A New Network Simulator 2 (NS-2) Module Based on RTP/RTCP Protocols to Achieve Multimedia Group Synchronization

Mario Montagud, Fernando Boronat
Universidad Politécnica de Valencia - IGIC Institute
46730, Grao de Gandia, Valencia, Spain
+34+96 284 93 41

mamontor@posgrado.upv.es; fboronat@dcom.upv.es

ABSTRACT
Network simulation represents a broadly methodology for communication network performance analysis. As a system modelling approach, simulation allows to model arbitrary scenarios that many times are very difficult to implement in real platforms, or when it is intended to evaluate some alternative solutions without necessity of implementing all of them. In this paper, we present a modification of the NS-2 code for the RTP/RTCP standard protocols by adding the attributes specified in RFC 3550 that the native code doesn't include or doesn't follow strictly. Also, we have extended this code to include a multimedia group synchronization approach based on these protocols. This approach was already implemented and evaluated in a real WAN scenario with satisfactory results, but we needed to validate its performance in other more complex heterogeneous scenarios using simulation techniques. The simulation results have proved this approach as a suitable solution for multimedia applications which require group synchronization.

Categories and Subject Descriptors
C.2.2. [**Computer-Communication Networks**]: Network Protocols; I.6. [**Computing Methodologies**]: Simulation and Modeling;

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

Keywords
Group Synchronization, Multimedia Systems, RTP/RTCP, Simulation, NS-2.

1. INTRODUCTION
Network Simulator 2 (NS-2), [1], is an open-source event-driven simulation tool, developed at UC Berkeley, that has become one of the most widely employed simulator tool for industry, teaching and researching as a way of designing, testing and evaluating new and existing protocols and technologies. Since its inception, NS-2 has been under constant improvement and nowadays it supports heterogeneous network architectures characterization, such as Mobile IP networks, WLAN, ad-hoc networks, grid architectures, satellite networks, sensor networks, and many others. Additionally, it contains modules for numerous network components such as MAC layer protocols, unicast and multicast routing algorithms, transport layer protocols (TCP, UDP, SRM, RTP, RTCP, …), traffic source behaviour (FTP, Telnet, HTTP, CBR, …), queue management mechanisms (Drop Tail, RED, WFQ, …), statistics measurements (throughput, delay, jitter, queue monitoring, drops at links and queues, …), etc.

Although this variety, sometimes we need to adapt the existing NS-2 modules to our requirements or incorporate new simulation modules which are beyond the scope of the built-in NS-2 code. The simulator is open source; hence, it allows everyone to make changes to the existing code besides to add new protocols and functionalities to it. This makes it very popular among the networking community which can easily evaluate the functionality of their new proposed and novel designs for network research.

We are interested in the RTP (Real Time Protocol) and RTCP (Real Time Control Protocol) implementation in NS-2. These protocols are defined in RFC 3550 [2]. Nowadays, more and more applications use these protocols for multimedia streaming (video, audio, graphics, etc.). While RTP cares about data delivery, RTCP deals with the transport and management of feedback reports (control messages) from all the participants of an RTP session.

Previously, we developed an algorithm to synchronize a group of receivers distributed in an IP network, using these protocols, known as "*RTP-based Feedback Global Synchronization Approach (RFGSA)",* described in [3], and based on *Feedback Protocol* [4] and *Feedback Global Protocol* [5]. The approach was implemented, by modifying existing open source RTP-based tools, such as *vic* (for video stream) and *rat* (for audio stream), and tested, both objectively and subjectively, using LAN and WAN environments. The satisfactory results validated this proposal as a suitable solution for multimedia applications which require group synchronization. Now, we are interested in validating its performance in other more complex heterogeneous scenarios using NS-2 simulator.

When we started to work with NS-2, we discovered that the native implementation of RTP and RTCP protocols in NS-2 is quite generic. Many attributes specified in RFC 3550 are not included

SIMUTools 2010 March 15–19, Torremolinos, Malaga, Spain.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8686
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8686

or don't meet the RFC requirements (specially the sizes of the variables representing the packet header fields).

For the above reasons, in this work, we decided to develop a new module for NS-2 providing a more complete RTP and RTCP implementation (following strictly the RFC specification), and to include an optional functionality with our group synchronization approach, taking advantage of the ability to extend and create new RTCP messages.

In this work, we use the multimedia synchronization concept to refer to the process of integration at the presentation instant (or playout point) of different types of media streams. There are three well-known kinds of multimedia synchronization: Intra-Stream Synchronization, Inter-Stream Synchronization and Group or Inter-Destination Synchronization. Figure 1 shows an example of all of them, in which we can see a group of distributed receivers over an IP network, playing video, data and audio streams. First, we can see that they begin the playout of the different streams at the same time (we call it *Initial Playout Instant*) and, at any moment, the three receivers are playing the same Media Data Units (MDU) of each stream (Group or Inter-Destination Synchronization). In addition, we can observe how the temporal relationships between the different streams are maintained at any time, as a sign of inter-stream synchronization (e.g. lip synchronization). Moreover, we can notice the proper and continuous playout of each media stream (intra-stream synchronization).

The rest of the paper is organized as follows. In the next section we discuss the basics of RTP and RTCP standard protocols, its native implementation in NS-2 and related works. In Section 3, our new NS-2 module for RTP and RTCP protocols including our group synchronization approach is presented. Next, Section 5 presents the evaluation of this approach, implemented in a typical scenario. Finally, we present our conclusions, summarize our contributions and suggest some ideas for future work in Section 6.

2. NATIVE IMPLEMENTATION OF RTP/RTCP IN NS-2 AND RELATED WORK

2.1 Overview of RTP and RTCP

RTP provides end-to-end delivery services for data with real-time or near real-time characteristics, such as audio and video data. RTP itself does not provide any mechanism to ensure timely delivery or provide other quality-of-service guarantees, but relies on lower-layer services to do it. Sequence numbers are included in RTP packets' headers to allow the receiver to reconstruct the sender's packet sequence. Moreover, sequence numbers might also be used to determine loses and proper locations of packets, for example in video decoding, without necessarily decoding packets in sequence.

RTCP is the companion control protocol for RTP. Media senders (sources) and receivers (sinks) periodically send RTCP feedback reports that are important for monitoring and maintaining of the quality of RTP packets delivery. Each compound RTCP packet ([2]) may contain various sub-packets, usually a Sender Report (SR) or Receiver Report (RR) followed by a Source Description (SDES). If a user leaves an RTP session, a BYE RTCP message is sent. Finally, Application messages (APP) can be used to add application-specific information to RTCP packets.

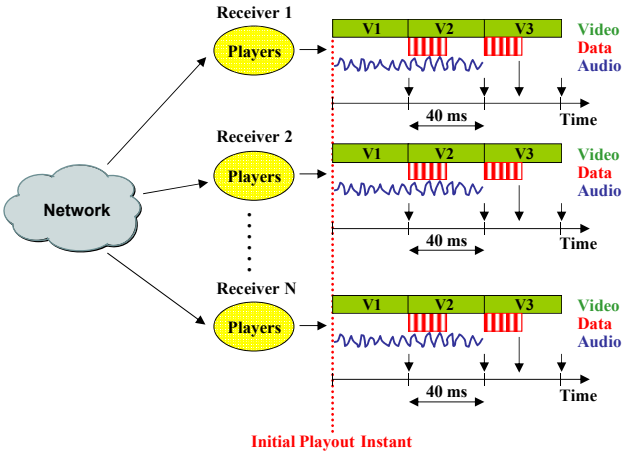

Figure 1. Multimedia Synchronization Types

Both, RTP and RTCP typically run over User Datagram Protocol (UDP) and make use of its multiplexing and checksum services. In this way, the conventional approach for media streaming is to use RTP/UDP for media data and RTCP/UDP for control data. However, they can use any other suitable underlying protocol or packet oriented transport protocol.

An RTP Session is defined as an association among a set of participants communicating with RTP. A participant may be involved in multiple RTP sessions at the same time. In a multimedia session, unless the encoding itself multiplexes multiple media into a single data stream, each medium (audio, video, etc.) is typically carried in a separate RTP session with its own RTCP packets. All participants in an RTP session may share a common destination transport address pair.

2.2 RTP/RTCP implementation in NS-2

As mentioned above, the implementation of RTP and RTCP protocols in NS-2 is too generic for us. It does not define many of the attributes specified in RFC 3550 or they are not defined correctly. Moreover, it only provides common transport protocol functions running on top of UDP.

Generally, specific protocols are implemented in NS-2 as Agents. These agents represent endpoints where network-layer packets are constructed or consumed, and can be used for the implementation of protocols at various layers. In this case, RTP and RTCP protocols are implemented using the *RTPAgent* class and the *RTCPAgent* class, respectively. Both classes derive from the *Agent* class and are implemented in the file *rtp.cc* (located in *~ns/common* directory) and file *rtcp.cc* (located in *~ns/apps* directory). The *Agent* superclass is implemented in both hierarchies (compiled and interpreted). Its C++ implementation is contained in *~ns/agent.cc* and *~ns/agent.h* files, and the OTcl support is in *~ns/tcl/lib/ns-agent.tcl* file. The *RTPAgent* has the functionality for sending and receiving RTP packets, whereas the *RTCPAgent* is responsible for transmission and reception of the RTCP packets.

RTPSession class (defined in *~ns/common/session-rtp.cc* file) principally deals with feedback report building and participant's information tables maintaining through the received control packets passed from its agents. This class is called by its binding OTcl class *Session/RTP* (defined in *~ns/tcl/rtp/session-rtp.tcl*

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8686
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8686

directory). It mainly defines the procedures for the session initialization, report interval calculation (and its initial value), RTP transmission rate and packet size setting, flow identifier assigning, associating new RTP sessions with nodes, managing join and leave processes to multicast groups, stopping RTP flow transmissions, liberating the session resources, etc.

All the C++ files we have cited use *rtp.h* as header file, which is located in *~ns/apps* directory. They appear in dark boxes in Figure 2.

If a node in the simulated network has to become a participant of a multicast RTP session, a new instance of the *Session/RTP* class has to be declared using the **[new Session/RTP]** command. Then, it has to be attached to each node invocating the **attach-node** method. With the new session, four objects are created: an RTP Agent (*Agent/RTP*), an RTCP Agent (*Agent/RTCP*), a Timer for the interval report calculation (*RTCPTimer*) and a source Agent (*RTPSource*). In fact, when *Session/RTP* is declared for a node, what is really attached to the node is the RTP and RTCP Agents created for that session. After that, this session has to be joined to a previous defined multicast group, by using the **join-group** method. In this process, RTP and RTCP agents are joined to the multicast group (same IP address) but with different port addresses (usually, consecutive numbers). The **start** procedure initializes the RTCP Agent whilst the **transmit** procedure launches the RTP agent. When a **transmit** procedure is called, a new Timer (*RTPTimer*) is created, which will deal with sending RTP packets with a specific rate. For each timer timeout an RTP packet will be sent and the timer will be re-scheduled with the same period (if we are simulating CBR – *Constant Bit Rate* – traffic). This timeout period (in seconds) is calculated as the relation between the RTP packet size (in bits) and the binary rate (in bits/second) specified as an input parameter in the **transmit** procedure.

During the streaming session, when an *RTPSession* receives data from a new RTP source, it includes this source in its senders information table by means of new *RTPSource* object, in order to register the sender reports statistics. In the same way, when an *RTPSession* receives control traffic (receiver reports) from new participants, it includes the receiver in its receivers information table by means of new *RTPReceiver* object, in order to hold the fields that are used by the receiving Agents for QoS measurements. The steps for new RTP Session initialization are illustrated in Figure 3. In this Figure we can also observe the appropriated syntax for the OTcl methods invocation and the associated events that are generated.

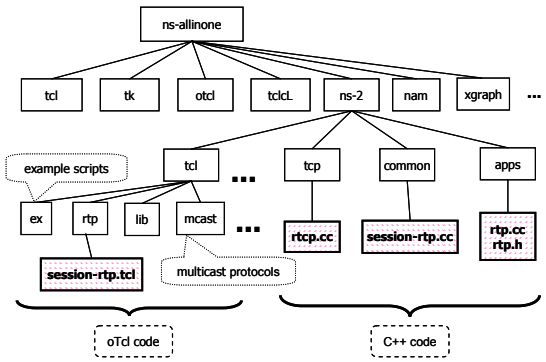

Figure 2. NS-2 directory structure.

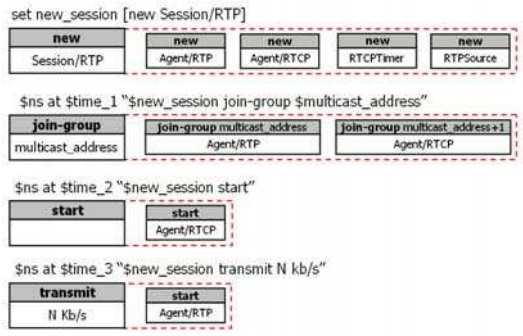

Figure 3. Basic OTcl commands for RTP/RTCP

For a better understanding of these processes, and as an example, a simple OTcl script code is presented below, in which RTP sessions are defined (lines initiated with '#' are author comments):

```
# First, a new simulation object is defined (instance of class Simulator)
set ns [new Simulator]
# The scenario will be multicast
$ns multicast
…
# We configure the multicast protocol (mproto)
# There are distinct alternatives:  CtrMcast, DM, ST, BST
set mproto DM
…
# Multicast agents are added to all the nodes
set mrthandle [$ns mrtproto $mproto {}]
…
# We define a new multicast group
set group [Node allocaddr]
…
# Network topology with a sender and two receiver nodes
set sender_node [$ns node]
set receiver_node_1 [$ns node]
set receiver_node_2[$ns node]
…
# RTP session are defined for the source and receivers
set sender_session [new Session/RTP]
set receiver_session_1 [new Session/RTP]
set receiver_session_2 [new Session/RTP]
…
# We attach the sessions to the nodes (agents are attached to the nodes)
$sender_session attach-node $sender_node
$receiver_session_1 attach-node $receiver_node_1
$receiver_session_2 attach-node $receiver_node_2
…
# Session bandwidth which will be used for the RTCP interval calculation
$sender_session session_bw 400kb/s
…
# Participants join to the multicast group
# RTCP Agents are initialized ('start' procedure invocation)
$ns at 0.1 "$sender_session join-group $group"
$ns at 0.1 "$sender_session start"
…
$ns at 0.1 "$receiver_session_1 join-group $group"
$ns at 0.1 "$receiver_session_1 start"
…
$ns at 0.1 "$receiver_session_2 join-group $group"
$ns at 0.1 "$receiver_session_2 start"
…
# RTP source initiates RTP packets transmission at 400 kbps rate
$ns at 0.5 "$sender_session transmit 400kb/s"
…
# The receivers leaveS the multicast group at 29 seconds
$ns at 29.0 "$receiver_session_1 leave-group"
…
# The source stops the RTP packets transmission
$ns at 30.0 "$ sender_session stop"
…
```

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8686
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8686

2.3 Related Works

Apart from the native implementation of RTP in NS-2, we have found two additional implementations of RTP/RTCP protocols in NS-2 including improvements to that native code and new modules for specific purposes ([6] and [7]).

On the one hand, in [6], new RTP and RTCP Agents are defined with further functionalities in order to provide loss and jitter control in MPEG-2 traces streaming over wireless environments with QoS (802.11e). In addition, new data structures are defined to generate the RTP and RTCP packets. These data structures contain more differentiated fields that the native code for RTP/RTCP but their size is not correct and some fields of these fields are not specified in [2]. The source code, MPEG-2 binary traces and the installation guide are accessible in the following URL: http://gridnet.upc.es/~vcarrascal/ns2/.

On the other hand, the Research Academic Computer Technology Institute and the University of Patras made new extensions to the legacy RTP/RTCP code in NS-2, [7], in order to, on one hand, provide this code additional features defined in RFC 3550 and related to QoS metrics (loss and jitter control) and, on the other hand, employ TCP Friendly bandwidth share behaviour of multimedia data transmission from a server to a number of receivers, through multicasting. Simulations examples, the source code and its documentation are available in the following URL: http://ru6.cti.gr/ru6/ns_rtp_extensions.php.

Both implementations have served us to comprehend better the performance of these protocols in NS-2. Nevertheless, to include our synchronization approach we need to develop a new module with complementary functionalities, extending and adding new RTCP packets, implementing new timers, receiver buffers, algorithms and methods.

3. NS-2 RTP/RTCP MODULE WITH OUR GROUP SYNCHRONIZATION APPROACH

In this section, we present the new NS-2 module based on the RTP/RTCP protocols, including our group synchronization approach, [3]. During this module implementation we pursued two main goals. On the one hand, we wanted to extend the RTP code by providing the additional attributes specified in RFC 3550 and related to QoS metrics, as in [7]. So, we modified the existing RTP and RTCP packets to adapt them to the specified format in [2], and added new RTCP messages that were not included in the above mentioned implementations, such as RTCP SDES, RTCP BYE and RTCP APP packets. On the other hand, we included in the module an optional mechanism to acquire group synchronization between receivers (group synchronization) by defining new timers, new receiver buffers, extending the existing RTCP RR packets and programming new algorithms for synchronization purposes.

3.1 RTP/RTCP files location and content

Our new module can be included together with the other built-in NS-2 modules without needing to replace the RTP legacy code. As we can see in Figure 4, the C++ code is located in *~ns/rtp_gs* directory (*"gs"* stands for group synchronization) and the OTcl code is located in *~ns/tcl/rtp_gs* directory. We defined a new header file for each C++ file in contrast to the two other found implementations mentioned in Section 2.

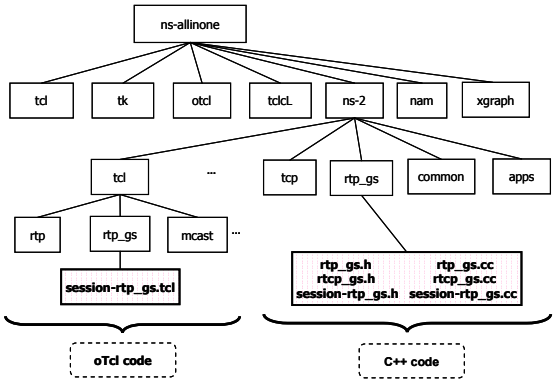

Figure 4. NS-2 directory structure after RTP/RTCP Group Synchronization modules installation.

In the *rtp_gs* files, we modified the native packet header from *"hdr_rtp"* to *"hdr_rtp_gs"*, including all the fields specified in RFC 3550 (and with the correct sizes). We redefined the RTP Agent, which holds all the functionality of sending and receiving RTP data units, and called it *RTP_gs_Agent*. We have also redefined the timer responsible for RTP packets sending, and called it *"RTP_gs_Timer"*.

In *rtcp_gs* files, we redefined the RTCP Agent responsible for RTCP packets transmission and reception, calling it *"RTCP_gs_Agent"*. We also created a new common header for all the RTCP packets with the same format specified in RFC 3550, calling it *"hdr_rtcp_gs"*, and defined new data structures for each RTCP packet.

As specified in [2], we defined a numeric constant for each RTCP packet to include it in the Payload Type field of the RTCP header:

```
typedef enum {
        RTCP_SR    = 200,
        RTCP_RR    = 201,
        RTCP_SDES =  202,
        RTCP_BYE =   203,
        RTCP_APP = 204
} rtcp_type_t;
```

SR packets (which are defined in the *"sender_report"* data structure) are generated by active participants who are sending media units (RTP sources). They describe the amount of data sent so far, as well as correlating the RTP sampling timestamp and absolute time (provided by NTP or GPS) to allow synchronization between session participants. In our simulated case, we take advantage of the existence of a global virtual time provided by the simulator scheduler clock to use it as absolute time.

RR packets (which are defined in the *"receiver_report"* data structure) are sent by participants who stand as receivers in the session. Each such report contains one block for each RTP source in the group. Each block describes the instantaneous and cumulative loss rate and jitter from that source. The block also indicates the last timestamp and the delay since receiving a sender report, allowing sources to estimate the Round Trip Time (RTT) to RTP sinks. We also defined an extended RR RTCP packet, named EXT RR RTCP packet, to include useful information for our synchronization approach, [3], which will be explained in the next sub-section.

SDES packets (which are defined in the *"source_description"* data structure) are used for session control. They contain the Canonical Name (CNAME), a participant's globally unique

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8686
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8686

identifier (similar in format to an email address), which can be used for resolving conflicts in the SSRC value and to associate different media streams generated by the same user. SDES packets also can identify the participant through their name, email, and phone number. This message also must be included in each compound RTCP packet because new receivers need to know the CNAME for a source as soon as possible to identify it and to begin to associate different media for different purposes, such as for lip synchronization (lip-sync).

When users leave an RTP session, they send BYE RTCP messages (which are defined in the *"bye"* data structure). If a participant receives an RTCP BYE packet, the SSRC associated in this packet is removed from its participants' table (senders or receivers), and the value for the session members is updated.

Finally, new APP (*Application-Defined*) RTCP packets are defined to send useful information to the receivers for the synchronization approach. These packets will be explained in the next sub-section.

In *session-rtp_gs* files, we re-implemented the "*RTPSession"* class, calling it *"RTP_gs_Session"*. When a new RTP session is instantiated in our simulation environment by means of the **[new Session/RTP_gs]** OTcl method invocation, a new C++ *RTP_gs_Session* class is returned and, in turn, two new agents are declared (*RTP_gs_Agent* and *RTCP_gs_Agent*). In addition, the *RTP_gs_Session* constructor initializes the *localsrc_* and *allsrcs_* instances of the *RTP_gs_Source* class and also the *receiver_* instance of the *RTP_gs_Receiver* class. The *localsrc_* stands for the originator of RTP and RTCP packets. It is possible that the *localsrc_* generates only RTCP packets if it is only a receiving source in the newly created session. In these C++ files, we implemented our group synchronization approach which is detailed in the next sub-section.

We provide to the NS-2 users the possibility, from their OTcl script, to enable or not our group synchronization approach in each receiver they define, by means of the **enable-gs** method invocation, defined in the OTcl **Session/RTP_gs** class. Therefore, the group synchronization will be enabled if the following OTcl command is executed:

```
#0 To disable; 1 to enable Group Synchronization mechanism
$session_name enable-gs 1
```

The default value is zero, which means that the group synchronization algorithm is disabled by default. In this way, the receivers will send conventional RR RTCP packets (not extended) and will not send the new APP RTCP packets mentioned above. Different receivers could coexist in the same OTcl script using or not the group synchronization approach.

3.2 Designing and implementing our group synchronization algorithm

Our group synchronization algorithm makes use of a *Synchronization Maestro Scheme* (SMS, [9]), based on the existence of a synchronization maestro (in our case the RTP source) which gathers the information of the playout processes of all the receivers and correct their playout timing by distributing RTCP control messages.

We tackle our synchronization problem by dividing it in two main phases: first phase, to get all the receivers starting the playout process at the same time (*Initial Playout Instant*); and second phase, to maintain the media stream playout process in a synchronized way between all the active receivers.

3.2.1 Initial Playout Instant

In the initial phase, if we suppose identical playout rates of the receivers and deterministic network delay between the source and all the receivers, we can guarantee media synchronization if the source initially indicates to all the receivers the exact instant to begin the playout of the RTP stream, referred as Initial Playout Instant (illustrated in Figure 1). For Initial Playout Instant calculation, we force the receivers to send several control messages, which we called RTCP APP RET packets, with global time information, allowing the source to estimate the network delay. The main fields of this packet are defined in the `APP_TIN_RET` data structure shown below and the complet format of the packet is illustrated in Figure 5.

```
struct APP_TIN_RET {
        /*source this RTCP packet refers to*/
        u_int32_t  sender_srcid_;
        /* name this RTCP packet refers to */
        char  name[4];
        double ntp_time_;
};
```

In the OTcl script configuration, the interval for sending the RTCP APP RET messages can be adjusted in each receiver by means of the **APPRET-Interval** OTcl method invocation, indicating this period as an input parameter. The default value for this interval was established to 100 ms, but this value will be dynamically adjusted according to the network load as specified in [2]. A new timer will take care of these packets transmission, including the global timestamp information of the time when the packet is sent. The source will receive these messages and register the network delay for each incoming message (difference between source's global time and the one stamped in the received message) to estimate the maximum, minimum and mean (using a temporal window) network delay value for each receiver.

```
void RTP_gs_Session::recv_ctrl(Packet* p)
{
   …
   /* we get the report header */
   hdr_rtp* rh = hdr_rtp::access(p);
   …
   /* if the received control packet is an APP RET
   packet*/
   if(rh->app_t_r_!=0)&&(rh->app_t_r_->name[0]=='R'))
   {
   …
   /* this function registers the delay for each
   incoming RET message and uses it to estimate the
   maximum, minimum and mean delay */
   delay_estimation(rh->srcid(),rh->app_t_r_->ntp_time_);
   …
}
```

Once the source has estimated the maximum and minimum delay, it uses these values to calculate the Initial Playout Instant, as explained in [3]. Then, the source sends to the receivers another RTCP APP packet, we called APP TIN packet, to indicate the global clock time when the receivers' playout process must start. The format for both packets is identical but they have different ASCII name ("RET" and "TIN"). In the NTP timestamp field of TIN messages the source indicates the global time when the receivers' playout process must start.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8686
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8686

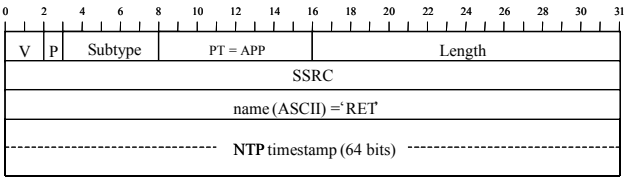

Figure 5. RTCP APP RET/TIN packet format.

After the RTCP APP TIN packet transmission, the source will start sending RTP data units, which will be buffered by the receivers until their local clock reach the Initial Playout Instant. In the receiver, 5 values are stored for each incoming RTP packet: its sequence number, its timestamp, the buffer input instant, the jitter value and the playout delay for that data unit. The C++ *buffer_iterator_* variable will be used to place the incoming packets in each receiver buffer and to manage its occupancy.

In this way, our group synchronization approach initial phase guarantees that all the receivers initiate the playout at the same time, as shown in Figure 1.

3.2.2 Fine synchronization between receivers

In the second phase of our approach, when the source starts sending RTP data units, the receivers will use extended Receiver Report RTCP packets (EXT RR packets) as feedback messages, with new extensions, including the sequence number of the current data unit the receiver is playing and the global time (NTP) timestamp of the instant in which the receiver started the playout of that data unit. The main fields of these packets are defined in the `receiver_report` data structure shown below and the complet format is illustrated in Figure 5.

```
/** rtcp_gs receiver reports */
struct receiver_report {
        /* data source being reported */
        u_int32 srcid_;
        /* fraction lost since last SR/RR */
        unsigned int fraction_lost_:8;
        /* total number of RTP packets lost since
        the beginning of the session (signed!) */
        int cum_pkts_lost_:24;
        /* last SR time from this source */
         double lsr_;
        /* delay since last sender report */
         double dlsr_;
        float jitter_; /* interarrival jitter */
        /*   last   sequence   number   data   unit
        played from all the sources  */
        u_int16_t seq_LDU_RR;
        ...
}
```

The receivers will send these packets at the interval specified in [2]. *The* `RTCP_gs_Agent` calls the build report function (`build_report()`), located in *session-rtp_gs.cc* file, as a result of the `RTCP_gs_Timer` timeout event.

```
   void RTCPAgent::timeout(int)
        {
        …
        if(running_)
        {
        size_ = session_->build_report();
        sendpkt();
        …
        }
        }
```

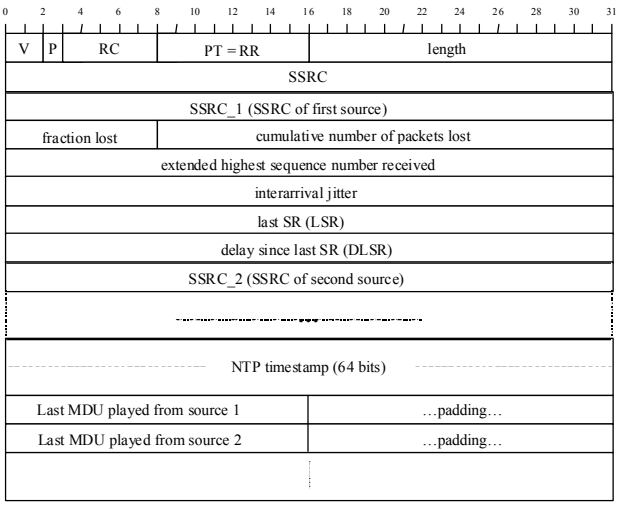

Figure 6. RTCP RR EXT packet format.

On one hand, the RTP sender generates new SR if it has sent RTP data units since the previous sent SR. On the other hand, each receiver constructs new EXT RR packets, in the same build report function, if it has received new RTP data units from a source. Its fields are completed as specified in the Appendix A.8 in [2].

```
     /* add receiver report */
     receiver_report* rr;
     rr = new receiver_report;
     /* fill the report */
     rr->receiver_srcid()=localsrc_->srcid();
     rr->cum_pkts_lost() = sp->cum_pkts_lost();
     …
     rr->jitter()= sp->jitter();
     rr->LSR() = sp->LSR();
     rr->DLSR()= NOW - sp->SRT();
     …
     rr->ntp_time_ = ntp_playing();
     rr->seq_LDU_RR_=seq_playing
     /* add the RR EXT to the RTCP packet*/
     rh_->rr_ = rr;
```

During the session, the source receives these RR EXT packets from the receivers and stores the information required by our approach in a memory table (specifically, receiver identifier (SSRC), the last data unit played by this receiver and the global time timestamp in which the data unit was played). Due to this, we previously included those required fields in the *RTP_gs_Receiver* class.

```
void RTP_gs_Session::recv_ctrl(Packet* p)
{
…
/* if is the source */
if(localsrc_->is_sender())
{
        /* get the Receiver report */
        if(rh->rr_!= 0)
        {
        …
        /* for each receiver (s), the source store
        these values */
        s->seq_LDU_RR_=rh->rr_->seq_LDU_RR_;
        s->ntp_time_=rh->rr_->ntp_time_;
        ckeck_ACT_sending()
        …
        }

}
```

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8686
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8686

This memory table must be updated with the most recent control messages received by each receiver. For each incoming RTCP RR EXT message, the source checks if it has received the reports from all the active participants and, if it is true, the source activates a *sending_ACT_message* boolean flag. In relation to this, we defined a new *RTCP_APP_Timer* which is continuously supervising this flag and, in case it is active, it calls a build function for new APP control packets (`build_APP_pkt(type)`).

```
void RTCP_gs_Agent::timeoutAPP(int)
{
        if(sending_APP_message)
        {
        size_=session_->build_APP_pkt(type);
        …
        sendpkt();
        /* the flag is deactivated */
        sending_APP_message= false;
        …
        }
}
```

Here, when the source has obtained the playout information of all the active receivers, it can estimate the state of the receivers' playout process.

```
int RTPSession::build_APP_pkt(int type)
{
        /* 1=ACT, 2=RET, 3=TIN*/
        if (type == 1)
        {
        …
        master_recv_select(master_alg_,async_);
        ACT_parameters_calculation();
        …
        }
        …
}
```

In this function, the source selects a receiver as the master receiver, according to a master selection algorithm variable (e.g. the faster or the slower one), which we called *master_alg_* and must be specified in the following OTcl invocation:

#01= To the mean, 10= To the slower, 11= To the faster
$session_name master-algorithm 11

The master receiver playout point will be taken as the reference to determine the playout point state (advanced or delayed from that one) in the other receivers (slave receivers). If the source detects an asynchrony (deviation) between the receivers' playout processes exceeding an *async_* threshold (configurable value in the OTcl script), it will multicast action messages to make the receivers correct their playout timing. As a result, late slave receivers will accelerate their playout timing and fast slave receivers will restrain their playout timing.

These action messages are new control APP RTCP packets, called APP ACT packets, with an extension including a data unit sequence number and the global time in which this data unit should be played by all the receivers. The main fields of the packet are defined in the APP_ACT data structure shown below and the format is illustrated in Figure 5.

```
struct APP_ACT {
        u_int32_t sender_srcid_;
        char name[4];
        double ntp_time_;
        u_int16_t seqno_;
        …
};
```

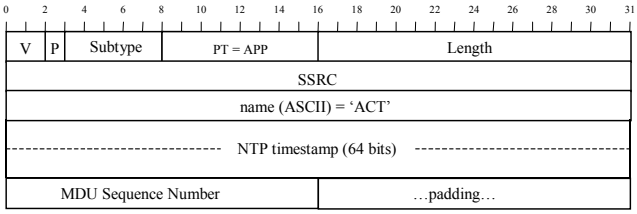

Figure 6. RTCP RR EXT packet format.

The APP ACT RTCP parameters are calculated in the `ACT_parameters_calculation()` function according to our previous work in [3].

4. EVALUATION

We have proved our approach in several network topologies by means of running multiple simulations with similar satisfactory results. With these simulations we pursued two objectives: first, verify the proper functioning of this code, and second, test our group synchronization performance.

Consequently, in this section we show only a typical wired scenario and the results of a single run, with the purpose of showing clearly the good performance of the implementation of our group synchronization approach. The chosen simple simulation scenario is illustrated in the Figure 7. It consists of one RTP Source (simulating a multimedia server) and three RTP Receivers distributed over the network topology (with different end-to-end delays between the source and receivers and different network load between them). All the links have a capacity of 1.5 Mbps and the delay between the network components is detailed in the same Figure.

In this simulation the RTP Source transmitted RTP packets with a specific rate of 400 kb/s. The RTP packets size was set to 1000 bytes. The RTCP transmission interval was initially set to 0.3 seconds to all the participants, but this value was dynamically adjusted according to the network load. In addition, we intentionally configured background CBR/UDP traffic between the *CBR Source* and the *RTP Receiver 3* in order to create a *bottleneck* in this link, affecting the delay and jitter estimations for that receiver.

For the multicast transmission we configured the PIM-DM (*Protocol Independent Multicast – Dense Mode*) protocol.

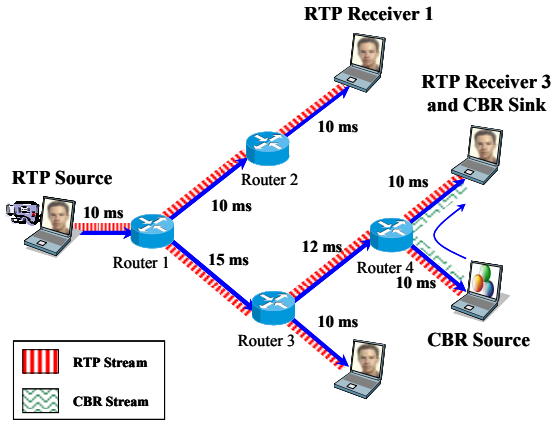

Figure 7. Simulated network topology.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8686
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8686

In our simulation case, we forced a random deviation or *drift* in the receivers playout rate limited by $\pm\rho$. In this way, if the receivers have a nominal playout rate of θ data units per second (equal to the source transmission rate), in the worst case, one receiver would be able to playout the stream with the maximum rate of $\theta*(1+\rho)$, whereas another receiver would be able to playout the RTP stream with the minimum rate of $\theta*(1-\rho)$. This playout drift can be configured by means of the **assign-drift** OTcl method invocation. In this single simulation, we assigned a random deviation limited by ±10% to all the receivers. By default the receivers' playout process drift is zero (i.e. the RTP packets are played at the same rate that they were generated by the source). Concretely, during the simulation, whose results are shown in next figures, Receiver 1 played the RTP data units with a nominal rate of θ_1=48.78 data units per second, i.e., each data unit was played during 20.5 ms; Receiver 2 played the RTP data units with a nominal rate of θ_2=50.2 data units per second, i.e., each data unit was played during 19.9 ms; and Receiver 3 played the RTP data units with a nominal rate of θ_3 = 45.45 data units per second, i.e., each data unit was played during 22 ms.

4.1 Without the Group synchronization approach

Figure 8 shows the consumption process of the RTP data units in all the receivers throughout the session, with the group synchronization mechanism disabled. In it, we can observe how the asynchrony between the receivers is continuously increasing due to the forced deviations in their playout rates.

In Figure 9, a zoom view of the left bottom corner of the previous graphic is presented. We can notice how the playout processes of the receivers were not synchronized at the Initial Playout Instant without enabling our group synchronization approach. Furthermore, we can appreciate in this graphic how faster receiver (Receiver 2) advances in their RTP data units consumption process to slower receivers (Receivers 1 and 3), despite the fact that the playout processes of both slower receivers start before. It was produced due to the random playout rate deviations ($\theta_2 > \theta_1 > \theta_3$) chosen for this simulation.

4.2 With the Group synchronization approach

In this case, we enabled our group synchronization mechanism to all the receivers by means of the **enable-gs** OTcl method invocation. We also indicated to the RTP source the master receiver selection algorithm to use. This was done by means of the **master-algorithm** OTcl method invocation. In this command, we set the *master_alg_* variable value to 11, so slower (slave) receivers (Receiver 1 and Receiver 3) took the playout point of the faster (master) receiver (Receiver 2) as a the reference for synchronization, because its playout rate was higher than the ones of the slave receivers ($\theta_2 > \theta_1 > \theta_3$).

In Figure 10, we can observe how slave receivers playout processes were adjusted, at several points, to the master receiver playout process as a reactive response to the action messages (RTCP APP ACT packets) reception from the RTP source. In this case, as a result of the ACT messages, slower receivers accelerated their playout processes to synchronize to the playout process of the fastest one (in continuous red line).

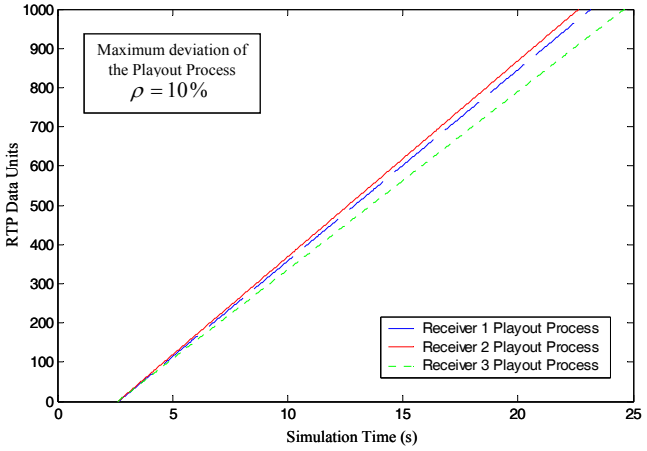

Figure 8. Receivers' playout process without group synchronization.

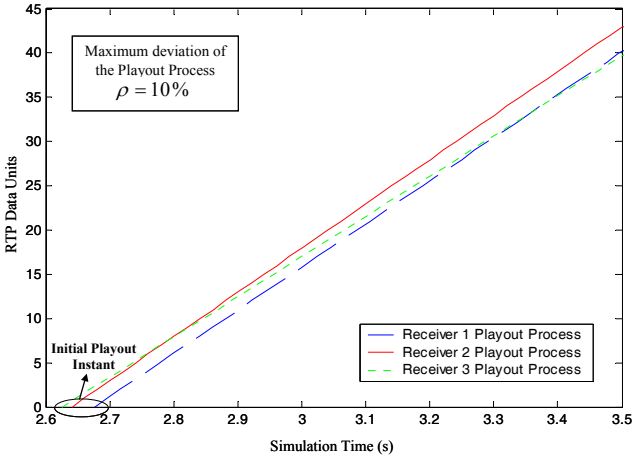

Figure 9. Initial Playout Instant without group synchronization.

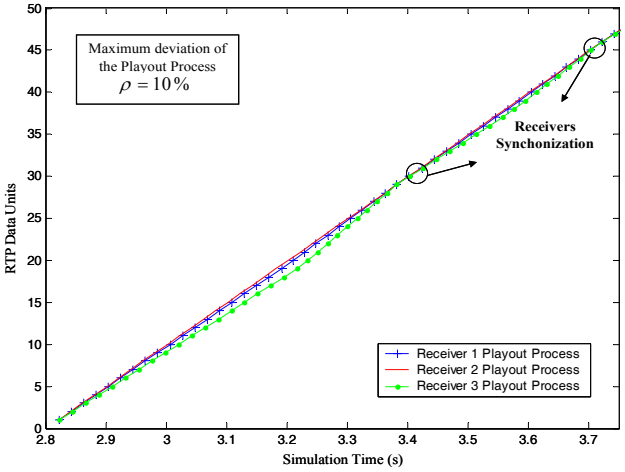

Figure 10. Receivers' playout process with group synchronization.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8686
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8686

In Figure 11, which is a zoom view of the the previous graphic, we appreciate that all the receivers began their RTP data units playout processes at the same time (Initial Playout Instant), as a result of the initial phase of the group synchronization approach.

During the first minute of the simulation, 2869 RTP data units were sent by the RTP source. Additionally, the source sent a number of APP ACT messages that supposed around 2 % regarding the total number of RTP data units it sent, including the only one RTCP APP TIN message to communicate the Initial Playout Instant. In our previous work [3], the number of RTCP APP ACT packets only supposed 1,1 % regarding the total number of RTP packets sent. The reason was because in the present work, a RTCP APP ACT is sent when the source receives the RTCP RR EXT packets from all the receivers, since the *async_* threshold value was set to 0 ms. In the real case, the RTCP APP ACT were sent only if the source detected an asynchrony exceeding a threshold value of 120 ms [3]. Additionally, during the simulated session, each receiver sent around 190 RTCP RR EXT control messages. It supposed around 6 % regarding the total number of RTP data units sent by the source. The overload generated by the synchronization approach consists of 192 bits of each ACT packets and the 80 bits extension of each EXT RR packets. Notice that the common part of RR packets is also sent when the approach is not enabled. So, the overload introduced by our approach is very low. Moreover, in this simulation, it is very small compared to the 1000 Bytes of each RTP data unit sent by the source.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a new module for NS-2 based on RTP and RTCP standard protocols. On the one hand, we have improved the native code in NS-2 for these protocols by including all the functionalities specified in [2]. On the other hand, we have included an optional functionality to this module in order to achieve group synchronization between the RTP receivers distributed in the simulated network. This group synchronization approach was developed and tested in real WAN scenarios in a previous work, [3].

One of the most important characteristics of our approach is that the overload generated is very low because we do not define a new protocol for synchronization purpose. Instead, we use well-known protocols as RTP/RTCP and we take advantage of their extension capabilities. We extended two types of RTCP packets with useful information for synchronization purposes: RTCP RR EXT and RTCP APP packets.

The proposed group synchronization solution has obtained satisfactory results in our evaluation, which validates it as a possible solution for multimedia applications which require group synchronization.

For future work, we plan to adapt this code to make it valid for Cluster-to-Cluster applications, in which one or more sources, located in a sender cluster, transmit (point-to-multipoint or multipoint-to-multipoint), in one-way, independent but semantically related data streams to end systems distributed in one or several receiver clusters. In this context, when there is more than one receiver cluster, the source should know which receivers belong to each receiver cluster and should perform the synchronization calculations separately for each cluster, only taking into account the feedback reports received from the

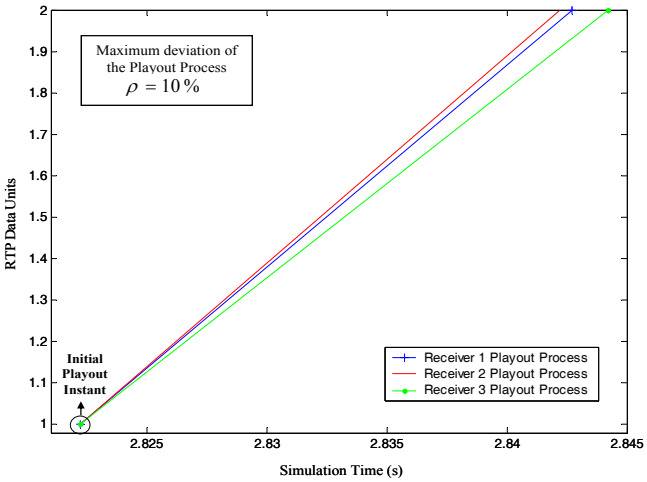

Figure 11. Initial Playout Instant without group synchronization.

receivers of each cluster. Then APP ACT packets would be sent only affecting each cluster separately.

We also have to improve this code to support multiple RTP streams in the same node and to provide inter-stream synchronization between them. In addition, we want to extend this code to allow that a node can be a receiver and, at the same time, an active sender to use it in bidirectional communications and to be able to synchronize interactive services, such as network games or collaborative work applications, etc.

Finally, the source code, its installation guides and simulation examples will soon be available in the following URL: http://personales.gan.upv.es/~fboronat/MultimediaGSynch.html

6. ACKNOWLEDGMENTS

This work has been financed partially by Polytechnics University of Valencia (UPV) under its R&D Support Program in PAID-06-08-002-585 and PAID-05-09-4335 Projects.

7. REFERENCES

[1] NS-2 Simulator. http://www.isi.edu/nsnam/ns

[2] Schulzrinne, H., Casner, S., Frederick R. and Jacobson V. 2003. RTP: A Transport Protocol for Real-Time Applications. RFC-3550, July 2003.

[3] Boronat F., Guerri, J. C. and Lloret, J. 2008. An RTP/RTCP based approach for multimedia group and inter-stream synchronization, Multimedia Tools and Applications Journal, Vol. 40 (2), pp. 285-319, 2008.

[4] Rangan, P. V., Ramanathan, S., Sampathkumar, S., "Feedback techniques for continuity and synchronization in multimedia information retrieval" ACM Transactions on Information Systems (TOIS), Vol. 13, Issue 2, April 1995, pp 145 – 176, ISSN:1046-8188.

[5] Guerri, J.C., Esteve, M, Palau, C.E., Casares, V., "Feedback Flow Control with Hysteresial Techniques for Multimedia Retrievals", Multimedia Tools and Applications, 13(3), pp. 307-332, 2001.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8686
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8686

[6] Carrascal, V., http://sertel.upc.es/_vcarrascal/ns2/, Grupo de Servicios Telemáticos, Universidad Politécnica de Cataluña, España.

[7] Bouras C., Gkamas A., and Kioumoutzis G. SIMUTools 2008. Extending the Funtionality of RTP/RTCP Implementation in Network Simulator (NS-2) to support TCP friendly congestion control. http://ru6.cti.gr/ru6/ns_rtp_extensions.php

[8] Simek M., Komosny D., Burget R. 2007. One Source Multicast Model using RTP in NS2. International Journal of Computer Science and Network Security, Vol.7 No.11, November 2007.

[9] Ishibashi, Y. and Tasaka, S. 1997. A group synchronization mechanism for live media in multicast communications. In Conf. Rec. IEEE GLOBECOM' 97, pp. 746–752, November 1997.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8686
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8686