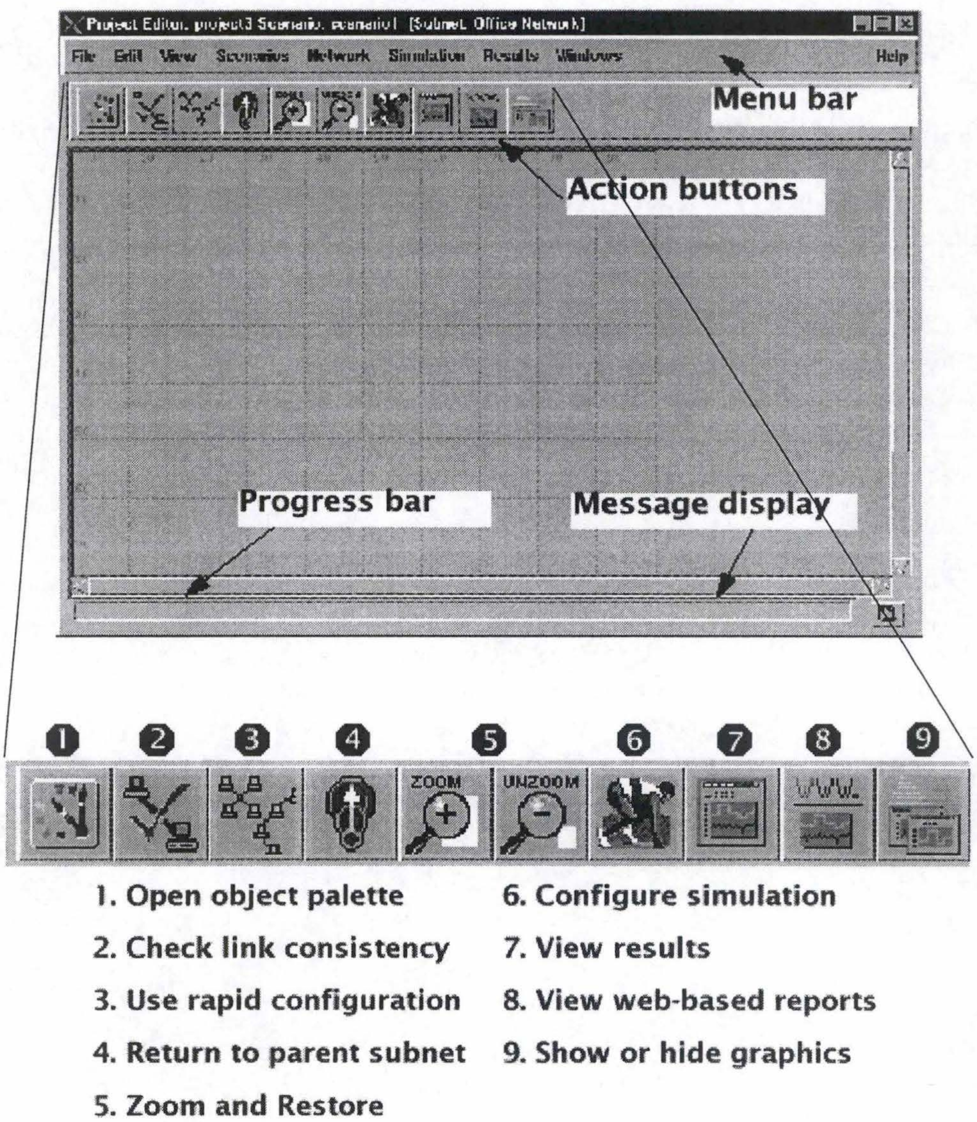


Figure A.1: Project editor window

- **The Network layer**: a network model defines the overall scope of a system to be simulated. It is a high-level description of the objects contained in the system. The network model specifies the objects in the system, as well as their physical locations, interconnections and configurations. The size and scope of the networks modelled can range from simple to complex. A network model may contain a single node, a single sub-network, or many interconnected nodes and sub-networks, since the structure and complexity of a network model typically follows those of the system to be modelled. Every network object (except links) has an underlying node model, which specifies the internal flow of information in the object. Node models are made up of one or more modules connected via packet streams or statistic wires. Node modules in turn contain process models, which are represented by state transition diagram.

- **The node editor**: develops node models. It is used to define the behaviour of each network object. Their main components are modules, packet streams and statistic wires (cf. Figure A.2). The internal functionalities of the modules (process models) will be explained in the next point.

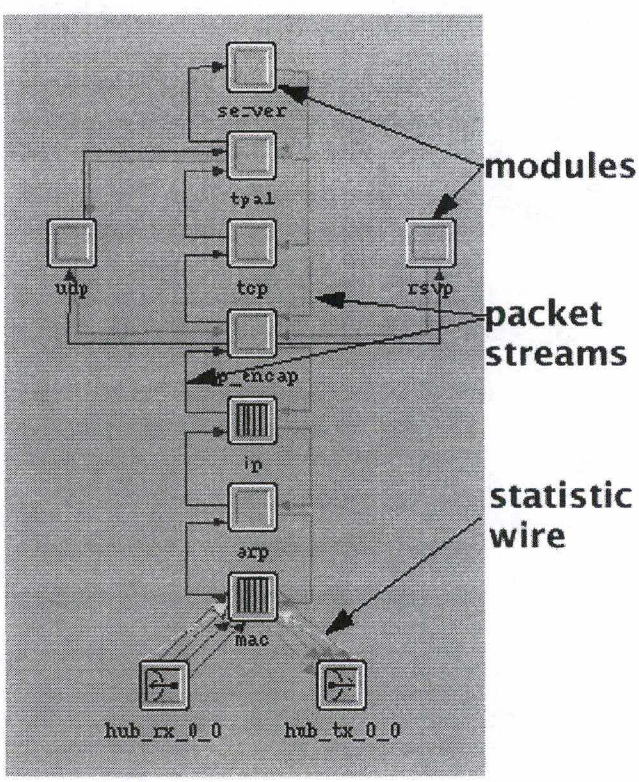


Figure A.2: Node editor window

- **The process model**: it is used to control the underlying functionalities of the node model created in the node editor. Finite state machines, composed of states and transition, represent process models. Every actions performed in a state are defined in C or C++ language. C language has been used here (cf. Figure A.3)

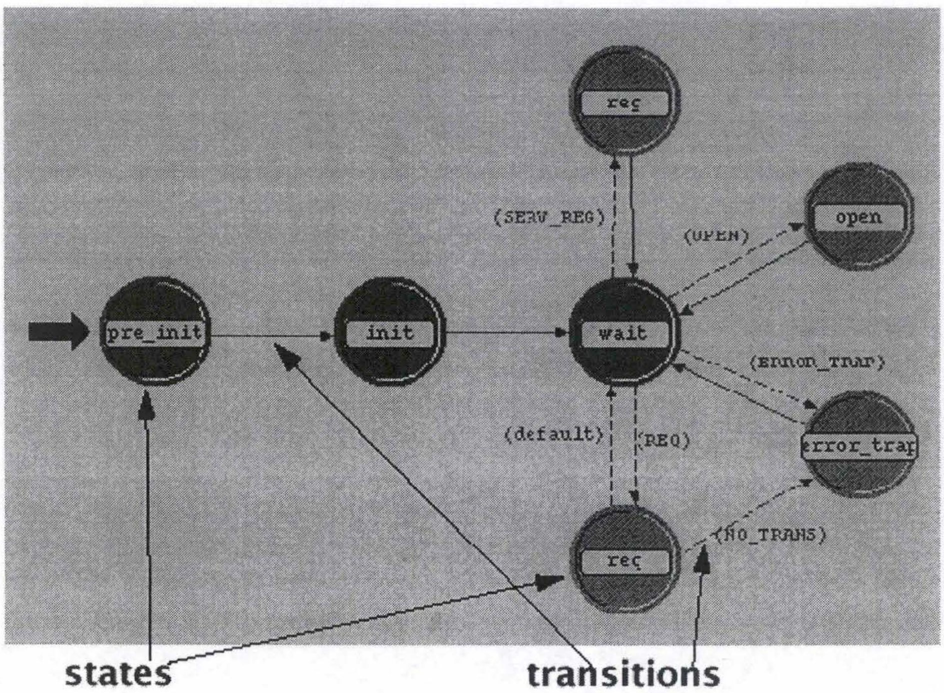


Figure A.3: Process model window

Some explanations:

- Each state contains an *enter executive* and an *exit executive*, executed when a process enters and leaves a state, written in C or C++ language.
- As you can see, there are dark (red in OPNET) and light (green in OPNET) states. The dark ones are called *unforced states*, this means that after executing the enter executive, the process model blocks and returns control to the simulation kernel. The next time the process model is invoked; execution begins again from the state in which it was blocked. In the *forced states*, on the other hand (light ones), the process model does not stop after the enter executive but carries on straight to the exit executive and follows the transition to the next state.
- As you can also see, they are two kind of transition represented by doted and solid lines. The solid lines are *unconditional transition*: this means that after having executed the exit executive, the process model directly proceeds to the next state. The doted lines are *conditional transition*. This condition is defined in a macro and explains at which condition after executing the exit executive the process model is allowed to carry on to the next state, otherwise the process model is stopped there and looks for a "true" condition. If every condition are negative and no unconditional transition carries out of this state, the simulation stops.
- This simulation tool also offers the possibility to create, edit, and view link models with specific parameters. You are also able to develop packet formats models. Packet formats dictate the structure and order of information stored in a packet and used during the simulations.

Appendix B

Implemented modules

In this section, we will describe the source, the destination and the router module based on the different layers explained in the last section. The project editor combined with the network layer are mainly used to describe the general configuration from the physical point of vue with high-level objects (here the single bottleneck topology) and are depicted in Chapter 3.

The node editor will show the collaboration and interactions between all the components of every object of the above layers.

Finally the process model layer, which stands at a law-level, contains and puts in play the implementation code. (For the complete code, please refer to the appendix)

B.1 Network layer

Here is described the general network topology (single bottleneck) at the higher level. If we look at Figure B.1, we can see 5 sources (A), 2 router (B) and 5 destinations (C). The sources can be assimilated to ISP.

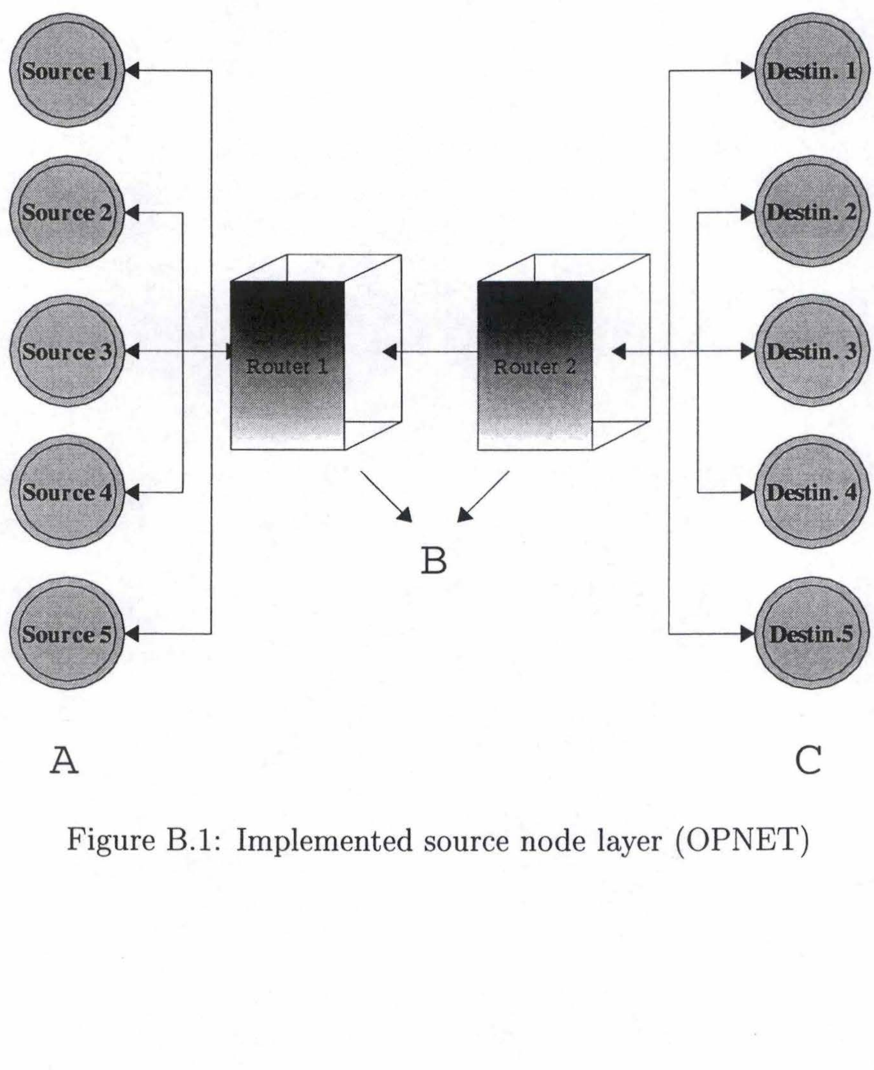


Figure B.1: Implemented source node layer (OPNET)

B.2 Node layer

B.2.1 Sources

Figure B.2 depicts the node layer structure of one source object (A) and the way of working for this source. Some simplifications have been brought to make the graph easier to read.

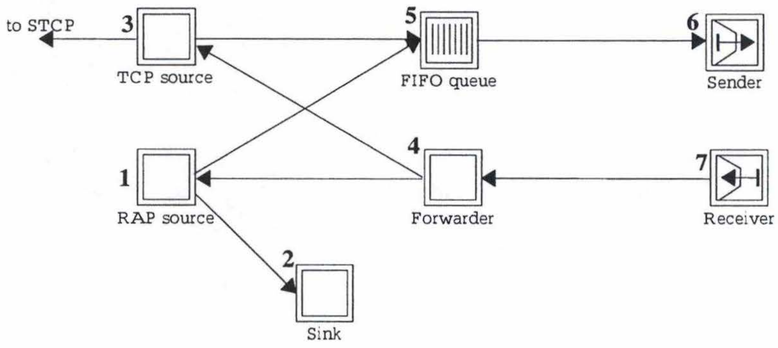


Figure B.2: Implemented source node layer (OPNET)

Explanation:

Node 1 : **The *RAP* source**: contains the packets generator controlled by the *RAP* congestion control mechanism (main point with the process model depicted above).

Node 2 : **The sink**: just used to drop ACK after use.

Node 3 : **The forwarder**: dispatches the received ACK (*TCP* and *RAP*) to the correct source.

Node 4 : **The *TCP* source**: classical object in OPNET, it is the connection between the STCP module and the *TCP* node in this module.

Node 5 : **The FIFOqueue**: simulate the co-existence between *TCP* and *RAP* flows on the same host. It also simulates the access link time.

Node 6 : **The sender**: is the start point of a connection between two modules (here between the source and a router).

Node 7 : **The receiver**: is the end point of a connection between two modules (here between a router and the source).

B.2.2 Destinations

Figure B.3 depicts the node layer structure of one destination object (C) and the way of working for this destination. Some simplifications have been brought to make the graph easier to read. In fact, it is exactly the same than at the source with the same goals.

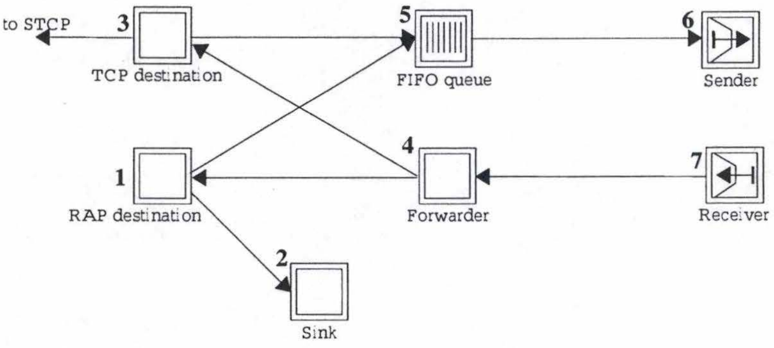


Figure B.3: Implemented source node layer (OPNET)

Explanation:

Node 1 : **The *RAP* destination**: contains the ACK generator controlled by the *RAP* congestion control mechanism (main point with the process model depicted above).

Node 2 : **The sink**: just used to drop the packets after use.

Node 3 : **The forwarder**: dispatches the received packets (*TCP* and *RAP*) to the correct destination.

Node 4 : **The *TCP* destination**: classical object in OPNET, it is the connection between the STCP module and the *TCP* node in this module.

Node 5 : **The FIFOqueue**: simulate the co-existence between *TCP* and *RAP* flows on the same host. It also simulates the access link time.

Node 6 : **The sender**: is the start point of a connection between two modules (here between the source and a router).

Node 7 : **The receiver**: is the end point of a connection between two modules (here between a router and the source).

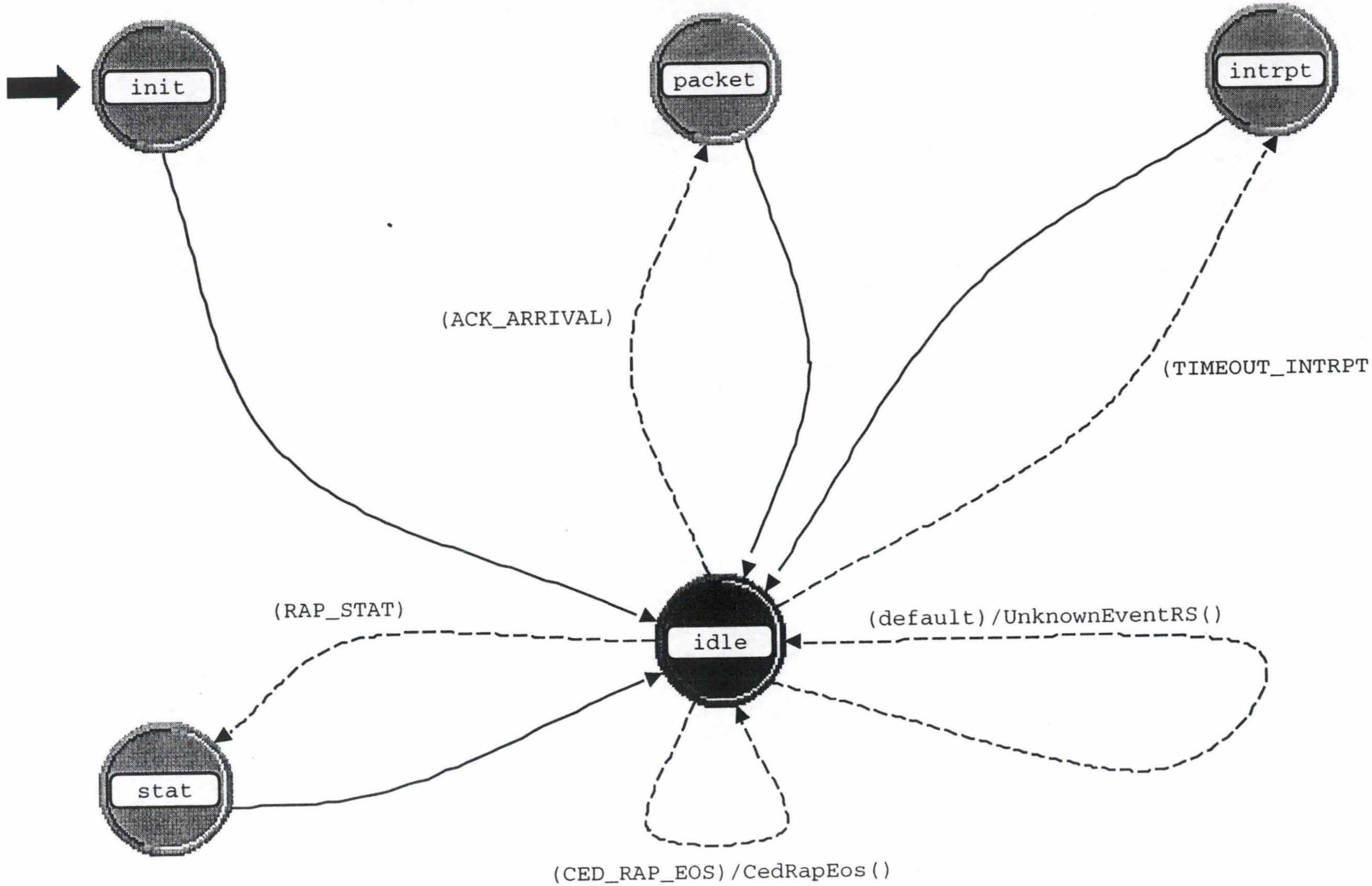
B.3 Process layer

The next page show the process model implemented to simulate *RAP*, describing the *RAP* source node, the destination one and the router implementation. In those states and through those transitions is represented the behaviour of each side of the studied congestion control mechanism and the router policy. Here will stand the functions explained in Chapter 3.

B.3.1 Sources

As we can see, only the "Idle" state is unforced i.e. the process stops after the execution of the enter section. Next pages give the complete code of *RAP* in OPNET running behind the 5 different processes.

Process Model: CED_RAP



```
/* Inter Packet Gap = (ipg*srtt) / (ipg+srtt) */
double  \ipg;

/* Smooth Round Trip Time = srtt + delta*diff */
/* Used in rttTimeout                          */
double  \srtt;

/* = 0 if fine grain not used (default) */
int \finegrain;

/* Used to compute the waitperiode to reschedule ipgtimer */
/* Used in ipgTimeout                                     */
double  \frtt;

/* Used to compute the waitperiode to reschedule the ipgtimer */
/* Used in ipgTimeout                                         */
double  \xrtt;

/* ATTR (0.5): To increase ipg so to decrease rate */
double  \beta;

/* ATTR (0.9): Weight of the samplertt to compute frtt variable */
double  \kfrtt;

/* ATTR (0.01): Weight of the samplertt to compute xrtt variable */
double  \kxrtt;

/* For UpdateTimeValues, first initialisation of some variables */
/* TRUE == 1 init                                               */
/* FALSE == 0                                                   */
int \initial;

/* Variable used to check if losses occur based on timer fire */
/* Used in TimerLostPacket                                    */
double  \timeout;

/* ATTR (1.2): Used to compute timeout variable : mu*srtt + phi*variance */
double  \mu;

/* ATTR (4.0): Used to compute timeout variable : mu*srtt + phi*variance */
double  \phi;

/* Used to compute timeout variable : mu*srtt + phi*variance */
double  \variance;

/* ATTR (0.5): Used to compute variance and srtt variables */
/* variance = variance + delta*(diff - variance)           */
/* srtt = srtt + delta*srtt                                */
double  \delta;
```

```
Liste*  \list;

int \seqnum;

Objid   \my_self;

/* name of the sending source */
char    \nameSRC[30];

int \totpack;

int \totack;

Evhandle    \event;

FILE*   \statfile;

char    \nameISP[30];

double  \starttime;

FILE*   \graph;

int \udpsize;

FILE*   \statfile2;

int \pacwhilesrtt;

int \lossoccur;
```

```
#include<stdio.h>

#define ACK_ARRIVAL     ( op_intrpt_type() == OPC_INTRPT_STRM )
#define CED_RAP_EOS     ( op_intrpt_type() == OPC_INTRPT_ENDSIM )
#define TIMEOUT_INTRPT ((op_intrpt_type() == OPC_INTRPT_SELF) && (op_intrpt_code() != 3))

#define TRUE 1
#define FALSE 0


#define RAP_STAT        ((op_intrpt_type() == OPC_INTRPT_SELF) && (op_intrpt_code() == 3))

// Structure of an element (TransHistoryEntry) of the records table
typedef struct listelement
    {
        int seqno;
        int state;
        double departureTime;
        struct listelement * next;
    }TransHistoryEntry;

// Structure of the pointer (Liste) to the records table
typedef struct
    {
        TransHistoryEntry * first;
        TransHistoryEntry * last;
        int size;
    }Liste;
```

```
void UnknownEventRS(void)
{
    int stat;

    printf("UnknownEvent in %s\t%s\n", nameISP, nameSRC);

    op_ima_obj_attr_get(my_self, "stat_file", &stat);
    if(stat)
    {
        fprintf(graph,"££       END_OF_SIMULATION UE        ££\n");
        if(fclose(graph) != 0) {printf("Graph file not well closed!!!\n");}

        fprintf(statfile,"\n£££££££££££££££££££££££££££££\n");
        fprintf(statfile,"££     END_OF_SIMULATION UE   ££\n");
        fprintf(statfile,"£££££££££££££££££££££££££££££££\n\n");
        if(fclose(statfile) != 0) {printf("Stat file not well closed!!!\n");}

    }
    op_sim_end("END OF SIM : "," UNKNOWNEVENT IN ","RAP SRC","");
}

void CedRapEos(void)
{
    int stat;

    if(totpack != 0)printf("In %s, %s sent %d packets and received %d ACKs\n", nameISP, nameSRC, totpack, totack);

    KillList();

    op_ima_obj_attr_get(my_self, "stat_file", &stat);
    if(stat)
    {
        fprintf(statfile,"END_OF_SIMULATION\n");
        if(fclose(statfile) != 0) {printf("Stat file not well closed!!!\n");}

        fprintf(graph, "%d\n", totpack);
        fprintf(graph,"END_OF_SIMULATION\n");
        if(fclose(graph) != 0) {printf("Graph file not well closed!!!\n");}
    }
}

int DecreaseIpg(void)   //void
{
//1 printf("\t%f\t->\t", ipg);
    fprintf(statfile2,"%f;",op_sim_time());
    ipg = (ipg * srtt) / (ipg + srtt);
    fprintf(statfile2, "%f;%f\n", ipg, srtt);
//1 printf("%f\n", ipg);
}

```

```
int IncreaseIpg(void)   //void
{
//1 printf("%f\t->\t", ipg);
    fprintf(statfile2,"%f;", op_sim_time());
    ipg = (ipg / beta);
//  ipg = (ipg * 3.0);
    fprintf(statfile2, "%f;%f\n", ipg, srtt);
//1 printf("%f\n", ipg);
}

void GenPacket(void)
{
    Packet* pkptr;
    int srcdst;

    pkptr = op_pk_create_fmt("CED_UDP");
    op_pk_bulk_size_set(pkptr, udpsize);       // Should be 6 * 32 for the fields, but for the simulation...

    // same source than destination, "connection" between same src and dst
        op_ima_obj_attr_get (my_self, "RAP_UDP_ADDRESS", &srcdst);
        op_pk_fd_set (pkptr, 0, OPC_FIELD_TYPE_INTEGER, srcdst, 0);
        op_pk_fd_set (pkptr, 1, OPC_FIELD_TYPE_INTEGER, srcdst, 0);

    // Incrementation and introduction of the seqNum
        seqnum++;
        op_pk_fd_set(pkptr, 4, OPC_FIELD_TYPE_INTEGER, seqnum, 0);

    SendPacket(pkptr);
}

int SendPacket(Packet* pkptr)  //void
{
    // Creation of the TransHistoryEntry
        TransHistoryEntry * temp = (TransHistoryEntry*)CreateTHE();

    temp->seqno = seqnum;
    temp->departureTime = op_sim_time();

        printf("%f\t%d in list\n", op_sim_time(), seqnum);

    // Introduction in the List
        Append(temp);

    // Send the rap packet
        totpack++;
        op_pk_send (pkptr, 1);
}

int RecvAck(Packet* pkptr) //void
{
```

```
    int seq;
    TransHistoryEntry *temp;

    // remove the entry number 'seqNum' (field 2) from transmission history table.
    op_pk_fd_get(pkptr, 2, &seq);

    temp = (TransHistoryEntry*)RemoveSeqno(seq);

    if(temp != NULL)
    {
        // Packet with such seqnum in the records table
        if(temp->state != 1)
        {
            // sample Rtt
            double samplertt = op_sim_time () - (temp->departureTime);

            fprintf(statfile, "%f;",samplertt);
            totack++;

            // update Rtt
            UpdateTimeValues(samplertt);

            // deallocate the memory for that entry
            free(temp);
        }
        if(LossDetection(1, pkptr, 0))
        {
            LossHandler(1);
        }
    }
    else
    {
        printf("%f\tPacket with such seqnum (%d) not in the records table\n", op_sim_time(), seq);
    }
    // Send to sink
    op_pk_send(pkptr, 0);
}

int UpdateTimeValues(double sample)    //void
{
    double diff;

    if(initial)
    {
        frtt = xrtt = srtt = sample;
        variance = 0;
        printf("UTV : Au premier ACK : time = %f\n", op_sim_time());
    }
    diff = sample - srtt;
//  srtt = delta*srtt + (1 - delta)*diff;                                    // test 3
```

```
//  srtt += delta * diff;       // Done this way based on the ns implementation code              // test 2

    srtt = (delta * srtt) + ((1 - delta) * sample);                                               // test 1
    fprintf(statfile, "%f;", srtt);

    diff = (diff <0) ? diff * -1 : diff;
    variance = variance + (delta * (diff - variance));

    if(lossoccur)   {timeout = (mu * srtt) + (phi * variance);}
    else            {timeout = 1.0;}

//  timeout = (mu * srtt) + (phi * variance);
//  if(initial) {timeout = timeout + 0.05;}     // 0.05 orsrtt DOUTE???????
//  timeout = 1.0;

    fprintf(statfile, "%f;", timeout);
    fprintf(statfile, "%f\n", variance);

    if(initial) {initial = FALSE;}

    if(finegrain)
    {
        frtt = ((1 - kfrtt) * frtt) + (kfrtt * sample);
        xrtt = ((1 - kxrtt) * xrtt) + (kxrtt * sample);
//      fprintf(statfile, "UTV : frtt = %d et xrtt = %d (finegrain)", frtt, xrtt);
    }
}

/***********************************************************************/
/*                                                                     */
/*   Gestion of the timers IPG (cod = 0) and RTT (cod = 1)             */
/*                                                                     */
/***********************************************************************/

int Timeout(int code)   //void
{
    switch(code)
    {
        case 0: //fprintf(statfile, "TO : IpgTimeout\n");
                IpgTimeout();
                break;

        case 1: //fprintf(statfile, "TO : RttTimeout\n");
                RttTimeout();
                break;

        default: printf("TO : Error of code in the intrpt_self (param of timeout)\n");
                 break;
    }
}
```

```
int RttTimeout(void)        //void  // one step done, must increase rate so decrease ipg
{
    int loss = 0;

    loss = LossDetection(0, NULL, 1);
    if((loss == 0) && (pacwhilesrtt != 0))
    {
        DecreaseIpg();
        pacwhilesrtt = 0;
    }
    else
    {
        fprintf(statfile2, "%f;%f;%f\n", op_sim_time(), ipg, srtt);
    }
    /* Re-scheduling of the intrpt_self */
    event = op_intrpt_schedule_self (op_sim_time() + srtt, 1); // RTT : confert text
}

int IpgTimeout(void)    //void
{
    double waitPeriod;
    int loss;

    loss = LossDetection(0, NULL, 0);
    if(loss)    // timer based losses detection
    {
        LossHandler(0);
    }
    else
    {
        GenPacket();
        pacwhilesrtt++;
    }

    if(finegrain)       // fine grain used
    {
        waitPeriod = (frtt / xrtt) * ipg; // frtt et xrtt initialised in init to 1 till the first ACK
    }
    else
    {
        waitPeriod = ipg;
    }
    // Schedule the next IpgTimeout for the generation of a new packet
    op_intrpt_schedule_self(op_sim_time() + waitPeriod, 0);
}

int LossHandler(int code)   //void
{
    TransHistoryEntry* curr;
```

```

/*1 if(code == 0)
    {
        fprintf(statfile, "LH : CONGESTION TLP at %f\n", op_sim_time());
    }
    else
    {
        if(code == 1)
        {
            fprintf(statfile, "LH : CONGESTION ALP at %f\n", op_sim_time());
        }
        else
        {
            fprintf(statfile, "LH : CONGESTION ??? at %f\n", op_sim_time());
        }
    }
1*/
    IncreaseIpg();

    // Put all the status in the Liste at 2 (= INACTIVE) to react only one time to a loss
    for(curr = list->first; curr != NULL; curr = curr->next)
    {
        curr->state = 2;
    }

    // cancel Rtt interruption done from RttTimeout no more needed
    op_ev_cancel(event);
    event = op_intrpt_schedule_self(op_sim_time() + srtt, 1); // To re-compute ipg
}

int LossDetection(int type, Packet* pkptr, int code)
{
    int numlosses;
    switch(type)
    {
        case 0: // RAP_TIMER_BASED
                if(code == 0)
                {
                    numlosses = TimerLostPacket(0);
                }
                else
                {
                    if(code == 1)
                    {
                        numlosses = TimerLostPacket(1);
                    }
                    else
                    {
                        printf("LD : Wrong code for TimerLostPacket\n");
                    }
```

```
                    }
                    if(numlosses)   printf("%f\tLOSS TIMER (%d)\t", op_sim_time(),  numlosses);
                    break;

          case 1: // RAP_ACK_BASED
                    numlosses = AckLostPacket(pkptr);
                    if(numlosses)   printf("%f\tLOSS ACK (%d)\t", op_sim_time(), numlosses);
                    break;

          default:// wrong code
                    printf("Bad type for loss detection: not RAP_TIMER_BASED nor RAP_ACK_BASED\n");
                    break;
     };

     // Purge of every packets with status = PURGED (1)
     Purge(1);
     if(numlosses)   {lossoccur = TRUE;}
     return(numlosses);
}

int TimerLostPacket(int code)
{
     int numlosses, session;
     TransHistoryEntry * curr;

     numlosses = 0;
     session = 0;
     for(curr = list->first; curr != NULL; curr = curr->next)
     {
          if((curr->departureTime + timeout) - op_sim_time() <= 0.000001)// loss in rap session
          {
               session += 1;
               if((curr->state) == 0)
               {
                    numlosses += 1;
               }
               if(code == 0)   {curr->state = 1;}
          }
     }
//   if(code == 0)   return(numlosses);
//   else            return(session);
     return(numlosses);
}

int AckLostPacket(Packet* pkptr)
{
     int numlosses;
     TransHistoryEntry *temp;

     int lr, lm, pr;
```

```
    op_pk_fd_get (pkptr, 3, &lr);
    op_pk_fd_get (pkptr, 4, &lm);
    op_pk_fd_get (pkptr, 5, &pr);

    numlosses = 0;
    for(temp = list->first; temp != NULL; temp = temp->next)
    {
        int seq = temp->seqno;

        if(seq <= lr)
        {
            if((seq > pr) && (seq <= lm))
            {
                if((lr - seq) >= 3)
                {
                    if(temp->state == 0)
                    {
                        numlosses++;
                    }
                    temp->state = 1;
                }
            }
        }
    }
    return(numlosses);
}

/* ****************************************************************************** */
/*                                                                                */
/*              FUNCTIONS RELATED TO THE LIST OF RECORDS (Transmission Table)     */
/*                                                                                */
/* ****************************************************************************** */

/* ******************************************************************************
    Append a TransHistoryEntry to the end of the Liste    must use ListElmnt before!!!
    ****************************************************************************** */
int Append(TransHistoryEntry* item)//void
{
    if(IsEmpty())
    {
        list->first = item;
        list->last = item;
    }
    else
    {
        (list->last)->next = item;
        list->last = item;
    }
    list->size =list->size + 1;
}
```

```c

/* ****************************************************************************************
                    Init a Liste
   **************************************************************************************** */
int CreateList(void)    //void
{
    list = (Liste*)malloc(sizeof(Liste));
    list->first = NULL;
    list->last = NULL;
    list->size = 0;
}

/* ****************************************************************************************
    Create an element of type TransHistoryEntry
        default values :    state = 0 (SENT)
                            next = NULL
   **************************************************************************************** */
TransHistoryEntry* CreateTHE(void)
{
    TransHistoryEntry * the = (TransHistoryEntry*)malloc(sizeof(TransHistoryEntry));
    // seqno not init
    the->state = 0;
    // departureTime not init
    the->next = NULL;
    return (the);
}

/* ****************************************************************************************
    DisplayAllList all info of the elements of the list
        return "end of list" when list is empty
   **************************************************************************************** */
int DisplayAllList(void)    //void
{
    TransHistoryEntry* curr;

    fprintf(statfile, "Display : \n");
    for(curr = list->first; curr != NULL; curr = curr->next)
    {
        fprintf(statfile, "%d\t%d\t%f\n", curr->seqno, curr->state, curr->departureTime);
    }
}

/* ****************************************************************************************
    DisplayList all seqNum of the elements of the list
        return "end of list" when list is empty
   **************************************************************************************** */
int DisplayList(void)   //void
{
    TransHistoryEntry* curr;

```

```
    fprintf(statfile, "Display : ");
    for(curr = list->first; curr != NULL; curr = curr->next)
    {
        fprintf(statfile, "%d\t", curr->seqno);
    }
    fprintf(statfile, "end of list\n");
}

/* **************************************************************************************
                 Test if the Liste is empty,
                    return TRUE if empty
   ************************************************************************************** */
int IsEmpty(void)
{
    if(list->first == NULL) return(TRUE);
    else return(FALSE);
}

/* **************************************************************************************
         Test if the TransHistoryEntry with seqno = keyseq is in the Liste
            return TRUE if in
   ************************************************************************************** */
int IsPresentSeqno(int keyseq)
{
    TransHistoryEntry* temp;
    for(temp = list->first;temp != NULL; temp = temp->next)
    {
        if(temp->seqno == keyseq)  return(TRUE);
    }
    return(FALSE);
}

/* **************************************************************************************
         Test if the TransHistoryEntry with state = keysta is in the Liste
            return TRUE if in
   ************************************************************************************** */
int IsPresentState(int keysta)
{
    TransHistoryEntry* temp;
    for(temp = list->first;temp != NULL; temp = temp->next)
    {
        if(temp->state == keysta)  return(TRUE);
    }
    return(FALSE);
}

/* ************************************************************************************
        Find in the Liste the THE with seqno = keyseq, NULL if not in
            return TransHistoryEntry* if found
            return NULL otherway
```

```
   ****************************************************************************************** */
TransHistoryEntry* Find(int keyseq)
{
    if((IsEmpty()) || (!IsPresentSeqno(keyseq)))
    {
        return(NULL);
    }
    else
    {
        TransHistoryEntry * temp;
        for(temp = list->first; temp != NULL; temp = temp->next)
        {
            if((temp->seqno) == keyseq) return(temp);
        }
    }
}

/* ******************************************************************************************
                    Deallocate a Liste
   ****************************************************************************************** */
int KillList(void) //void
{
    TransHistoryEntry *temp = (TransHistoryEntry*)Remove();
    while(temp != NULL)
    {
        free(temp);
        temp = (TransHistoryEntry*)Remove();
    }
}

/* ******************************************************************************************
         Destroy all TransHistoryEntry with state = keysta from the Liste
   ****************************************************************************************** */
int Purge(int keysta)  //void
{
    TransHistoryEntry *item, *prec, *curr;

    if(!IsEmpty() && IsPresentState(keysta))
    {
        prec = NULL;
        curr = list->first;
        while( curr != NULL )
        {
            if(curr->state == keysta)
            {
                if(curr == list->first)     // the first one to destroy
                {
                    item = curr;
                    list->first = curr->next;
                    item->next = NULL;
```

```
                    curr = list->first;
                    printf("%d\n", item->seqno);
                    free(item);
                }
                else
                {
                    if(curr == list->last) // the last one to destroy
                    {
                        item = curr;
                        prec->next = NULL;
                        list->last = prec;
                        curr = NULL;
                        printf("%d\n", item->seqno);
                        free(item);
                    }
                    else            // one in the middle to destroy
                    {
                        item = curr;
                        prec->next = curr->next;
                        item->next = NULL;
                        curr = prec->next;
                        printf("%d\n", item->seqno);
                        free(item);
                    }
                }
                list->size = list->size - 1;
            }
            else
            {
                prec = curr;
                curr = curr->next;
            }
        }
    }
//  printf("END\n");
}

/* ***********************************************************************************
             Remove the first TransHistoryEntry from the Liste,
                return TransHistoryEntry* removed
                return NULL if Liste is empty
   *********************************************************************************** */
TransHistoryEntry* Remove(void)
{
    TransHistoryEntry* item;

    if(IsEmpty())
    {
        return(NULL);
    }
```

```
    else
    {
        item = list->first;
        if(list->size == 1)
        {
            list->first = NULL;
            list->last = NULL;
        }
        else
        {
            list->first = item->next;
            item->next = NULL;
        }
        list->size = list->size - 1;
        return(item);
    }
}

/* ******************************************************************************************
     Remove the TransHistoryEntry with seqno = keyseq from the front of the Liste,
        return TransHistoryEntry* removed
        return NULL if Liste is empty or no such TransHistoryEntry
   ****************************************************************************************** */
TransHistoryEntry* RemoveSeqno(int keyseq)
{
    TransHistoryEntry *prec, *curr, *item;

    if(!IsEmpty() && IsPresentSeqno(keyseq))
    {
        for(prec = NULL, curr = list->first; curr != NULL; prec = curr, curr = curr->next)
        {
            if(curr->seqno == keyseq)
            break;
        }
    }
    else
    {
        return(NULL);
    }
    if(curr == list->first)     // the first one to remove
    {
        item = Remove();
    }
    else
    {
        item = curr;
        if(curr == list->last) // the last one to remove
        {
            prec->next = NULL;
            list->last = prec;
```

```
    }
    return(item);
}

/* ***************************************************************************************
    SizeofList displays the compute sizeof the list (not list->size)
   ***************************************************************************************/
int SizeofList(void)    // void
{
    TransHistoryEntry* curr;
    int taille = 0;

    for(curr = list->first; curr != NULL; curr = curr->next)
    {
        taille++;
    }
    fprintf(statfile, "list's size = %dand COMPUTED size = %d\n", list->size, taille);
}
```

```
            }
            else
            {
                prec->next = curr->next;
                curr->next = NULL;
            }
            list->size = list->size - 1;
        }
        return(item);
    }

    /* ********************************************************************************************
         Remove the TransHistoryEntry with state = keysta from the front of the Liste,
            return TransHistoryEntry* removed
            return NULL if Liste is empty or no such TransHistoryEntry
       ******************************************************************************************** */
    TransHistoryEntry* RemoveState(int keysta)
    {
        TransHistoryEntry *prec, *curr, *item;

        if(!IsEmpty() && IsPresentState(keysta))
        {
            for(prec = NULL, curr = list->first; curr != NULL; prec = curr, curr = curr->next)
            {
                if(curr->state == keysta)
                break;
            }
        }
        else
        {
            return(NULL);
        }
        if(curr == list->first)     // the first one to remove
        {
            item = Remove();
        }
        else
        {
            item = curr;
            if(curr == list->last)  // the last one to remove
            {
                prec->next = NULL;
                list->last = prec;
            }
            else
            {
                prec->next = curr->next;
                curr->next = NULL;
            }
            list->size = list->size - 1;
```

```
Objid cousin, papa;
int statONOFF;
int durtime, i;

my_self = op_id_self();
op_ima_obj_attr_get(my_self, "name", nameSRC);
papa = op_id_parent(my_self);
op_ima_obj_attr_get(papa, "name", nameISP);

totpack = 0;
totack = 0;

/* Getting the general attributs which configure the RAP protocol */
op_ima_obj_attr_get (my_self, "Fine Grain use", &finegrain);  // ...
op_ima_obj_attr_get (my_self, "BETA",           &beta);       // IncreaseIpg
op_ima_obj_attr_get (my_self, "KFRTT",          &kfrtt);      // Weigth of samplertt in frtt
op_ima_obj_attr_get (my_self, "KXRTT",          &kxrtt);      // Weigth of samplertt in xrtt
op_ima_obj_attr_get (my_self, "MU",             &mu);         // Compute timeout for check losses based on timer
op_ima_obj_attr_get (my_self, "PHI",            &phi);        // Compute timeout for check losses based on timer
op_ima_obj_attr_get (my_self, "DELTA",          &delta);      // Compute variance and srtt
op_ima_obj_attr_get (my_self, "Start time",     &starttime);  // Simulation start time
op_ima_obj_attr_get (my_self, "Duration time", &durtime);     // Duration time of the simulation
op_ima_obj_attr_get (my_self, "UDP size",       &udpsize);    // Size of UDP packets

CreateList();

initial = TRUE;     // first time of initialisation for variables frtt, xrtt, srtt, variance
lossoccur = FALSE;  // To start computing the timeout after probing the network (good estimated rtt)
seqnum  = 0;        // sequence number of rap packet
xrtt = frtt = 1;    // initialised to one in case of using finegrain before receiving the first ACK
timeout = 1.0;
ipg = 0.05;
srtt = 0.5;
pacwhilesrtt = 0;

if(durtime != 0)
{
    // Code 1 interruption to be sure to have sth to cancel in case of timeout triggered
    event = op_intrpt_schedule_self(srtt + op_sim_time() + starttime, 1); // RTT

    // Code 0 interruption for the first packet
    op_intrpt_schedule_self(op_sim_time() + starttime , 0);
}

// File to collect statistics for graph.
statfile = (FILE*)0;
op_ima_obj_attr_get (my_self, "stat_file",     &statONOFF);
if(statONOFF)
{
    char file[30];
```

```

    graph = fopen(graphfile,"w");
    if(graph == NULL)    {printf("Graph file not well opened!!!\n");}
    else                 {fprintf(graph, "%s\n", file);}

    stepfile[0] = '\0';
    strcat(stepfile, "/home/users/rosmanc/simul/SIMULATION/step");
    strcat(stepfile, file);
    strcat(stepfile, ".csv");

    statfile2 = fopen(stepfile,"w");
    if(statfile2 == NULL)   {printf("Step file not well opened!!!\n");}
    else                    {fprintf(statfile2, "%s\n", file);fprintf(statfile2, "Time;ipg;srtt\n", file);}
}

for(i = 0; i < durtime - 1; i++)
{
    op_intrpt_schedule_self( i + 0.25, 3);
    op_intrpt_schedule_self( i + 0.5 , 3);
    op_intrpt_schedule_self( i + 0.75, 3);
    op_intrpt_schedule_self( i + 1.0 , 3);
}
    op_intrpt_schedule_self( i + 0.25, 3);
    op_intrpt_schedule_self( i + 0.5 , 3);
    op_intrpt_schedule_self( i + 0.75, 3);
    /* till 104.75 */

printf("INITIALISATION OF RAP COMPLETED!!! (%d)\n", udpsize);

```

```
        char fullname[100];
        char graphfile[100];
        char stepfile[100];

        fullname[0] = '\0';
        file[0] = '\0';

        strcat(fullname, "/home/users/rosmanc/simul/SIMULATION/");
        op_ima_obj_attr_get (my_self, "file_name", file);
        strcat(fullname, file);
        strcat(fullname, ".csv");
        statfile = fopen(fullname, "w");
        if(statfile == NULL){printf("Stat file not well opened!!!\n");}
        else
        {
            fprintf(statfile, "%s\n", file);

            fprintf(statfile, ";;****************************************\n");
            fprintf(statfile, ";;** SIMULATION PARAMETERS FOR %s **\n", file);
            fprintf(statfile, ";;****************************************\n\n");

            fprintf(statfile, ";Attributs:\n");
            if(finegrain)   {   fprintf(statfile, ";;Fine Grain;Yes\n");}
            else            {   fprintf(statfile, ";;Fine Grain;No\n");}
            fprintf(statfile, ";;BETA;%f\n", beta);
            fprintf(statfile, ";;KFRTT;%f\n", kfrtt);
            fprintf(statfile, ";;KXRTT;%f\n", kxrtt);
            fprintf(statfile, ";;MU;%f\n", mu);
            fprintf(statfile, ";;PHI;%f\n", phi);
            fprintf(statfile, ";;DELTA;%f\n\n", delta);

            fprintf(statfile, ";Variables:\n");
            fprintf(statfile, ";;ipg;%f\n", ipg);
            fprintf(statfile, ";;timeout;%f\n", timeout);
            fprintf(statfile, ";;srtt;%f\n", srtt);
            fprintf(statfile, ";;udp size;%d\n\n", udpsize);
            fprintf(statfile, ";;duration t.;%d\n", durtime);
            fprintf(statfile, ";;start time;%f\n\n", starttime);

            fprintf(statfile, ";;**********************\n");
            fprintf(statfile, ";;** SIMULATION DEBUG **\n");
            fprintf(statfile, ";;**********************\n\n");

            fprintf(statfile, "Sample;Smoothrtt;Timeout\n");
        }

        graphfile[0] = '\0';
        strcat(graphfile, "/home/users/rosmanc/simul/SIMULATION/graph");
        strcat(graphfile, file);
        strcat(graphfile, ".csv");
```

intrpt : Enter Execs

12:21:03 Jan 26 2001 | 1/1

```
// Interruption from op_intrpt_schedule_self(code)
// where code == 0  for IPGinterrupt
//     or code == 1  for RTTinterrupt
// Call 'Timeout' function

Timeout(op_intrpt_code());
```

stat : Enter Execs

12:21:15 Jan 26 2001 | 1/1

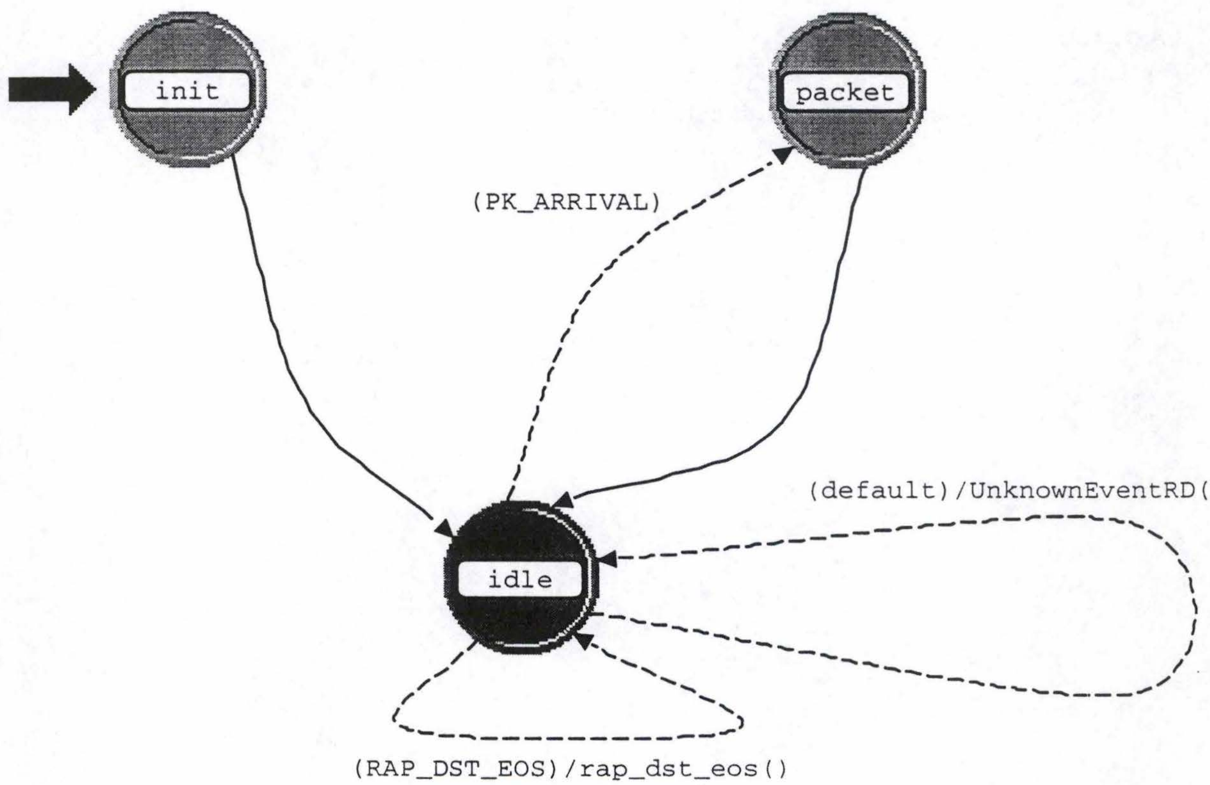
```
int stat;

op_ima_obj_attr_get(my_self, "stat_file", &stat);
if(stat)
{
    fprintf(graph, "%d\n", totpack);
}
```

```
Packet* pkptr;
char fmt[30];

/* Getting the packet */
pkptr = op_pk_get (op_intrpt_strm ());

/* Could be udp packet or rap ack packet */
op_pk_format (pkptr, fmt);
if (strcmp(fmt, "CED_RAP_FORMAT_ACK") == 0)// so it's rap ack
{
//  fprintf(statfile, "pa : ACK (lance RecvAck)\n");
    RecvAck(pkptr);
}
else                                        // so it's bad packet
{
        printf("pa : Wrong format in RAP\n");
        /* Destroy the packet with wrong format */
        /* Be carefull if stcp packet */

        if(strcmp(fmt, "stcp_ippkt") == 0)
        {
            op_stcp_discard_packet(pkptr);
        }
        else
        {
            op_pk_destroy(pkptr);
        }
}
```

B.3.2 Destinations

As we can see, only the "Idle" state is unforced i.e. the process stops after the execution of the enter section. Next pages give the complete code of *RAP* in OPNET running behind the 5 different processes.

Process Model: CED_RAP_DST



```
/* Process model C form file: CED_RAP_DST.pr.c */
/* Portions of this file copyright 1992-2000 by OPNET, Inc. */



/* This variable carries the header into the object file */
static const char CED_RAP_DST_pr_c [] = "MIL_3_Tfile_Hdr_ 70B 30A modeler 7 3A66AB84 3
#include <string.h>



/* OPNET system definitions */
#include <opnet.h>

#if defined (__cplusplus)
extern "C" {
#endif
FSM_EXT_DECS
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif


/* Header Block */

#define RAP_DST_EOS (op_intrpt_type () == OPC_INTRPT_ENDSIM)
#define PK_ARRIVAL  (op_intrpt_type () == OPC_INTRPT_STRM)

/* End of Header Block */


#if !defined (VOSD_NO_FIN)
#undef  BIN
#undef  BOUT
#define BIN     FIN_LOCAL_FIELD(last_line_passed) = __LINE__ - _block_origin;
#define BOUT    BIN
#define BINIT   FIN_LOCAL_FIELD(last_line_passed) = 0; _block_origin = __LINE__;
#else
#define BINIT
#endif /* #if !defined (VOSD_NO_FIN) */



/* State variable definitions */
typedef struct
    {
    /* Internal state tracking for FSM */
    FSM_SYS_STATE
    /* State Variables */
    int                          lastRecv;
    int                          lastMiss;
    int                          prevRecv;
    Objid                        my_self;
    int                          totpack;
    int                          totack;
    int                          acksize;
    } CED_RAP_DST_state;

#define pr_state_ptr                    ((CED_RAP_DST_state*) SimI_Mod_State_Ptr)
#define lastRecv                        pr_state_ptr->lastRecv
#define lastMiss                        pr_state_ptr->lastMiss
#define prevRecv                        pr_state_ptr->prevRecv
#define my_self                         pr_state_ptr->my_self
#define totpack                         pr_state_ptr->totpack
#define totack                          pr_state_ptr->totack
#define acksize                         pr_state_ptr->acksize

```

```
/* This macro definition will define a local variable called  */
/* "op_sv_ptr" in each function containing a FIN statement.*/
/* This variable points to the state variable data structure, */
/* and can be used from a C debugger to display their values. */
#undef FIN_PREAMBLE
#define FIN_PREAMBLE    CED_RAP_DST_state *op_sv_ptr = pr_state_ptr;


/* Function Block */

enum { _block_origin = __LINE__ };
void UnknownEventRD(void)
{
    Objid papa;
    char nameDST[15];
    char nameISP[15];

    op_ima_obj_attr_get(my_self, "name", nameDST);
    papa = op_id_parent(my_self);
    op_ima_obj_attr_get(papa, "name", nameISP);

    printf("UnknownEvent dans %s\t%s\n", nameISP, nameDST);
    op_sim_end("END OF SIM : "," UNKNOWNEVENT IN ","CED RAP DST","");
}

void rap_dst_eos(void)
{
    Objid papa;
    char nameDST[15];
    char nameISP[15];

    op_ima_obj_attr_get(my_self, "name", nameDST);
    papa = op_id_parent(my_self);
    op_ima_obj_attr_get(papa, "name", nameISP);

    if(totpack != 0)printf("In %s, %s received %d packets and sent %d ACKS\n", nameISP
}

//-----------------------------------------------------------------------
// UpdateLastHole
//      Update the last hole in sequence number space at the receiver.
//      "seqNum" is the sequence number of the data packet received.
//-----------------------------------------------------------------------
void UpdateLastHole(int seqNum)
{
    if (seqNum > (lastRecv + 1))                          // Loss occurs (1 or more)
    {
        prevRecv = lastRecv;
        lastRecv = seqNum;
        lastMiss = seqNum - 1;
        return;
    }

    if (seqNum == (lastRecv + 1))                         // Received in sequence
    {
        lastRecv = seqNum;
        return;
    }

    if ((lastMiss < seqNum) && (seqNum <= lastRecv))    // Duplicate
    {
        return;
    }

    if (seqNum == lastMiss)
    {
        if ((prevRecv + 1) == seqNum)                     // Hole of 1 packet filled
```

```
            {
                prevRecv = 0;
                lastMiss = 0;
            }
            else                            // Hole of [n..n+m] packets (m>1) -> [n..n+m-1]
            {
                lastMiss--;
            }
            return;
        }

        if ((prevRecv < seqNum) && (seqNum < lastMiss))     // Packet received in a hole
        {
            prevRecv = seqNum;
            return;
        }
}

/* End of Function Block */

#if defined (__cplusplus)
extern "C" {
#endif
    void CED_RAP_DST (void);
    Compcode CED_RAP_DST_init (void **);
    void CED_RAP_DST_diag (void);
    void CED_RAP_DST_terminate (void);
    void CED_RAP_DST_svar (void *, const char *, char **);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif




/* Process model interrupt handling procedure */


void
CED_RAP_DST (void)
    {
    int _block_origin = 0;
    FIN (CED_RAP_DST ());
    if (1)
        {



        FSM_ENTER (CED_RAP_DST)

        FSM_BLOCK_SWITCH
            {
            /*---------------------------------------------------------*/
            /** state (init) enter executives **/
            FSM_STATE_ENTER_FORCED (0, state0_enter_exec, "init", "CED_RAP_DST () [ini
                {
                my_self = op_id_self();

                /* init of variables for hole informations*/
                lastRecv = 0;       // see SV comment
                lastMiss = 0;       // see SV comment
                prevRecv = 0;       // see SV comment

                totpack = 0;
                totack = 0;

                op_ima_obj_attr_get(my_self, "ACK Size", &acksize);
```

```
          }


     /** state (init) exit executives **/
     FSM_STATE_EXIT_FORCED (0, state0_exit_exec, "init", "CED_RAP_DST () [init
          {

          }


     /** state (init) transition processing **/
     FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
          /*---------------------------------------------------------*/



     /** state (idle) enter executives **/
     FSM_STATE_ENTER_UNFORCED (1, state1_enter_exec, "idle", "CED_RAP_DST () [i
          {

          }


     /** blocking after enter executives of unforced state. **/
     FSM_EXIT (3,CED_RAP_DST)


     /** state (idle) exit executives **/
     FSM_STATE_EXIT_UNFORCED (1, state1_exit_exec, "idle", "CED_RAP_DST () [idl
          {

          }


     /** state (idle) transition processing **/
     FSM_INIT_COND (PK_ARRIVAL)
     FSM_TEST_COND (RAP_DST_EOS)
     FSM_DFLT_COND
     FSM_TEST_LOGIC ("idle")

     FSM_TRANSIT_SWITCH
          {
          FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;)
          FSM_CASE_TRANSIT (1, 1, state1_enter_exec, rap_dst_eos();)
          FSM_CASE_TRANSIT (2, 1, state1_enter_exec, UnknownEventRD();)
          }
          /*---------------------------------------------------------*/



     /** state (packet) enter executives **/
     FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "packet", "CED_RAP_DST () [p
          {
          Objid papa;
          Packet* pkptrRecv, *pkptrAck;
          char nameISP[200];
          char nameSRC[200];
          char fmt[30];
          int dest, src, seqNum;
          int srct, destt, seqnumt, lrt, lmt, prt;

          op_ima_obj_attr_get(my_self, "name", nameSRC);
          papa = op_id_parent(my_self);
          op_ima_obj_attr_get(papa, "name", nameISP);

          /* Pick up the packet */
          pkptrRecv = op_pk_get (op_intrpt_strm ());
```

```
                    op_pk_format(pkptrRecv, fmt);
                    if(strcmp(fmt, "CED_UDP") == 0)
                    {
                        totpack++;

                        /* RAP's seqnum */
                        op_pk_fd_get (pkptrRecv, 4, &seqNum);


                        /* Update info about hole in packets sequence */
                        UpdateLastHole(seqNum);

                        /* Generate and Send ACK */
                            /* creation of the ack packet*/
                            pkptrAck = op_pk_create_fmt ("CED_RAP_FORMAT_ACK");
                            op_pk_bulk_size_set (pkptrAck, acksize);   // 320 (ATTR)

                            /* Getting the fields values of pkptrRecv */
                            op_pk_fd_get (pkptrRecv, 0, &dest);
                            op_pk_fd_get (pkptrRecv, 1, &src);

                            /* Init of the fields of the ack packet */
                            op_pk_fd_set (pkptrAck, 0, OPC_FIELD_TYPE_INTEGER, src, 0);
                            op_pk_fd_set (pkptrAck, 1, OPC_FIELD_TYPE_INTEGER, dest, 0);
                            op_pk_fd_set (pkptrAck, 2, OPC_FIELD_TYPE_INTEGER, seqNum, 0);
                            op_pk_fd_set (pkptrAck, 3, OPC_FIELD_TYPE_INTEGER, lastRecv, 0
                            op_pk_fd_set (pkptrAck, 4, OPC_FIELD_TYPE_INTEGER, lastMiss, 0
                            op_pk_fd_set (pkptrAck, 5, OPC_FIELD_TYPE_INTEGER, prevRecv, 0

                        /* Send packet to sink via output stream 0 (manual config)*/
                        op_pk_send_quiet(pkptrRecv,0);

                        totack++;
                        /* Send the ack */
                        op_pk_send(pkptrAck, 1);
                    }
                    else printf("Wromg type of packet received at RAP dest (%s)\n", fmt);
                    }


                /** state (packet) exit executives **/
                FSM_STATE_EXIT_FORCED (2, state2_exit_exec, "packet", "CED_RAP_DST () [pac
                    {

                    }


                /** state (packet) transition processing **/
                FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
                    /*---------------------------------------------------------*/




                }


            FSM_EXIT (0,CED_RAP_DST)
            }
        }

#if defined (__cplusplus)
    extern "C" {
#endif
    extern VosT_Fun_Status Vos_Catmem_Register (const char * , int , VosT_Void_Null_Pr
    extern VosT_Address Vos_Catmem_Alloc (VosT_Address, size_t);
    extern VosT_Fun_Status Vos_Catmem_Dealloc (VosT_Address);
#if defined (__cplusplus)
```

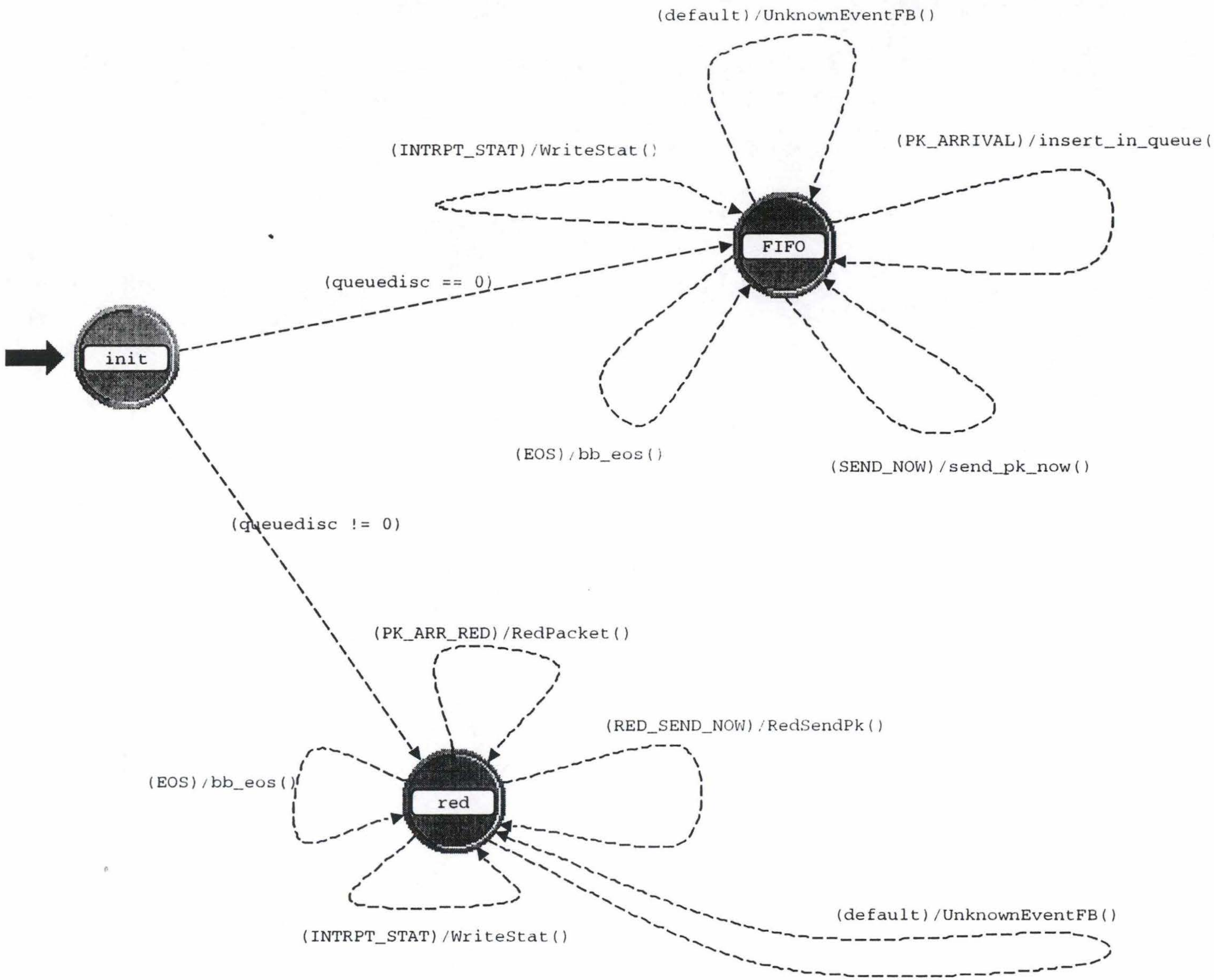
```
        }
#endif


Compcode
CED_RAP_DST_init (void ** gen_state_pptr)
    {
    int _block_origin = 0;
    static VosT_Address obtype = OPC_NIL;

    FIN (CED_RAP_DST_init (gen_state_pptr))

    if (obtype == OPC_NIL)
        {
        /* Initialize memory management */
        if (Vos_Catmem_Register ("proc state vars (CED_RAP_DST)",
            sizeof (CED_RAP_DST_state), Vos_Vnop, &obtype) == VOSC_FAILURE)
            {
            FRET (OPC_COMPCODE_FAILURE)
            }
        }

    *gen_state_pptr = Vos_Catmem_Alloc (obtype, 1);
    if (*gen_state_pptr == OPC_NIL)
        {
        FRET (OPC_COMPCODE_FAILURE)
        }
    else
        {
        /* Initialize FSM handling */
        ((CED_RAP_DST_state *)(*gen_state_pptr))->current_block = 0;

        FRET (OPC_COMPCODE_SUCCESS)
        }
    }



void
CED_RAP_DST_diag (void)
    {
    /* No Diagnostic Block */
    }




void
CED_RAP_DST_terminate (void)
    {
    int _block_origin = __LINE__;

    FIN (CED_RAP_DST_terminate (void))

    if (1)
        {


        /* No Termination Block */

        }
    Vos_Catmem_Dealloc (pr_state_ptr);

    FOUT;
    }
```

```
/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in CED_RAP_DST_svar function. */
#undef lastRecv
#undef lastMiss
#undef prevRecv
#undef my_self
#undef totpack
#undef totack
#undef acksize



void
CED_RAP_DST_svar (void * gen_ptr, const char * var_name, char ** var_p_ptr)
	{
	CED_RAP_DST_state		*prs_ptr;

	FIN (CED_RAP_DST_svar (gen_ptr, var_name, var_p_ptr))

	if (var_name == OPC_NIL)
		{
		*var_p_ptr = (char *)OPC_NIL;
		FOUT;
		}
	prs_ptr = (CED_RAP_DST_state *)gen_ptr;

	if (strcmp ("lastRecv" , var_name) == 0)
		{
		*var_p_ptr = (char *) (&prs_ptr->lastRecv);
		FOUT;
		}
	if (strcmp ("lastMiss" , var_name) == 0)
		{
		*var_p_ptr = (char *) (&prs_ptr->lastMiss);
		FOUT;
		}
	if (strcmp ("prevRecv" , var_name) == 0)
		{
		*var_p_ptr = (char *) (&prs_ptr->prevRecv);
		FOUT;
		}
	if (strcmp ("my_self" , var_name) == 0)
		{
		*var_p_ptr = (char *) (&prs_ptr->my_self);
		FOUT;
		}
	if (strcmp ("totpack" , var_name) == 0)
		{
		*var_p_ptr = (char *) (&prs_ptr->totpack);
		FOUT;
		}
	if (strcmp ("totack" , var_name) == 0)
		{
		*var_p_ptr = (char *) (&prs_ptr->totack);
		FOUT;
		}
	if (strcmp ("acksize" , var_name) == 0)
		{
		*var_p_ptr = (char *) (&prs_ptr->acksize);
		FOUT;
		}
	*var_p_ptr = (char *)OPC_NIL;

	FOUT;
	}
```

B.3.3 Routers

The state machine just engages the queuing policy based on the initial choice.

Process Model: CED_queue_fifo_BB1_buff



```
int \buff_size;

double  \servicerate;

int \use_buffer;

FILE*   \statfile;

int \tcp_drop[NUMBER_TCP_SRC*2];

int \tcp_for[NUMBER_TCP_SRC*2];

int \udp_drop[NUMBER_UDP_SRC*2];

int \udp_for[NUMBER_UDP_SRC*2];

int \queuedisc;

double  \min_th;

double  \max_th;

double  \drop_max;

double  \aveqsize;

double  \queuesize;

double  \weight;

FILE*   \fred;

double  \count;

FILE*   \ffifo;

int \totpacketserved;

int \totpacketarrived;

int \IN512[NUMBER_TCP_SRC*2];

int \DROP512[NUMBER_TCP_SRC*2];

int \IN800[NUMBER_TCP_SRC*2];

int \DROP800[NUMBER_TCP_SRC*2];

int \IN512size;

int \IN800size;

int \DROP512size;

int \DROP800size;
```

```
#define PK_ARRIVAL        ( op_intrpt_type() == OPC_INTRPT_STRM   )
#define SEND_NOW          ((op_intrpt_type() == OPC_INTRPT_SELF   ) && (op_intrpt_code() == 0))
#define EOS               ( op_intrpt_type() == OPC_INTRPT_ENDSIM)

// Total number of udp sources (need to be adapted with the network configuration)Number of ISP
#define NUMBER_UDP_SRC 5 → 6
// Total number of tcp sources (need to be adapted with the network configuration)Number of ISP
#define NUMBER_TCP_SRC 5 → 6

#define RED_SEND_NOW      ((op_intrpt_type() == OPC_INTRPT_SELF) && (op_intrpt_code() == 1))
#define PK_ARR_RED        ( op_intrpt_type() == OPC_INTRPT_STRM)

#define INTRPT_STAT       ((op_intrpt_type() == OPC_INTRPT_SELF) && (op_intrpt_code() == 3))
```

```
void UnknownEventFB()
{
    fprintf(statfile,"Error: UNKNOWNEVENT in BB1\n");
    op_sim_end("END OF SIMULATION", "IN BB1", "UNKNOWN EVENT", "");
}

void bb_eos(void)
{
    int i;

//p fprintf(statfile, "\nfffffffffffffffffffffffffffffffffffff\n\n");
    if(NUMBER_UDP_SRC > 0)
    {
        for(i = 0; i < (NUMBER_UDP_SRC*2); i++)
        {
            fprintf(statfile, "UDP%d OK;%d\n"  , i*2, udp_for[i]);
//          fprintf(statfile, "%d\n", udp_for[i]);
            fprintf(statfile, "UDP%d DROP;%d\n", i*2, udp_drop[i]);
//          fprintf(statfile, "%d\n", udp_drop[i]);
        }
        fprintf(statfile, "\n");
    }

    if(NUMBER_TCP_SRC > 0)
    {
        for(i = 0; i < (NUMBER_TCP_SRC*2); i++)
        {
            fprintf(statfile, "TCP%d OK;%d\n"  , i*2, tcp_for[i]);
            fprintf(statfile, "512 TCP%d OK;%d\n"  , i*2, IN512[i]);
            fprintf(statfile, "800 TCP%d OK;%d\n"  , i*2, IN800[i]);
            fprintf(statfile, "TCP%d DROP;%d\n", i*2, tcp_drop[i]);
            fprintf(statfile, "512 TCP%d DROP;%d\n", i*2, DROP512[i]);
            fprintf(statfile, "800 TCP%d DROP;%d\n", i*2, DROP800[i]);

            fprintf(statfile, "\n");
        }
        fprintf(statfile, "TCP512;INsize;%d\n",IN512size);
        fprintf(statfile, "TCP800;INsize;%d\n",IN800size);
        fprintf(statfile, "TCP512;DROPsize;%d\n",DROP512size);
        fprintf(statfile, "TCP800;DROPsize;%d\n",DROP800size);

        fprintf(statfile, "\n");
    }
    fprintf(statfile, "END_OF_SIMULATION\n");

    if(fclose(statfile) != 0)  {fprintf(statfile, "Stat file badly closed!!!\n");}

    if(queuedisc != 0)
    {
        int in1, in2, drop1, drop2;
```

```
        in1 = in2 = drop1 = drop2 = 0;

        fprintf(fred, "%f;;;%f\n", queuesize, aveqsize);
        fprintf(fred, "ARRIVED;%d\n", totpacketarrived);
        fprintf(fred, "SERVED;%d\n", totpacketserved);

        for(i = 0; i < (NUMBER_TCP_SRC*2); i++)
        {
            in1 = in1 + IN512[i];
            in2 = in2 + IN800[i];
            drop1 = drop1 + DROP512[i];
            drop2 = drop2 + DROP800[i];
        }
        fprintf(fred, "IN;pac 512 = ;%d;totsize = ;%d\n", in1, IN512size);
        fprintf(fred, "IN;pac 800 = ;%d;totsize = ;%d\n", in2, IN800size);
        fprintf(fred, "DROP;pac 512 = ;%d;totsize = ;%d\n", drop1, DROP512size);
        fprintf(fred, "DROP;pac 800 = ;%d;totsize = ;%d\n", drop2, DROP800size);
        fprintf(fred, "END_OF_SIMULATION\n");

        if(fclose(fred) != 0)  {printf("Red stat file badly closed!!!\n");}
    }
    else
    {
        int in1, in2, drop1, drop2;
        in1 = in2 = drop1 = drop2 = 0;

        fprintf(ffifo, "%d\n", buff_size);
        fprintf(ffifo, "ARRIVED;%d\n", totpacketarrived);
        fprintf(ffifo, "SERVED;%d\n", totpacketserved);
        for(i = 0; i < (NUMBER_TCP_SRC*2); i++)
        {
            in1 = in1 + IN512[i];
            in2 = in2 + IN800[i];
            drop1 = drop1 + DROP512size;
            drop2 = drop2 + DROP800size;
        }
        fprintf(ffifo, "IN;pac 512 = ;%d;totsize = ;%d\n", in1, IN512size);
        fprintf(ffifo, "IN;pac 800 = ;%d;totsize = ;%d\n", in2, IN800size);
        fprintf(ffifo, "DROP;pac 512 = ;%d;totsize = ;%d\n", drop1, DROP512size);
        fprintf(ffifo, "DROP;pac 800 = ;%d;totsize = ;%d\n", drop2, DROP800size);
        fprintf(ffifo, "END_OF_SIMULATION\n");

        if(fclose(ffifo) != 0) {printf("fifo stat file badly closed!!!\n");}
    }
}

/* FIFO mode: arrival of packet */
void insert_in_queue(void)
{
    Packet* pkptr;
```

```
    int pklength;
    char fmt[30];
    int vpi;

    pkptr = op_pk_get(op_intrpt_strm());
    op_pk_format(pkptr, fmt);
    pklength = op_pk_bulk_size_get(pkptr);

//  if((strcmp(fmt, "CED_UDP") == 0) || (strcmp(fmt, "stcp_ippkt") == 0)) totpacketarrived++;
    totpacketarrived++;

    op_pk_fd_get(pkptr, 0, &vpi);

    if(buff_size - pklength >= 0)
    {
        if(strcmp(fmt, "CED_UDP") == 0)
        {
//p           fprintf(statfile, "Bingo UDP%d\n", vpi);
            udp_for[vpi/2] += 1;
        }
        else
        {
            if(strcmp(fmt, "stcp_ippkt") == 0)
            {
                if(pklength <= 512)
                {
                    IN512[vpi/2]++;//numstrpac512IN++;
                    IN512size += pklength;
                }
                else
                {
                    if((pklength > 512) && (pklength < 800))
                    {
                        IN800[vpi/2]++;//numstrpac800IN++;
                        IN800size += pklength;
                    }
                    else
                    {
                        tcp_for[vpi/2] += 1;
                    }
                }
//p             fprintf(statfile, "Bingo TCP%d\n", vpi);
            }
            else {printf("BB1 : Wrong type of packet to forward (%s)\n", fmt);}
        }

        buff_size -= pklength;
//      printf("%f\t%d\n", op_sim_time(), buff_size);
        if (servicerate > 0)
        {
```

```
            if (op_subq_empty(0))
            {
                double servicetime;
                servicetime = 1.0 * pklength / servicerate;
                op_intrpt_schedule_self(op_sim_time() + servicetime, 0);
            }
            op_subq_pk_insert(0, pkptr, OPC_QPOS_TAIL);
        }
        else
        {
            op_pk_send(pkptr, 0);
        }
    }
    else
    {
        if(strcmp(fmt, "CED_UDP") == 0)
        {
//p         fprintf(statfile, ";;Merde UDP%d\n", vpi);
            udp_drop[vpi/2] += 1;
            op_pk_destroy(pkptr);
        }
        else
        {
            if(strcmp(fmt, "stcp_ippkt") == 0)
            {
                if(pklength <= 512)
                {
                    DROP512[vpi/2]++;//numstrpac512DROP++;
                    DROP512size += pklength;
                }
                else
                {
                    if((pklength > 512) && (pklength < 800))
                    {
                        DROP800[vpi/2]++;//numstrpac800DROP++;
                        DROP800size += pklength;
                    }
                    else
                    {
                        tcp_drop[vpi/2] += 1;
                    }
                }
//p             fprintf(statfile, ";;Merde TCP%d\n", vpi);
                op_stcp_discard_packet(pkptr);
            }
            else    {printf("BB1 : wrong type of packet to destroy (%s)\n", fmt);}
        }
    }
}
```

```
/* FIFO mode: service of packet */
void send_pk_now(void)
{
    Packet* pkptr;
    int pklength;
    char fmt[30];

    pkptr = op_subq_pk_remove(0, OPC_QPOS_HEAD);
    op_pk_format(pkptr, fmt);

    pklength = op_pk_bulk_size_get(pkptr);
    buff_size += pklength;

//  if((strcmp(fmt, "CED_UDP") == 0) || (strcmp(fmt, "stcp_ippkt") == 0)) totpacketserved++;
    totpacketserved++;

    op_pk_send(pkptr, 0);
    if (!op_subq_empty(0))
    {
        double servicetime;

        pkptr = op_subq_pk_access(0, OPC_QPOS_HEAD);
        pklength = op_pk_bulk_size_get(pkptr);
        servicetime = 1.0 * pklength / servicerate;
        op_intrpt_schedule_self(op_sim_time() + servicetime, 0);
    }
}

/* ************************************************************************** */
/* ************************************************************************** */
/*                                                                          */
/*                         RED QUEUING DISCIPLINE                           */
/*                                                                          */
/* ************************************************************************** */
/* ************************************************************************** */

void RedPacket(void)
{
    Packet* pkptr;
    int pklength, vpi;
    char fmt[30];

    pkptr = op_pk_get(op_intrpt_strm());
    pklength = op_pk_bulk_size_get(pkptr);

    op_pk_format(pkptr, fmt);
    op_pk_fd_get(pkptr, 0, &vpi);

//  if((strcmp(fmt, "CED_UDP") == 0) || (strcmp(fmt, "stcp_ippkt") == 0)) totpacketarrived++;
    totpacketarrived++;
```

```

    aveqsize = (1-weight)*aveqsize + weight*queuesize;
    queuesize += (double)pklength;

    if(queuesize > buff_size)
    {
    /* MANDATORY DROP */
        if(strcmp(fmt, "CED_UDP") == 0)
        {
//p             fprintf(statfile, ";;;;;;Merde UDP%d mandatory (buffer size)\n", vpi);
            op_pk_destroy(pkptr);
            udp_drop[vpi/2] += 1;
        }
        else
        {
            if(strcmp(fmt, "stcp_ippkt") == 0)
            {
                if(pklength <= 512)
                {
                    DROP512[vpi/2]++;//numstrpac512DROP++;
                    DROP512size += pklength;
                }
                else
                {
                    if((pklength > 512) && (pklength < 800))
                    {
                        DROP800[vpi/2]++;//numstrpac800DROP++;
                        DROP800size += pklength;
                    }
                    else
                    {
                        tcp_drop[vpi/2] += 1;
                    }
                }
//p             fprintf(statfile, ";;;;;;Merde TCP%d mandatory (buffer size)\n", vpi);
                op_stcp_discard_packet(pkptr);
            }
            else
            {
                printf("Bad type of format in bb1 (IN buffer size)\n");
            }
        }
        queuesize-= (double)pklength;
    }
    else
    {
        if(aveqsize <= min_th)
        {
        /* IN QUEUE */
            if(strcmp(fmt, "CED_UDP") == 0)
```

```
                {
//p                 fprintf(statfile, "Bingo UDP%d mandatory\n", vpi);
                    udp_for[vpi/2] += 1;
                }
                else
                {
                    if(strcmp(fmt, "stcp_ippkt") == 0)
                    {
                        if(pklength <= 512)
                        {
                            IN512[vpi/2]++;//numstrpac512IN++;
                            IN512size += pklength;
                        }
                        else
                        {
                            if((pklength > 512) && (pklength < 800))
                            {
                                IN800[vpi/2]++;//numstrpac800IN++;
                                IN800size += pklength;
                            }
                            else
                            {
                                tcp_for[vpi/2] += 1;
                            }
                        }
//p                     fprintf(statfile, "Bingo TCP%d mandatory\n", vpi);
                    }
                    else
                    {
                        printf("Bad type of format in bbl (IN mandatory)\n");
                    }
                }

                if (op_subq_empty(0))
                {
                    double servicetime;
                    servicetime = 1.0 * (double)pklength / servicerate;
                    op_intrpt_schedule_self(op_sim_time() + servicetime, 1);
                }
                op_subq_pk_insert(0, pkptr, OPC_QPOS_TAIL);
            }
            else
            {
                if((min_th < aveqsize) && (aveqsize < max_th))
                {
                    /* PROBAPILISTIC DROP */
                    double pkprob, dropb, dropa;
                    pkprob = op_dist_uniform(1.0);
                    dropb = (drop_max*((aveqsize - min_th)/(max_th - min_th)));
                    dropa = (dropb / (1.0 - (count * dropb)))*((double)pklength / 12000.0);
```

```
                if((dropa < 0) || (dropa > 1.0))
                {
                    dropa = 1.0;
                }
                if(pkprob >= dropa)
                {
                    /* IN QUEUE */
                    if(strcmp(fmt, "CED_UDP") == 0)
                    {
//p                     fprintf(statfile, ";Bingo UDP%d between\n", vpi);
                        udp_for[vpi/2] += 1;
                    }
                    else
                    {
                        if(strcmp(fmt, "stcp_ippkt") == 0)
                        {
                            if(pklength <= 512)
                            {
                                IN512[vpi/2]++;//numstrpac512IN++;
                                IN512size += pklength;
                            }
                            else
                            {
                                if((pklength > 512) && (pklength < 800))
                                {
                                    IN800[vpi/2]++;//numstrpac800IN++;
                                    IN800size += pklength;
                                }
                                else
                                {
                                    tcp_for[vpi/2] += 1;
                                }
                            }
//p                         fprintf(statfile, ";Bingo TCP%d between\n", vpi);
                        }
                        else
                        {
                            printf("Bad type of format in bbl (IN between)\n");
                        }
                    }

                    count = count + ((double)pklength / 12000.0);

                    if (op_subq_empty(0))
                    {
                        double servicetime;
                        servicetime = 1.0 * pklength / servicerate;
                        op_intrpt_schedule_self(op_sim_time() + servicetime, 1);
                    }
                    op_subq_pk_insert(0, pkptr, OPC_QPOS_TAIL);
```

```
                    }
                    else
                    {
                        /* DROP BECAUSE UPPER THAN MIN_TH AND PROBABILISTIC DROP */
                        if(strcmp(fmt, "CED_UDP") == 0)
                        {
//p                             fprintf(statfile, ";;Merde UDP%d between\n", vpi);
                            udp_drop[vpi/2] += 1;
                            op_pk_destroy(pkptr);
                        }
                        else
                        {
                            if(strcmp(fmt, "stcp_ippkt") == 0)
                            {
                                if(pklength <= 512)
                                {
                                    DROP512[vpi/2]++;//numstrpac512DROP++;
                                    DROP512size += pklength;
                                }
                                else
                                {
                                    if((pklength > 512) && (pklength < 800))
                                    {
                                        DROP800[vpi/2]++;//numstrpac800DROP++;
                                        DROP800size += pklength;
                                    }
                                    else
                                    {
                                        tcp_drop[vpi/2] += 1;
                                    }
                                }
//p                                 fprintf(statfile, ";;Merde TCP%d between\n", vpi);
                                op_stcp_discard_packet(pkptr);
                            }
                            else
                            {
                                printf("Bad type of format in bb1 (OUT between)\n");
                                op_pk_destroy(pkptr);
                            }
                        }
                        count = 0.0;
                        queuesize = queuesize - (double)pklength;
                    }
                }
                else
                {
                    /* MANDATORY DROP */
                    if(strcmp(fmt, "CED_UDP") == 0)
                    {
//p                         fprintf(statfile, ";;;;Merde UDP%d mandatory\n", vpi);
```

```
                    udp_drop[vpi/2] += 1;
                    op_pk_destroy(pkptr);
                }
                else
                {
                    if(strcmp(fmt, "stcp_ippkt") == 0)
                    {
                        if(pklength <= 512)
                        {
                            DROP512[vpi/2]++;//numstrpac512DROP++;
                            DROP512size += pklength;
                        }
                        else
                        {
                            if((pklength > 512) && (pklength < 800))
                            {
                                DROP800[vpi/2]++;//numstrpac800DROP++;
                                DROP800size += pklength;
                            }
                            else
                            {
                                tcp_drop[vpi/2] += 1;
                            }
                        }
//p                     fprintf(statfile, ";;;;Merde TCP%d mandatory\n", vpi);
                        op_stcp_discard_packet(pkptr);
                    }
                    else
                    {
                        printf("Bad type of format in bb1 (OUT mandatory)\n");
                        op_pk_destroy(pkptr);
                    }
                }
                queuesize = queuesize - (double)pklength;
            }
        }
    }
}

void RedSendPk(void)
{
    Packet* pkptr;
    int pklength;
    char fmt[30];

    pkptr = op_subq_pk_remove(0, OPC_QPOS_HEAD);
    pklength = op_pk_bulk_size_get(pkptr);
    op_pk_format(pkptr, fmt);

//  if((strcmp(fmt, "CED_UDP") == 0) || (strcmp(fmt, "stcp_ippkt") == 0)) totpacketserved++;
```

```
    totpacketserved++;

    queuesize -= (double)pklength;
    op_pk_send(pkptr, 0);

    if(!op_subq_empty(0))
    {
        double servicetime;

        pkptr = op_subq_pk_access(0, OPC_QPOS_HEAD);
        pklength = op_pk_bulk_size_get(pkptr);
        servicetime = 1.0 * pklength / servicerate;
        op_intrpt_schedule_self(op_sim_time() + servicetime, 1);
    }
//  else printf("buffer empty!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
}

void WriteStat(void)
{
    int i;

    if(queuedisc == 1)
    {
        fprintf(fred, "%f;;;%f\n", queuesize, aveqsize);
    }
    else
    {
        fprintf(ffifo, "%d\n", buff_size);
    }
}
```

```
Objid my_self, papa;
int time, l, i;
char file[30];
char fullname[100];

my_self = op_id_self();
papa = op_id_parent(my_self);

op_ima_obj_attr_get(my_self, "Service Rate", &servicerate);
op_ima_obj_attr_get(my_self, "Use buffer", &use_buffer);
if(use_buffer == 1) {printf("Use buffer = YES\n");}
else                {printf("Use buffer = NO\n");}
printf("Service Rate = %f\n", servicerate);
if(use_buffer)
{
    op_ima_obj_attr_get(my_self, "Buffer size", &buff_size);
}
else
{
    buff_size = 100000000;
}
printf("Buffer size = %d\n", buff_size);

/* ******************************************************
   **                                                  **
   **                    STATFILE                      **
   **                                                  **
   ****************************************************** */

file[0] = '\0';
fullname[0] = '\0';

strcat(fullname, "/home/users/rosmanc/simul/SIMULATION/");
op_ima_obj_attr_get(papa, "name", file);
strcat(fullname, file);
strcat(fullname, ".csv");
statfile = fopen(fullname, "w");
if(statfile == NULL)    {printf("Stat file of %s not opened!!!\n", fullname);}
else
{
    fprintf(statfile, ";;**************************\n");
    fprintf(statfile, ";;** SIMULATION PARAMETERS **\n");
    fprintf(statfile, ";;**************************\n\n");

    fprintf(statfile, ";Serv. rate;%f\n", servicerate);
    fprintf(statfile, ";Buffer size;%d\n\n", buff_size);
}

/* QUEUING DISCIPLINE */

op_ima_obj_attr_get(my_self, "FIFO OR RED", &queuedisc);

if(queuedisc == 1)
{
    char redname[100];

    printf("Queuing discipline = RED\n");
    op_ima_obj_attr_get(my_self, "min_th"   , &min_th   );
    min_th = (min_th*(double)buff_size)/100.0;
    op_ima_obj_attr_get(my_self, "max_th"   , &max_th   );
    max_th = (max_th*(double)buff_size)/100.0;
    op_ima_obj_attr_get(my_self, "drop_max" , &drop_max);
    drop_max = drop_max/100.0;
    op_ima_obj_attr_get(my_self, "Weight"   , &weight   );
    fprintf(statfile, ";queue discipline = RED\n");
    fprintf(statfile, ";;min_th;%f\n", min_th);
    fprintf(statfile, ";;max_th;%f\n", max_th);
```

```
        fprintf(statfile, ";;drop_max;%f\n", drop_max);
        fprintf(statfile, ";;weight;%f\n", weight);
        aveqsize  = 0.0;
        queuesize = 0.0;
        count     = 0.0;

        totpacketarrived = 0;
        totpacketserved = 0;


    /* ***********************
       **       REDFILE     **
       *********************** */

        redname[0] = '\0';
        strcat(redname, "/home/users/rosmanc/simul/SIMULATION/RED");
        strcat(redname, file);
        strcat(redname, ".csv");
        fred = fopen(redname, "w");
        if(fred == NULL)   {printf("Stat file of REDbackbone1.csv not opened!!!\n");}
        else
        {
            fprintf(fred, "RED RESULTS\n");
            fprintf(fred, "queuesize;;aveqsize\n");
        }
    }
    else
    {
        char fifoname[100];

        fprintf(statfile, ";queue discipline = FIFO\n");

    /* ****************************
       **       FIFOFILE         **
       **************************** */

        fifoname[0] = '\0';
        strcat(fifoname, "/home/users/rosmanc/simul/SIMULATION/FIFO");
        strcat(fifoname, file);
        strcat(fifoname, ".csv");
        ffifo = fopen(fifoname, "w");
        if(ffifo == NULL)   {printf("Stat file of FIFOabckbone1.csv not opened!!!\n");}
        else
        {
            fprintf(ffifo, "FIFO RESULTS\n");
            fprintf(ffifo, "buff_size;%d\n", buff_size);
        }
    }

    if(statfile != NULL)
    {
        fprintf(statfile, "\n");
        fprintf(statfile, ";;***********************\n");
        fprintf(statfile, ";;** SIMULATION RESULTS **\n");
        fprintf(statfile, ";;***********************\n\n");
    }

    op_ima_obj_attr_get(my_self, "Duration time", &time);

    for(l = 0; l < time; l++)
    {
        op_intrpt_schedule_self(l       , 3);
        op_intrpt_schedule_self(l + 0.25, 3);
        op_intrpt_schedule_self(l + 0.5 , 3);
        op_intrpt_schedule_self(l + 0.75, 3);
    }

```

```
for(i = 0; i < (NUMBER_UDP_SRC*2); i++)
{
    udp_drop[i] = 0;
    udp_for[i] = 0;
}
for(i = 0; i < (NUMBER_TCP_SRC*2); i++)
{
    tcp_drop[i] = 0;
    tcp_for[i] = 0;
    IN512[i] = 0;
    IN800[i] = 0;
    DROP512[i] = 0;
    DROP800[i] = 0;
    IN512size = 0;
    IN800size = 0;
    DROP512size = 0;
    DROP800size = 0;
}
```