Extending the Functionality of RTP/RTCP Implementation in Network Simulator (NS-2) to support TCP friendly congestion control

Christos Bouras
Research Academic Computer Technology Institute and University of Patras
N. Kazantzaki Str., University of Patras, 26500 Rion, Greece
+30 2610 960375
bouras@cti.gr

Apostolos Gkamas
Research Academic Computer Technology Institute and University of Patras
N. Kazantzaki Str., University of Patras, 26500 Rion, Greece
+30 2610 960465
gkamas@cti.gr

Georgios Kioumourtzis
University of Patras, Department of Computer Engineering and Informatics
University of Patras, 26500, Rion, Patras, Greece
+30 2610 960316
gkioumou@ceid.upatras.gr

ABSTRACT
In this paper, we present a modification of the ns2 code for the RTP/RTCP protocols. The legacy RTP/RTCP code in ns2 has not yet been validated but it provides a framework of the protocol's specification for experimental use. We have modified the code by adding all the RTP/RTCP protocol's attributes that are defined in RFC 3550 and related to QoS metrics. We have also implemented additional algorithms and functions in order to enhance our modified code with TCP friendly bandwidth share behavior. Our protocol, named RTPUP ("UP" stands for the University of Patras), is offered as a package and is fully documented so that it can be used for simulations and research within the ns2 simulation environment.

Categories and Subject Descriptors
I.6 [**Simulation and Modeling**]: Applications, Model Validation and Analysis, Model Development, Simulation Output Analysis, Miscellaneous.

General Terms
Algorithms, Performance, Design, Standardization, Verification.

Keywords
Network Simulator (NS-2), RTP/RTCP protocol, Multimedia transmission, TCP Friendly.

1. INTRODUCTION
Real time multimedia applications have enjoyed the global interest over the last years. These applications are characterized by three main properties: the demand for high data transmission rate (bandwidth-consuming applications), the sensitiveness to packet delays (latency and jitter), and last the tolerance to packet losses (packet-loss tolerant applications), when compared to other kind of applications. The Transmission Control Protocol (TCP) is the dominant and most widely used protocol at the transport layer. However, there are three characteristics of this protocol that makes it insufficient for real time data delivery:

- TCP has a built-in retransmission mechanism that may be useless for delay-sensitive applications.
- TCP does not carry any time related information, which are needed by real time applications, and lastly,
- TCP employs a "strict" congestion control mechanism that reacts even in the light of a single packet loss event.

Similarly, the User Datagram Protocol (UDP) does not provide any support for multimedia applications. Therefore, the need of a new protocol led the research community to design the Real Time Protocol (RTP) and the associated RTP Control Protocol (RTCP) [1], in order to support multimedia applications. The RTP protocol constitutes a new de facto standard and is the dominant transport protocol for multimedia data transmission.

The implementation of the RTP in NS2 [2] is very generic. It only provides the main functions of a "common" transport protocol and runs on top of UDP. In this work, we extend the functionality of the RTP and RTCP code in NS2 to include:

- The feedback functions that are described [1] and related to QoS metrics.
- TCP friendly behavior with the meaning that the transmitted flow consumes no more bandwidth than a TCP connection, which is traversing the same path with the transmitted flow.

With these new feedback functions any multimedia application can employ the internal mechanisms of the RTP and RTCP for Quality of Service (QoS) measurements. The TCP friendly bandwidth share mechanism is based on the TCP Friendly Rate Control (TFRC) protocol presented in [3]. Our motivation is to use the RTP modified code for simulations of multimedia data transmission from a server to a number of receivers, through multicasting and different multicast RTP streams. The ns2 code provides the framework for these simulation scenarios. However, one has to extend the code to support these scenarios because the RTP code in ns2 cannot support multiple RTP streams running in one network node. The rest of this paper is organized as follows: The next section briefly describes RTP and the RTCP protocols. Section 3 discusses the Algorithmic aspects. The extensions

SIMUTools 2008, March 3-7, 2008, Marseille, France.

made to RTP code in ns2 are presented in section 4, as section 5 presents the performance evaluation of our modified code. Conclusions and future work are discussed in section 6.

2. REAL TIME PROTOCOL (RTP) AND CONTROL RTP (RTCP)

In this section we briefly discuss the attributes of the Real Time Protocol and the associated Control protocol (RTCP)

2.1 Real Time Protocol (RTP)

RTP is a real time transport protocol that is usually being used on top of the UDP protocol (other transport protocols are also being supported by RTP). By saying this we already accept a transport protocol on top of other transport protocols and this statement may be misleading. On the other hand, RTP is highly coupled to the application that it carries. Therefore, RTP would be better viewed as a framework for real time applications and not only as a transport protocol. RTP neither provides any guarantees for data delivery nor packet delivery in order. The main functions of RTP include:

- Identification of payload type
- Identification of the source sending the RTP packets
- Timestamps to RTP packets
- Sequence numbers to RTP packets

2.2 RTP Control Protocol (RTCP)

The RTCP protocol provides to participants of the RTP session feedback information concerning the network conditions. RTP and RTCP protocols use different port numbers. The main functions of RTCP are:

- Network measurements for QoS (packet loss ratio, delay jitter, timestamps of sender and receiver reports etc.)
- Identification of the source sending the RTCP packets
- Estimation of the session size and scaling mechanisms

The RTCP sender (SR) and receiver (RR) reports provide direct information on the packet losses, cumulative number of RTP packets sent by the source and delay jitter. They provide also additional fields that can be used for implementation of congestion control policies by separate protocols, for example the TCP-like flow control, which we have implemented. A separate entity, like a network management, can obtain network metrics based on the reception of the RTCP reports without actually taking part in the RTP session.

Other information carried by the RTCP packets include a source identifier of the transport layer (CNAME), the e-mail address, the name, the phone and location of the source originated the RTCP report.

3. ALGORITHMIC ASPECTS

In this section we describe the algorithm to estimate a TCP friendly bandwidth share. Then we explain how we estimate the packet loss ratio and the Round trip Time (RTT) that are used for the TCP friendly bandwidth calculations. Finally, we present the inter-arrival jitter delay estimations, which are based on the RFC 3550 recommendations.

3.1 TCP Friendly Bandwidth Share Estimations

The subject of transmission of TCP friendly flows over networks has engaged researchers all over the world, [4], [5] and [6]. Various adaptation schemes deploy an analytical model of TCP [4] in order to estimate a TCP friendly bandwidth share. With the use of this model, the average bandwidth share ($r^{i}_{r\_tcp}$) of a TCP connection is determined (in bytes/sec) with the following equation:

$$r^{i}_{r\_tcp} = \frac{P}{t^{e-i}_{RTT}\sqrt{\frac{2l_i}{3}} + 4t^{e-i}_{RTT}\min(1,3\sqrt{\frac{3l_i}{8}})l_i(1+32l_i^{2})} \quad (1)$$

where $r^{i}_{r\_tcp}$ is the receiver's estimation of the TCP friendly bandwidth share, P is the packet size in bytes, l is the packet loss rate, t_{RTT} is the Round Trip Time (RTT) of the TCP connection.

If the receiver does not experience packet losses the $r^{i}_{r\_tcp}$ must not be increased more than a packet / RTT. For this reason the receiver calculates the value of $r^{i}_{r\_tcp}$ with the following equation (in bytes/sec):

$$r^{i}_{r\_tcp} = r^{i}_{r\_tcp} + \frac{1}{t^{e-i}_{RTT}}P \quad (2)$$

Each time the receiver sends a receiver report to the sender it includes the average value of $r^{i}_{r\_tcp}$ since the last receiver report.

3.2 Packet Loss Rate Estimation

Every receiver that joins the RTP session can measure the packet loss rate based on RTP packet sequence numbers. In order to prevent a single spurious packet loss having an excessive effect on the packet loss estimation, the receivers smooth the values of packet loss rate using the filter presented in [6], which computes the weighted average of the m most recent loss rate values. The authors of [6] have also evaluated this filter and the results are very positive.

3.3 RTT Estimations

When a receiver i receives a RTP packet from a sender, it uses the following algorithm to estimate the RTT between the sender and the receiver:

if no feedback has been received before

RTT = sqrt(effective_RTT)

else

*RTT = q * RTT + (1-q) * effective_RTT* (3)

where, *q* has a default value of 0.9

This calculation is based on the sender estimation of the RTT time (*effective_RTT*) and is measured by using the timestamps of the RTCP sender and receiver reports. The algorithm above is described in [3] with the difference that in the TFRC specification the sender makes the RTT estimations. In our implementation we have the receivers estimating the RTT time.

3.4 Inter-arrival Jitter Estimations

Our implementation for delay jitter calculations is based on the algorithm defined in RFC 3550. Shortly explaining, let S_i is the RTP timestamp of packet *i*, and R_i is the arriving time in RTP timestamp units of packet *i*, then for two sequentially packets *i* and *j*, delay *D* may be expressed as:

$$D = (R_j - R_i) - (S_j - S_i) \quad (4)$$

This delay variation should be calculated for each RTP packet. RFC 3550 suggests a filter function to avoid temporal fluctuation and the delay jitter is computed with the use of the following equation:

$$J_i = (15/16)J_{i-1} + (1/16)D \quad (5)$$

All the above-described algorithms are implemented in our RTP modified code.

4. EXTENSIONS TO RTP CODE

In this section we describe the extensions made to RTP code in ns2. Our work is mainly divided into two main areas:

- Providing the RTP code the additional functionality defined in RFC 3550 and related to QoS metrics.
- Employing TCP friendly bandwidth share mechanisms for experimental use.

The extensions made in the ns2-2.30 version, on a Linux platform running Fedora 6 operating system.

4.1 Software Architecture

We present the structure of the RTPUP code using the UML diagram in figure 1. First of all we have renamed the RTP packet header from "*hdr_rtp*" to "*hdr_rtpup*" ("up" stands for University of Patras) to distinguish our code from the legacy code in ns2 in order to avoid confusion within the ns2 users community. We also defined new data structures named *server_report* and *receiver_report* to store the fields of the RTCP SR and RR, respectively. A new class named *RTPUPReceiver* was declared to hold the fields that are used by the receiving Agents for QoS measurements. Every new instance of the *RTPUPSession* class creates two instances of the *RTPUPSource* and one instance of the *RTPUPReceiver* classes, accordingly. The *RTPUPSessionClass* is called by the TCL script and in turn two new Agents (*RTPUPAgent* and *RTCPUPAgent*) are assigned to every node in the network that participates in the multicast stream. The *RTPUPAgent* holds all the functionality for sending and receiving RTPUP packets, whereas the *RTCPUPAgent* is responsible for transmission and reception of the RTCPUP sender and receiver reports. We have implemented a one-to-many scheme of the RTP/RTCP protocol, in which one sender transmits a multicast stream to a set of receivers. It is however, easy and quite straightforward to extend the code so that a node can be a receiver and at the same time an active sender. This applies to VoIP applications in which the sender is also a receiver during the VoIP session. Last, new functions are also used for the implementation of the algorithms described in the previous section.

We will explain in more details the functionality of the RTPUP code in the following paragraphs with the following TCL usage example:

```
set s0 [new Session/RTPUP]
$ns at 0.1 "$s0 join-group $group"
$ns at 1.0 "$s0 transmit 256kb/s"
$s0 enable-control 1
```

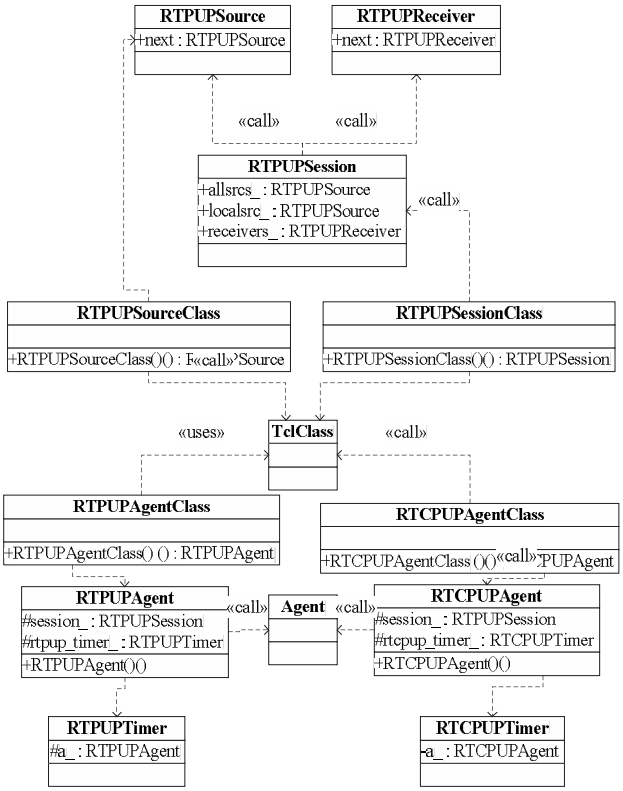

Figure 1. UML diagram of the RTPUP code

When a new session is created from the TCL programming language script in our simulation environment, a new instance of the *RTPUPSession* class is returned. The constructor of the *RTPUPSession* class initializes the *localsrc_* and *allsrcs_* instances of the *RTPUPSource* class and also the *receivers_*, which is an instance of the *RTPUPReceiver* class. The *localsrc_* stands for the originator of the RTPUP and RTCPUP packets. It is possible that the *localsrc_* generates only RTCP packets if it is only a receiving source in the newly created session. Next line of the TCL command calls the above-described functions:

```
set s0 [new Session/RTPUP]
```

The RTCP packets are originated by all the participants in the session. The creation of the RTP and RTCP packets is done by

calling the *RTPUPAgent* and *RTCPUPAgent* classes respectively. The instances of these classes are created in the initialization of the *RTPUPSession* TCL class. The two Agents are not active until the new session `s0` joins a multicast group. We can "manually" declare the instances of the *RTPUPAgent* and *RTCPUPAgent* classes but we recommend the use of the RTPUP code for multicast transmission. Next line calls session `s0` to join the multicast group at 0.1 second:

```
$ns at 0.1 "$s0 join-group $group"
```

where *group* is the multicast address. Until now there is no transmission of any RTPUP or RTCPUP packets because session `s0` was simply declared and joined a new multicast group. We need to call the *start* function for session `s0` to start transmitting RTCP packets at 0.1 second:

```
$ns at 0.1 "$s0 start"
```

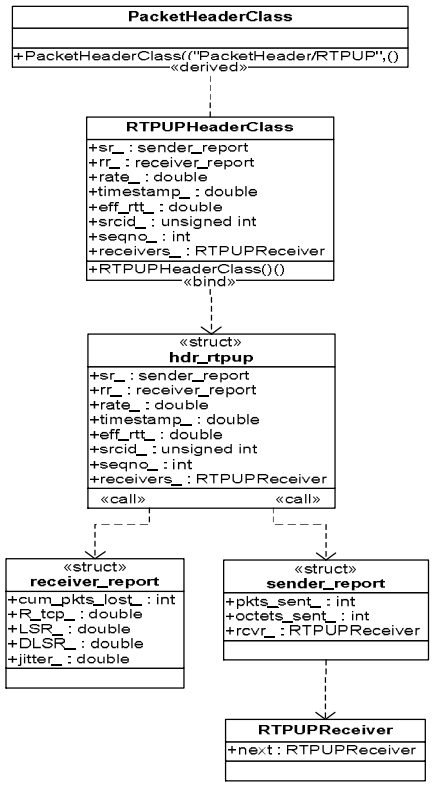

Figure 2. Class dependency of RTPUP and RTCPUP packet types

The most important field of the RTCP packets in this initial phase is the *srcid_*, which is the session unique identification. This field is an unsigned integer that is unique amongst all participants in the multicast group.

However, the transmission of "real" RTP data packets cannot start until the *transmit* function is called:

```
$ns at 1.0 "$s0 transmit 256kb/s"
```

The above function provides the "green light" to the *RTPUPAgent* and the transmission of RTPUP packets starts at the rate that we have decided in the above command.

In our implementation we defined a new function in the *Session/RTPUP* TCL class to enable the TFRC friendly congestion control. We did it in such manner so the user can choose from his TCL script to enable or not this congestion control. Therefore, to enable the congestion control the user should execute the next command in the TCL script:

```
$s0 enable-control 1
```

The default value is zero, which means that the congestion control is disabled by default. Figure 2 shows the class dependency for the creation of the RTPUP and RTCPUP (SR and RR) packets. We can see in the UML diagram the new fields that we added to provide QoS measurements and also the necessary information for the congestion control mechanism. New *inline* functions provide the accessibility to these fields. In the next subsections we will discuss and explain how the data collection and processing is done and how the TFRC congestion control is implemented.

4.2 Modified and New Functions

In our RTPUP code we distinguish three major functions/modules.

4.2.1 Send and Receive RTPUP Packets

RTPUP packets are generated based on a `timeout` event of the *RTPUPTimer*. The RTPUP Agent creates a new RTPUP packet by calling the send function:

```
void RTPUPAgent::sendpkt(){}
```

The send function invokes the make packet function, which creates the new RTPUP packet and adds the following fields in the packet header:

```
void RTPUPAgent::makepkt(Packet* p){}
```

- the sequence number of the RTP packet.

  ```
  rh->seqno() = seqno_++;
  ```

- the source id of the sending source

  ```
  rh->srcid() = session_->srcid();
  ```

- the timestamp

  ```
  rh->timestamp()= timestamp_;
  ```

- the receivers which this sender serves with the receiver source id field and the effective RTT

  ```
  rh->receivers_= session_->receivers_;
  ```

in which the effective RTT is defined by:

$$eff\_rtt = A - t_{LSR} - t_{DLSR} \quad (6)$$

where, t_{LSR} is the time during which the receiver received the last SR, t_{DLSR} is the time elapsed between the reception of the SR last report and the generation of a new RR report, and A stands for the current time of the reception of the RR. We will see later how the calculation of the effective RTT is done by the sender.

When the receiver receives the RTPUP packet it first calls a lookup function to check if the originator of the packet is a known source. If not, a new source is added by calling the *new-source* function of the *Session/RTPUP* TCL class. The processing of this newly receiving packet follows. We added a conditional statement to make sure that the receiving source is not identical with the sending source:

```
if(rh->srcid()!=localsrc_->srcid())
```

In the ns2 legacy source code the sending source is the first source that receives the packet, which it has just sent. In the lack of any documentation for the ns2 legacy code we regarded it as a flow that would affect our measurements. Therefore, when the condition is met, the receiving source extracts the effective RTT that the sender has assigned for this receiver by executing the next code segment:

```
for (RTPUPReceiver* p = rh->receivers_;
p != 0; p = p->next) {
  if(p->srcid() == localsrc_->srcid()) {
     eff_rtt = p->eff_rtt();
    }
}
if(eff_rtt != 0 ) {
 calculate_RTT(eff_rtt);                    (7)
}
```

The above lines are straightforward. If the sender and the receiver have not exchanged yet any SR or RR reports we assume symmetric links to avoid division by zero values. Thus, we set the RTT to double the value of the one-way trip. When the receiver gets non-zero effective RTT values, (which happens within the first seconds after the session establishment), it calls function (7) to calculate the estimated RTT time.
Next the receiver calculates the delay jitter. We use the code that is presented in RFC 3550 Appendix A.8.

```
double transit=arrival -rh->timestamp();
double d = transit - s->transit();
s->transit(transit);
 if (d < 0) d = -d;
s->jitter( s->jitter() + (1./16.) *
(d - s->jitter()));
```

where *transit* is the transit time of the received RTPUP packet, *d* is the difference in time units between two consequent RTPUP packets and *s->jitter()* holds the previous jitter delay measurement.

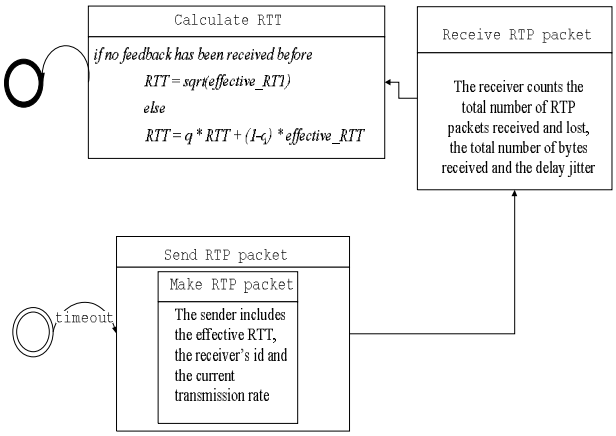

Figure 3. State chart of the send and receive functions

Finally, the receiver of the RTPUP packet assigns the following fields to `RTPUPSource s`:

```
//count received RTPUP packets
s->np(1);
// count lost RTPUP packets
s->cum_pkts_lost(pkts_lost);
//get the extended highest number
s->ehsr(rh->seqno());
// count the number of received bytes
s->nbytes(mh->size());
// the packet size in bytes
s->ps(mh->size());
```

Each *RTPUPSession* instance keeps in the *allsrcs_* field only the active sources in the session. Therefore, the receiving source is able to look up this field in order to locate the sending source identification number. To do so the receiving source invokes the `lookup` function that returns the *RTPUPSource* object `s`, which is the sending source. We can use the instance `s` to hold all the above values that we desire and add any other fields that we can use at a later time. Figure 3 depicts the state chart of the above-described functions.

4.2.2 Build RTCPUP Sender and Receiver Report Function

The build function is called by the *RTCPUPAgent* as a result of an *RTCPUPTimer* time-out event. The sender generates a new *SR* if it has sent *RTPUP* packets since the previous *SR*. When this condition is met the sender sets the `we_sent` flag to 1 and generates a sender report (SR). Next lines present the declaration and construction of the SR:

```
//add sender report
sender_report* sr;
//fill in the report
sr = new sender_report;
//assign the sender's id
sr->sender_srcid()= localsrc_->srcid();
//assign the RTPUP packets sent
sr->pkts_sent() = localsrc_->np();
//assign the total bytes sent
sr->octets_sent() = localsrc_->nbytes();
//include the receivers served
```

```
sr->rcvr_ = receivers_;
//store the report
rh_->sr_ = sr;
```

The sender includes the total number of RTPUP packets and the total number of bytes that has sent since the beginning of the session. It also includes the receivers that this source serves. We will explain in the next subsection how this instance of the *RTPUPReceiver* class is used by the receiving sources.

Alternately, each receiver before building the RR it has to calculate the TCP friendly bandwidth share. To do so, the receiver calculates the fraction of packets lost since the previous RR. We derive the algorithm for calculating the loss fraction from RFC 3550 specification. The loss fraction is defined as the fraction of the RTPUP packets lost over the number of RTPUP packets expected in the time interval between two successive RRs and has values between (0.0, 1.0). Next code segment calculates the loss fraction:

```
//calculate loss fraction since previous
report
int expected_interval  = sp->ehsr() -
last_ehsr_;
last_ehsr_ = sp->ehsr();
int lost_interval = expected_interval -
received;
if (lost_interval <= 0|| expected_interval
== 0 ) {
fraction = 0;}
else fraction = ((double)lost_interval /
(double)expected_interval);
```

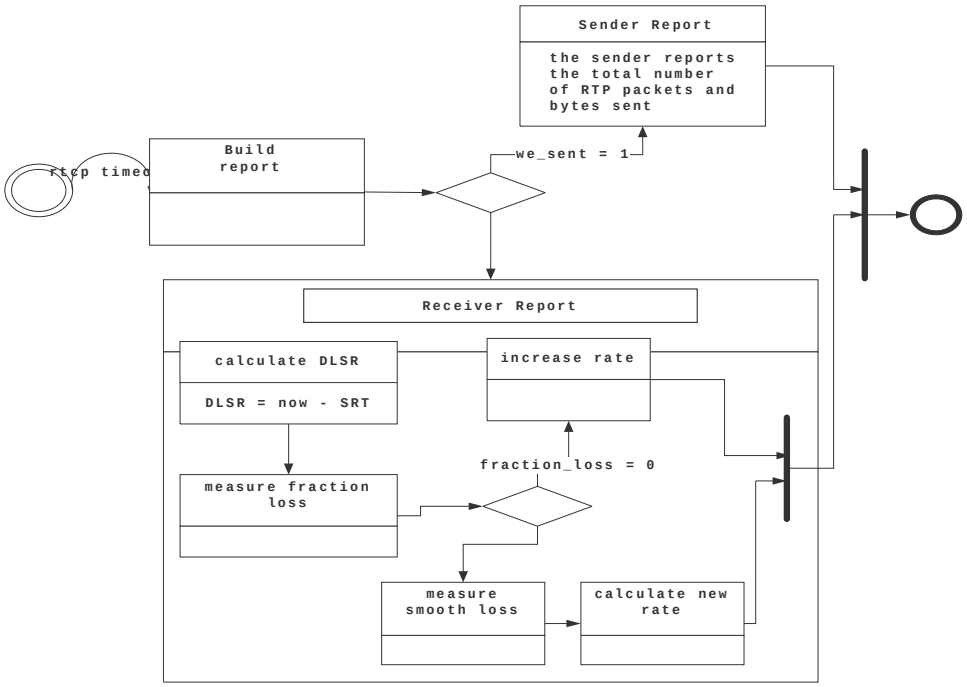

Figure 4. State chart of the build report function

```
// Update the R_tcp according to fraction
value
if( sp->np() >= 1) {// I have already
received RTPUP packets from the source
if(fraction == 0) {
increase_rate(sp->ps());                    (8)
}
else {
      measure_smooth_loss(fraction);
      calculateR_tcp(sp->ps());
      }
}
```

If the fraction loss is zero the receiver estimates a new transmission rate by executing (8) as shown below:

```
tx_rate_+=(double)ps/RTT_;
```

where, `ps` is the packet size of the RTPUP packet, *RTT_* is the estimated round trip time by the receiver and *tx_rate_* is the previous estimated TCP friendly transmission rate.

However, zero fraction loss is not always the case and the receiver calculates a smooth loss value by invoking the *measure_smooth_loss(fraction)* function:

```
double smooth_values = 0;
for (int i=0; i<7; i++) {
pkt_loss_history[i+1]= pkt_loss_history[i];
}
pkt_loss_history[0] = fraction;

double temp =0;
   for(int i=0; i<8; i++) {
   temp += weight[i];
   }

   for (int i =0; i<8; i++) {
```

```
   smooth_values += weight[i] *
pkt_loss_history[i];
   }

   smooth_loss_= smooth_values / temp;
```

The receiver has now the smooth loss ratio which is a consolidated value based on the previous seven measurements. The *pkt_loss_history* array holds these previous measurements. The `weight` array holds static values. The interesting reader can reference [7] for more details on these values. Next step for the receiver is to estimate the TCP friendly transmission rate by calling the following function that is the implementation of the TCP analytical model (1):

```
calculateR_tcp(sp->ps());
```

where *ps()* is an *inline* function that returns the size of the RTPUP packet. The receiver needs to know the packet size to perform the TCP calculation. We have seen previously when describing the receive RTPUP function how we obtained the RTPUP packet size and how we assigned this packet size to the sending source. We declared this field in the *RTPUPSource* class as different sources may use different packet size and wanted to have this direct accessibility.

After the computation of the various fields the receiver constructs the RR in the following lines:

```
//add receiver report
receiver_report* rr;
rr = new receiver_report;
//fill the report
// cumulative packets lost
rr->cum_pkts_lost()=sp->cum_pkts_lost();
// add TCP friendly rate
rr->R_tcp() = tx_rate_;
// last time sender report
rr->LSR() = sp->LSR();
// delay since receiving the SR
rr->DLSR()= now - sp->SRT();
//add jitter delay
rr->jitter() = sp->jitter();
//add RR to the RTCPUP packet
rh_->rr_ = rr;
```

4.2.3 Receive Control RTCPUP Packet Function

We have seen so far how the receivers access the RTPUP packets and how both sender and receivers build the SR and RR reports. We have also explained how the receivers perform the various calculations in order to provide the sender with QoS measurements. In this subsection we will describe what the actions are from the sender side in order to adjust its transmission rate. Therefore, the receive control function is the "merging" function in which the results of the program are presented and actions take place.

Upon the reception of a new RTCPUP report sender and receivers perform different functions. The sender firs evaluates if the originator of this RR does exist in its receiver's list. At this point it has to be mentioned that in the legacy ns2 code the *allsrcs_* field for the sending source is empty as long as it does not received any RTPUP packets from any source. That was the reason that led us to define the *RTUPReceiver* class, so that the sending source could be able to keep a list of all the receivers in the session it serves. Therefore, if the condition is not met (the originator of the RR has not "heard" by the sender) the sender adds the originator to his receivers' list. We use a similar function to the ns2 legacy code for constructing the receiver's list:

```
enter_rcv(RTPUPReceiver* s)
```

The sender processes the RR and calculates the effective RTT time as follows:

```
eff_rtt = alpha - rh->rr_->LSR() –

rh->rr_->DLSR();
```

where `alpha` is the current clock time

The TCP receiver's estimation is kept in a separate data structure. We use for it an instance of the class *list* in which its size is dynamically updated with the number of its elements so that we can hold a fair large amount of receivers. In addition, the class *list* offers a number of built-in functions that are very convenient for accessing and sorting its elements. Every time the sender receives a new report from the RTCPUP Agents in the multicast session it adjusts its transmission rate. The sender takes into account the minimum bandwidth estimations from the receiver set according to the algorithm below:

$$new\_rate = \min(r^1_{r\_tcp}, ..., r^i_{r\_tcp}) \quad (9)$$

where, $r^i_{r\_tcp}$ is the bandwidth estimation of receiver r^i_r. At this point and in order to prevent oscillations we use a *noFeedbackTimer* to check whether or not the sender has received feedback reports from all the receivers within a feedback interval. This feedback interval is defined as:

```
feedback_interval = 2 * ps/tx_rate_
```

where *ps* is the packet size of the RTP packet and *tx_rate_* is the current transmission rate. When the sender does not receive an expected RTCPUP report from a receiver within the feedback interval it cuts its sending rate to half. This is a congestion avoidance mechanism because a lost RTCPUP receiver report indicates a congested path. It has been noticed in our experimental simulations that this mechanism increases the overall performance of the protocol.

Next the sender updates its transmission rate by calling the `transmit` function in the *Session/RTPUP* TCL class.

In the receiver side when the receiver receives an RTCPUP SR it stamps the *SRT* field with the current clock time. The *SRT* stands for the Sender Report Time and this time will be used for the calculation of the *DLSR* :

```
source->LSR(rh->timestamp());

source->SRT(now);
```

With the above described procedures and functions we conclude the main modules of our modification to the ns2 RTP legacy code.

Figure 5 shows the state chart of the receive control function

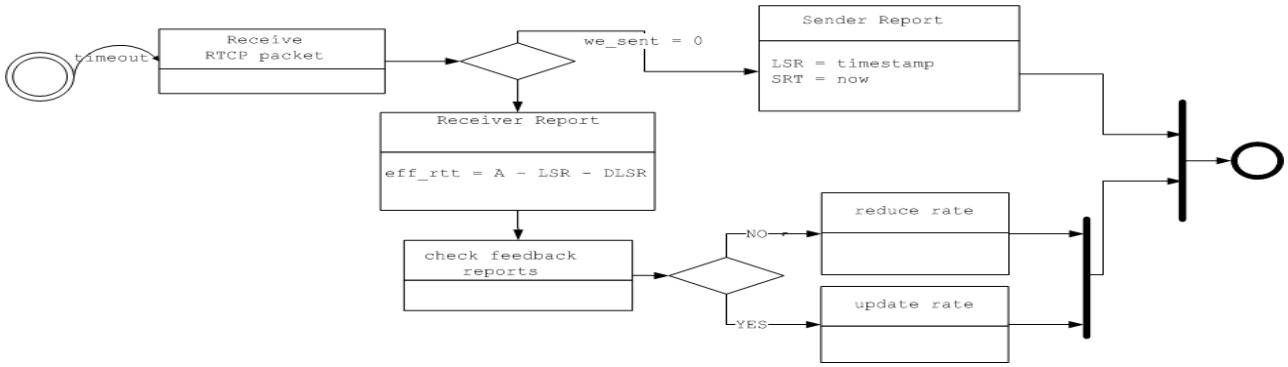

Figure 5. State chart of the receive control function

5. PERFORMANCE EVALUATION

We evaluate our model with simulations performed with the ns2 simulation software. Our main objective is first to verify that the RTPUP works properly and second that it has indeed friendly TCP behavior.

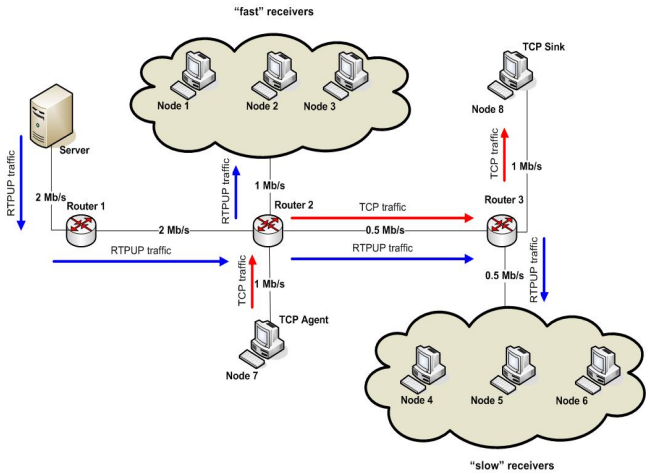

Figure 6. Simulated network topology

5.1 Simulation Environment and Network Topology Setup

Our benchmark for the evaluation of the RTPUP protocol is a Local Area Network (LAN), which consists of one multimedia server and six heterogeneous receivers. The heterogeneity of the receivers lays in the variation of the link capacity, which connects the receivers with the LAN. We have intentionally created a "bottleneck" between routers 2 and 3 to create two different sets of wired receivers. The first set of receivers (Nodes 1, 2, 3 "fast receivers") is able to receive at higher bit rates than the second set (Nodes 4, 5, 6 "slow receivers"). We run a simple simulation scenario in which the multimedia server transmits RTPUP traffic at an initial rate of 256Kb/s. The RTCP transmission interval is set to 500 msec. At the same time a File Transfer Protocol application (FTP) is transmitting TCP packets through the same pipe with the RTPUP traffic from Node 7 (TCP Agent) to Node 8 (TCP Sink). We run two different simulation sets to investigate:

- The behavior of our proposed protocol towards the TCP traffic
- The behavior of the TFRC implementation in ns2 towards the same TCP traffic
- Pros and cons between our implementation and the TFRC code in ns2.

Figure 6 depicts the network topology for the simulated scenarios.

5.2 First Simulation: Transmission of RTPUP Multicast Stream with Background TCP Traffic

We initially set the bandwidth capacity of the RTPUP traffic to 256Kb/s and the RTCPUP reporting interval to 500 msec. However, these two parameters do not remain unchangeable and adopt their values according to network conditions. As for the TCP protocol we use the standard TCP Reno version in ns2.

The transmission of both RTPUP and TCP traffic starts in the beginning of the simulation. We run our simulation for 200 seconds. In the chart presenting the simulation results (Figure 7) we can see the receiving rates from two representing nodes of the two different groups (Node 1,"fast receiver" and Node 4, "slow receiver") and also the TCP receiving rate named as "TCP Sink". We extract the following conclusions of the simulation results:

- The RTPUP protocol presents the characteristics of the TCP congestion control mechanism, in which the protocol increases its sending rate as long as the end-to-end path is congestion-free. This is a direct result of the TCP friendly algorithm that has been implemented in our code.
- The RTPUP protocol has also the characteristics of a multiplicative-decrease protocol, similar to TCP protocol. This is also the direct result of the implementation of the TCP analytical model in our code. However we have not implemented all the characteristics of the TCP protocol as our intention is to enrich the RTPUP with TCP friendly behavior and not to replicate the legacy TCP code.
- Another important conclusion is that our modified RTPUP protocol does have TCP friendly behavior. The TCP traffic is being delivered from the source to the destination node, although the path is heavily congested by the RTPUP traffic.

- RTPUP presents the same oscillations with the TCP protocol due to the implementation of the congestion control mechanism. We observe that when the TCP transmission rate increases, the RTUP transmission rate decreases and vice versa.

- One last important observation is that RTPUP has similar delivery ratio to both "slow" and "fast" receivers. This is a desired attribute of the protocol as it ensures a fair delivery ratio, which most times is above 100 Kb/s. We will see in the next simulation test whether or not the TFRC implementation in ns2 is able to keep an equal delivery ratio to the whole set of receivers.

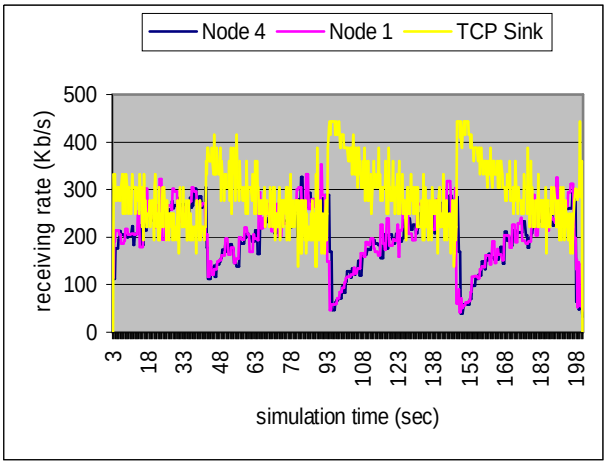

Figure 7. RTPUP versus TCP traffic

5.3 Delay Jitter Measurement

The delay jitter measurement is straightforward and is being done with a procedure that is defined in the TCL *Session/RTPUP* class. The results below (Figure 8) present the measurements of the "slow" receiver (Node 4) in contrast to the delay jitter of the "fast" receiver (Node 1). These results were measured during the previous simulation and are presented in the same chart, although the delay jitter values are in different scales. We present the results from Node 4 on the left *Y*-axis and the results from Node 1 on the right *Y*-axis. All the results are represented in seconds. We extract the following observations:

- Fast receivers enjoy minimum values of delay jitter; the highest observed delay jitter value throughout the simulation time is 2 msecs. We regard this as a good performance metric for the RTPUP protocol as the simulation scenario was set up in such way to challenge the protocol's performance.

- Slow receivers present delay jitter values between 10 and 15 msec and in general one-way jitter up to 150 msec is considered to be acceptable even for VoIP applications.

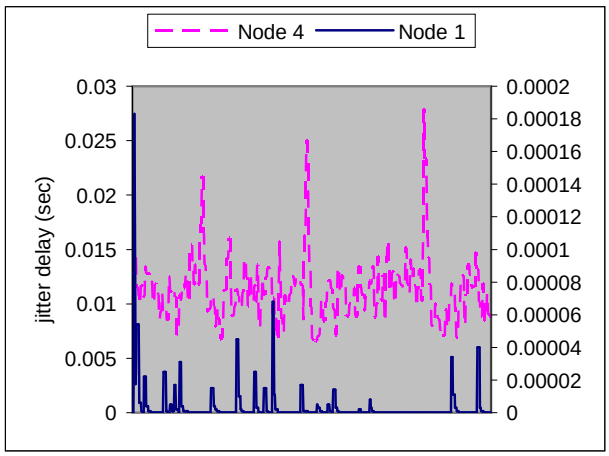

Figure 8. Delay jitter measurements

5.4 Packet Loss Rate Measurement

For the packet loss rate measurement we have also defined a new procedure in the TCL *Session/RTPUP* class. In this way we can get directly this metric from our simulation script. We measure the loss rate as the ratio of packets lost over the packets received during the sample interval. This sample interval is the time elapsed between two consequent Receiver Reports, (RR).

$$ploss\_rate = plost / prcv * 100 \qquad (10)$$

We can observe from the simulation results (figure 9) that lost events occur mainly when the network is heavily congested and this happens only for a very short period. We present only the packet loss ratio from a "slow" receiver (Node 4) as we have not observed any packet losses from fast receivers. This is a desired attribute of our RTPUP implementation as we have a multicast protocol that is able to transmit at high bit rates in a congested network, with low delay jitter and minimal packet losses. In the next simulation we will see how our implementation outperforms the TFRC implementation in ns2.

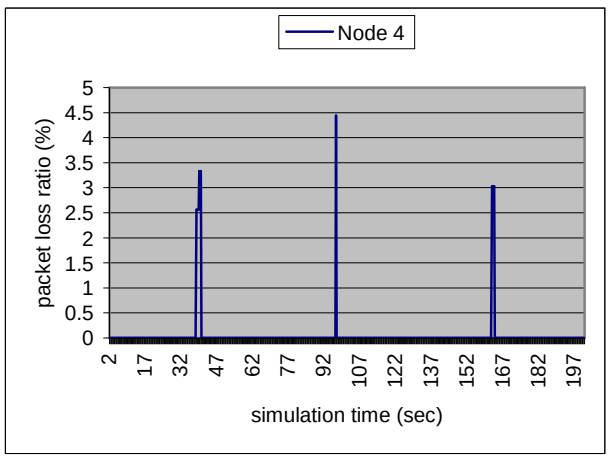

Figure 9. Loss rate measurements

5.5 Second Simulation: Comparison with the TFRC Implementation in ns2

In the last simulation we compare the TFRC implementation in ns2 against our RTP/RTCP with the TCP friendly enhancements. The TFRC code in *ns2* has been used for simulation by a number of researchers and provides an acceptable implementation of the TFRC specification.

The simulation scenario has exact the same network attributes with our previous simulation in order to achieve a fair comparison. In this case, RTP traffic is transmitted to the same set of receivers and the congestion control is left to TFRC protocol. We transmit also the same TCP traffic across the network from Node 7 to Node 8. Figure 10 depicts the simulation results.

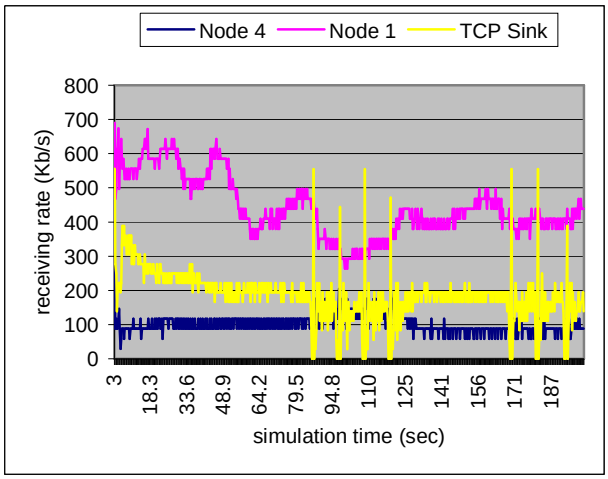

Figure 10. TFRC in ns2 versus TCP traffic

- We observe from the above results that the *ns2* TFRC implementation has "smoother" oscillations than our implementation, which is a desired attribute especially for video transmission. The TCP friendly behavior is also stable except for some cases, in which TCP traffic is reduced to zero. In our implementation the TCP traffic has always equal or higher values when compared to the initial transmission rate.
- A second observation is that although Node 1 ("fast receiver") enjoys high receiving rates, Node 4 ("slow receiver") has very low receiving rates. However, it has to be mentioned that the TFRC code in ns2 is used for unicast transmission. Thus, the sender transmits different unicast streams to each one of the receivers and adjusts its transmission rate accordingly.
- Our final conclusion is that our RTP/RTCP implementation introduces very good characteristics when we have multicast video stream that is transmitted via a congested path. The code and the implementation complexity of our implementation are very low when compared to the TFRC module in *ns2*.

6. CONCLUSIONS/FUTURE WORK

We present in this work an extension of the RTP code in ns2. Our motivation was to enrich the functionality of the existing code by including all the RTP/RTCP protocol's specification in RFC 3550, which are related to QoS metrics. We also extended our code to enhance it with TFRC mechanisms for research and experimental use. Our effort was to keep the functions and the data fields of the original ns2 code, to modify existing functions and to define only the necessary functions for the implementation of the new algorithms. We tried also to keep the "code style" of the ns2, document our code and offer it as a package for easier integration into ns2 libraries. There were several simulation runs and tests along with those that are presented in this work in order to verify that we get the correct QoS measurements. Simulation results show that the RTPUP performance has certain advantages for multicast transmission of delay sensitive data, such as VoIP and video streaming. In our future work we will extend the RTUP code to support simultaneous RTPUP multicast streams in one node for experimental use. Finally, simulation examples, sources and documentation are available in the following URL: http://ru6.cti.gr/ru6/ns_rtp_home.php

7. ACKNOWLEDGEMENT

We thank the anonymous SIMUTools 2008 reviewers for their helpful comments.

8. REFERENCES

[1] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, "*A Transport Protocol for Real-Time Applications*" RFC 3550, July 2003

[2] http://www.isi.edu/nsnam/ns/

[3] Handley, M.; Floyd, S.; Padhye, J.; Widmer, J. "*TCP Friendly Rate Control (TFRC): Protocol Specification" Request for Comments (RFC) 3448*, The Internet Society, January 2003

[4] J. Pandhye, J. Kurose, D. Towsley, R. Koodli, *"A model based TCP-friendly rate control protocol"*, Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), Basking Ridge, NJ, June 1999.

[5] D. Sisalem, A. Wolisz, *"MLDA: A TCP-friendly congestion control framework for heterogeneous multicast environments"*, in Eighth International Workshop on Quality of Service (IWQoS 2000), Pittsburgh, PA, June 2000.

[6] L. Vicisiano, L. Rizzo, J. Crowcroft, *"TCP - like congestion control for layered multicast data transfer"*, in IEEE INFOCOM, March 1998, pp. 996 - 1003.

[7] C. Bouras, A. Gkamas, G. Kioumourtzis, *"A Framework for Cross Layer Adaptation for Multimedia Transmission over Wired and Wireless Networks"*, The 2007 International Conference on Internet Computing (ICOMP'07), Las Vegas, Nevada, USA, 25 - 28 June 2007.