University of Cyprus
Department of Computer Science
Networks Research Laboratory

Adaptive Methods for the Transmission of Video Streams in Wireless Networks

Deliverable 3.1

Simulation Software

Project Title	:	Adaptive Methods for the Transmission of Video Streams in Wireless Networks
Deliverable Type	:	Intermediate Progress Report (M18)

Deliverable Number	:	D3.1
Title of Deliverable	:	Simulation Software
Work Package Title	:	Simulation of Adaptation Techniques
Work Package Number	:	WP3
Internal Document Number	:	
Contractual Delivery Date	:	
Actual Delivery Date	:	
Author(s)	:	Pavlos Antoniou Vasos Vassiliou Andreas Pitsillides Ioakim Sykopetritis
Email(s)	:	paul.antoniou@cs.ucy.ac.cy vasosv@cs.ucy.ac.cy Andreas.Pitsillides@ucy.ac.cy asykopetritis@gmail.com

Abstract

Having designed the two feedback techniques for the increase of the objective and subjective (perceptual) quality as well as two congestion control protocols for high speed networks, we need to provide a simulation framework for the evaluation of their performance. A combined wired and wireless network will be simulated and the proposed techniques will compose a fully adaptive system which will be tested for its ability to support real time multimedia applications in various realistic or extreme scenarios. This deliverable focuses on the development of simulation modules for the novel Content and Network adaptation techniques. We are going to give an in depth description of the simulation modules that were designed for the evaluation of the proposed algorithms.

Keywords:.

Table of Contents

List of Figures

List of Tables

1 Introduction

In Deliverable 2.2 we proposed two novel adaptive approaches for video transmission over wireless networks as well as two new congestion control algorithms for high speed networks. We also gave an in depth description of all the algorithms focusing on key ideas, functionalities and the different parameters that influence their operation.

Having designed the feedback techniques for the increase of the objective as well as the subjective (perceptual) quality, we have to provide a simulation framework for the evaluation of their performance.

This deliverable focuses on the development of simulation modules for Content and Network adaptation techniques. We are going to give an in depth description of the simulation modules that were designed for the evaluation of the proposed video transmission algorithms. A combined wired and wireless network will be simulated and the proposed techniques will compose a fully adaptive system which will be tested for its ability to support real time multimedia applications in various realistic or extreme scenarios. Moreover we will conduct some experiments based on these modules and the emerged results will be presented in detail in Deliverable 3.2.

In Section 2 of this deliverable we concentrate on the evaluation procedures and simulation modules for each algorithm.

2 Evaluation Procedures and Simulation Modules

The simulations in this deliverable are intended to illustrate the use of the proposed feedback adaptive models for wireless video transmission using NS2 [1]. We use both simple and complicated topologies as well as single and multiple video stream flows as workload.

Furthermore, the procedures and functions followed for the performance evaluation of each algorithm will be also presented in detail focusing primarily on the software packages and mathematical models as well.

The rest of this section is organized as follows. Section 2.1 provides an in depth description of the simulation module and procedures regarding the Adaptive Feedback Algorithm for Internet Video Streaming based on Fuzzy Control (ADIVIS) while Section 2.2 presents the module and procedures implemented for Receiver-driven Adaptive Feedback algorithm (RAF).

2.1 ADIVIS Evaluation Procedures and Simulation Modules

The ADIVIS algorithm deals with video streaming over the wireless Internet using a fuzzy controlled decision algorithm at the video streaming server side and a feedback algorithm. The feedback algorithm takes into consideration both receiver's critical information and network-oriented measurements in order to evaluate the available bandwidth of the network path. The two aforementioned procedures are illustrated in Fig. 1.

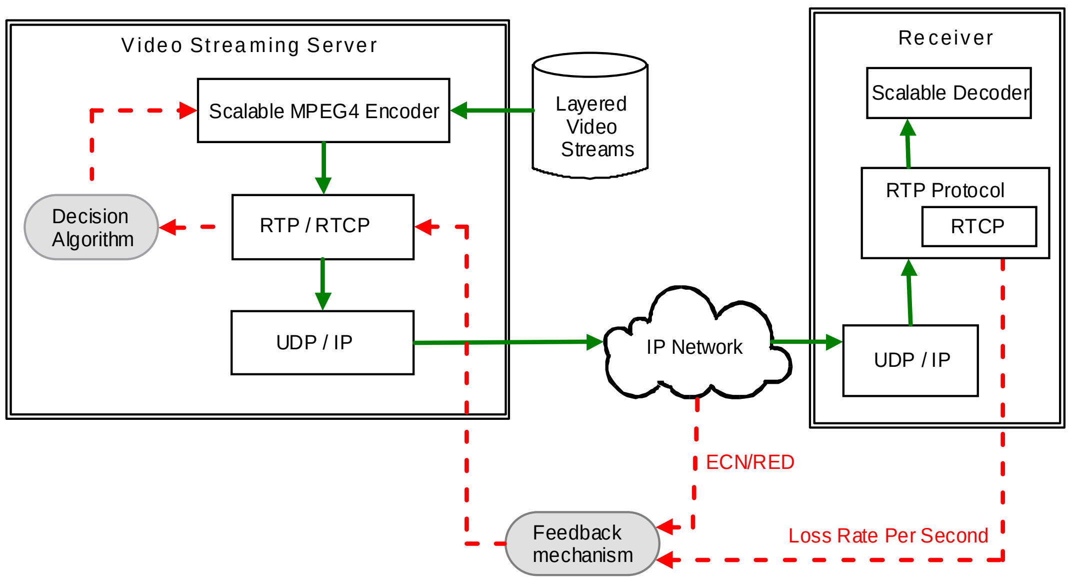

Fig. 1. ADIVIS scheme.

The available bandwidth (calculated by the feedback mechanism) is fed into the decision algorithm which intuitively decides the optimal number of layers that ought to be sent according to the available transmission rate.
This scheme requires that the video streams are encoded in a layered manner using a scalable encoder. Layered information needs to be adapted for a number of transmission rates in order to have smooth and optimal adaptation to the available bandwidth. The techniques for reducing the transmitted information are primarily based on dropping or adding layers.
We will compare adding/dropping layers and switching among different versions of the video and we will investigate how the layered information needs to be adapted for a number of transmission rates. To achieve optimal adaptation to any transmission rate we believe that it is necessary to have a large number of layers. However, we need to be aware of possible drawbacks of having a large layer number, such as difficulty in separating/generating the layers at the source, and find the optimal number for them.
These issues will be investigated thoroughly through various simulations conducted using the widely known Network Simulator version 2 (ns2). For this purpose we implemented some software modules written in C++ programming language and TCL scripting language. In this section we intend to give basic directions and comments on chosen points of the source code. The rest of this section is organized as follows. Section 2.1.1 mentions the operating system requirements and the packages required prior the installation of the simulation modules. Section 2.1.2 gives an overview of the QoS assessment framework including Evalvid ([3], [4], [5]) that will be used for end-to-end delay, end-to-end jitter and PSNR evaluations while Section 2.1.3 gives a detailed description of the new ns2 agents. Finally Section 2.1.4 deals with the operations that will be executed prior the simulations/scenarios.

2.1.1 Operating System and Required Packages

Before getting into the insights of our source code let us mention the operating system options and requirements needed prior the installation of our simulation modules. All the simulations were conducted in a linux-based Fedora Core 2 [2] operating system. All the required packages are shown in the table below.

Required Packages	Description
NS2 [1]	NS is a discrete event simulator targeted at networking research. Ns provides substantial support for simulation of many protocols, routing, and multicast protocols over wired and wireless (local and satellite) networks. All our simulations were conducted under this framework.
Evalvid in NS2 [3], [4] [5]	EvalVid [5] is a complete framework and tool-set for evaluation of the quality of video transmitted over a real or simulated communication network. It was integrated into NS2 as shown in [3] and [4]. We used it for end-to-end delay, end-to-end jitter and PSNR evaluations in NS2-based simulations. The commands we used for these evaluations will be shown below.
FFmpeg Multimedia System [6]	FFmpeg is a very fast video and audio converter. The command line interface is designed to be intuitive, in the sense that FFmpeg tries to figure out all parameters that can possibly be derived automatically. You usually only have to specify the target bit rate you want. We used for the encoding of raw video test sequences as will be shown later.

Table 1. Operating System and required packages used in ADIVIS.

2.1.2 Overview of QoS Assessment Framework

In this subsection we will give an overview of the QoS assessment framework including Evalvid. Evalvid is a complete framework and tool-set for evaluation of the quality of video transmitted over a real or simulated communication network. It was initially designed for the evaluation of the quality of video transmitted over a real or simulated communication network [5]. Besides measuring QoS parameters of the underlying network, like loss rates, delays, and jitter, it also supports a subjective video quality evaluation of the received video based on the frame-by-frame PSNR calculation. The tool-set has a modular construction, making it possible to exchange both the network and the codec. EvalVid is targeted for researchers who want to evaluate their network designs or setups in terms of user perceived video quality. The tool-set is publicly available in [7].
The structure of the Evalvid framework is shown as follows.

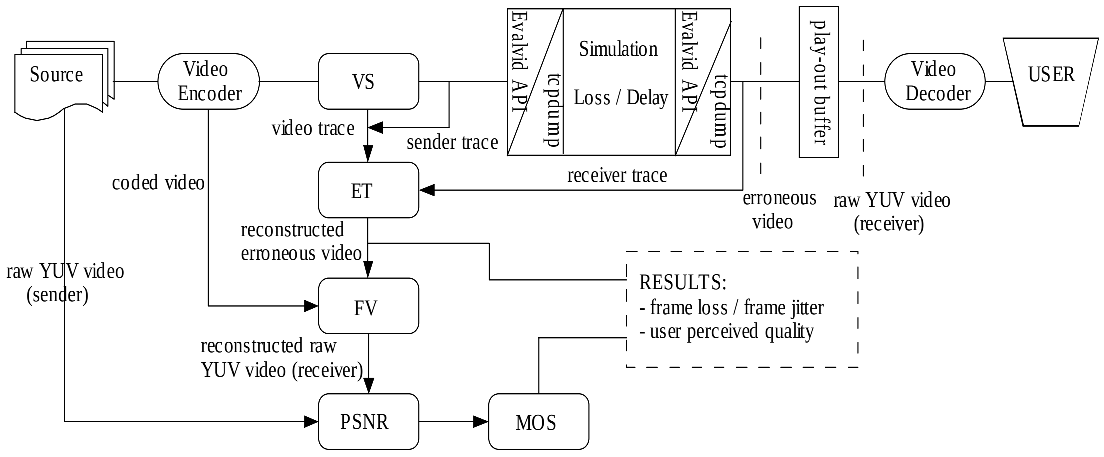

Fig. 2. Structure of the Evalvid framework.

The main components of the evaluation framework are described as follows:

Source: The video source can be either in the YUV QCIF (176 x 144) or in the YUV CIF (352 x 288) formats.

Video Encoder and Video Decoder: Currently, EvalVid supports two MPEG4 codecs, namely the NCTU codec and FFmpeg. In the present investigation, we arbitrarily choose the FFmpeg codec for video coding purposes that will be presented later.

VS (Video Sender): The VS component reads the compressed video file from the output of the video encoder, fragments each large video frame into smaller segments, and then transmits these segments via UDP packets over a real or simulated network. For each transmitted UDP packet, the framework records the timestamp, the packet id, and the packet payload size in the sender trace file with the aid of third-party tools, such as tcp-dump or win-dump, if the network is a real link. Nevertheless, if the network is simulated, the sender trace file is provided by the sender entity of the simulation. The VS component also generates a video trace file that contains information about every frame in the real video file. The video trace file and the sender trace file are later used for subsequent video quality evaluation.

ET (Evaluate Trace): Once the video transmission is over, the evaluation task begins. The evaluation takes place at the sender side. Therefore, the information about the timestamp, the packet id, and the packet payload size available at the receiver has to be transported back to the sender. Based on the original encoded video file, the video trace file, the sender trace file, and the receiver trace file, the ET component creates a frame/packet loss and frame/packet jitter report and generates a reconstructed video file, which corresponds to the possibly corrupted video found at the receiver side as it would be reproduced to an end user. In principle, the generation of the possibly corrupted video can be regarded as a process of copying the original video trace file frame by frame, omitting frames indicated as lost or corrupted at the receiver side. Nevertheless, the generation of the possibly corrupted video is trickier than this and the process is further explained in more details later. Furthermore, the current version of the ET component implements the cumulative inter-frame jitter algorithm for play-out buffer. If a frame arrives later than its defined playback time, the frame is counted as a lost frame. This is an optional function. The size of the play-out buffer must also be set, otherwise it is assumed to be of infinite size.

FV (Fix Video): Digital video quality assessment is performed frame by frame. Therefore, the total number of video frames at the receiver side, including the erroneous ones, must be the same as that of the original video at the sender side. If the codec cannot handle missing frames, the FV component is used to tackle this problem by inserting the last successfully decoded frame in the place of each lost frame as an error concealment technique.

PSNR (Peak Signal Noise Ratio): PSNR is one of the most widespread objective metrics to assess the application-level QoS of video transmissions. It was further analyzed in D1.1. The following equation shows the definition of the PSNR between the luminance component Y of source image S and destination image D:

$$PSNR(n) = 20\log_{10}\left(\frac{V_{peak}}{\sqrt{\frac{1}{N_{col}N_{row}}\sum_{i=0}^{N_{col}}\sum_{j=0}^{N_{row}}\left[Y_S(n,i,j) - Y_D(n,i,j)\right]^2}}\right) dB,$$

where Vpeak = 2k-1 and k = number of bits per pixel (luminance component). PSNR measures the error between a reconstructed image and the original one. Prior to transmission, one may then compute a reference PSNR value sequence on the reconstruction of the encoded video as compared to the original raw video. After transmission, the PSNR is computed at the receiver for the reconstructed video of the possibly corrupted video sequence received. The individual PSNR values at the source or receiver do not mean much, but the difference between the quality of the encoded video at the source and the received one can be used as an objective QoS metric to assess the transmission impact on video quality at the application level.

MOS (Mean Opinion Score): MOS is a subjective metric to measure digital video quality at the application level. This metric of the human quality impression is usually given on a scale that ranges from 1 (worst) to 5 (best). In this framework, the PSNR of every single frame can be approximated to the MOS scale using an appropriate mapping.

In addition Evalvid was implemented in NS2 as shown in [3] and [4]. Fig. 4 illustrates the QoS assessment framework for video traffic enabled by the new tool-set that combines EvalVid and NS2. The NS2 environment cloud will be further analyzed in and Fig. 5. Initially the raw (uncompressed) video streams are encoded in MPEG4 format using the FFmpeg [6] Video Encoder software and are stored at Video Sender side. Video Sender stores many streams having each one of them encoded in a different bit rate in order to simulate layered transmission. These encoded streams are fragmented into packets and transmitted from a dedicated video streaming server to a client in a layered manner using the NS2 simulation environment (see and Fig. 5). The transmitted packets experience the propagation delay and packet loss while traversing through the network path. The packets that finally manage to reach the client are registered in a dedicate file by the client. After the end of the NS2 simulation we are able to reconstruct the received video stream at client side using FFmpeg Video Decoder. It is expected that the reconstructed video stream will consist of frames (I, B, P) belonging to several layers.

Moreover we use ET program (included in Evalvid framework) in order to perform the aforementioned evaluations (end-to-end delay, jitter, PSNR, lost packets/frames). All these functions are performed using Unix AWK scripts.

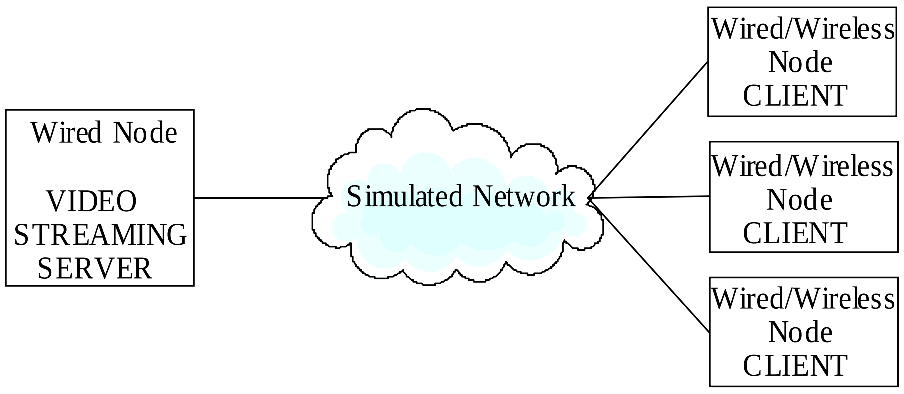
Wired Node
VIDEO STREAMING SERVER
Simulated Network
Wired/Wireless Node CLIENT
Wired/Wireless Node CLIENT
Wired/Wireless Node CLIENT

Fig. 3. NS2 Enviroment.

Fig. 3. NS2 Enviroment. depicts the NS2 simulation environment which includes the Video server and the wired/wireless clients.

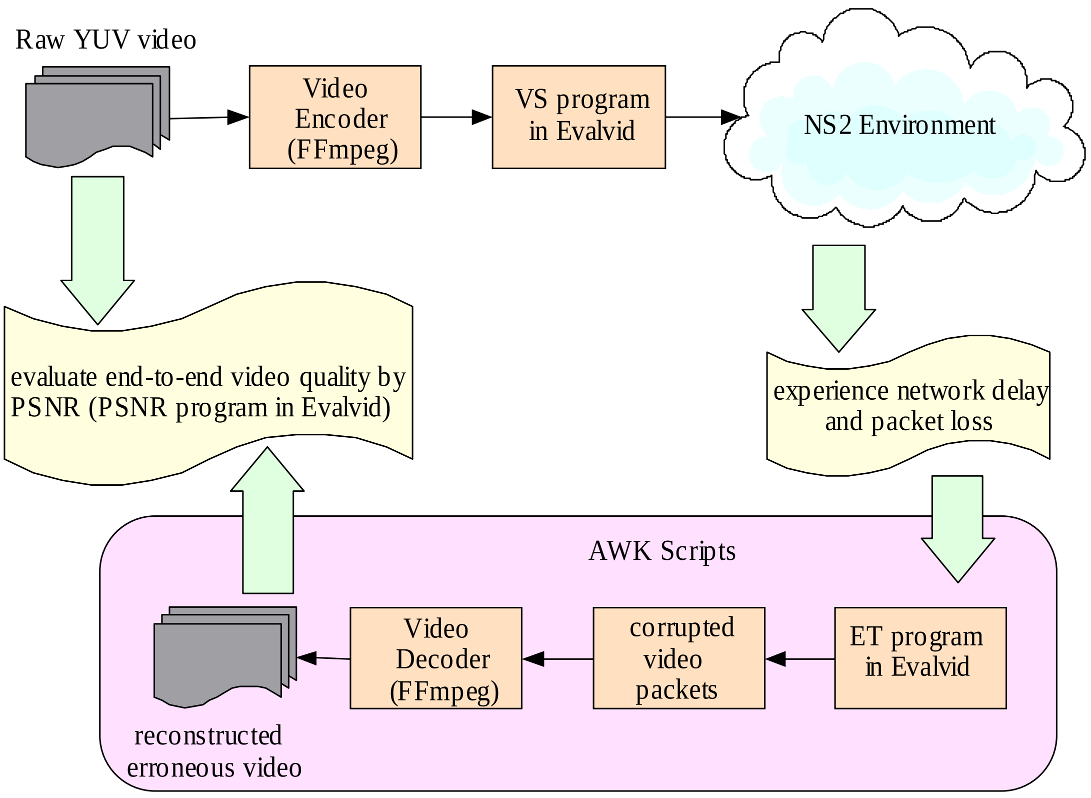

Fig. 4. QoS Assessment Framework including Evalvid.

The agents implemented between NS2 and EvalVid as shown in Fig. 4. These interfaces are designed either to read the video trace file or to generate the data required to evaluate the video delivered quality.

Protocols in NS2 are represented by objects, which are constructed as derived classes of the base Agent class (written in C++). If the protocol supports session mechanisms, the derived classes of base Session class (also written in C++) will be constructed. Above C++ classes are mirrored into a similar class hierarchy within the OTcl interpreter. For example, the RTP/RTCP implementation in NS2 environment consists of three C++ classes:

RTPAgent – RTP protocol processing mechanisms,
RTCPAgent – RTCP protocol processing mechanisms,
RTPSession – the RTP session management.
C++ classes are closely mirrored by corresponding objects in the OTcl class hierarchy, respectively, Agent/RTP, Agent/RTCP and Session/RTP.

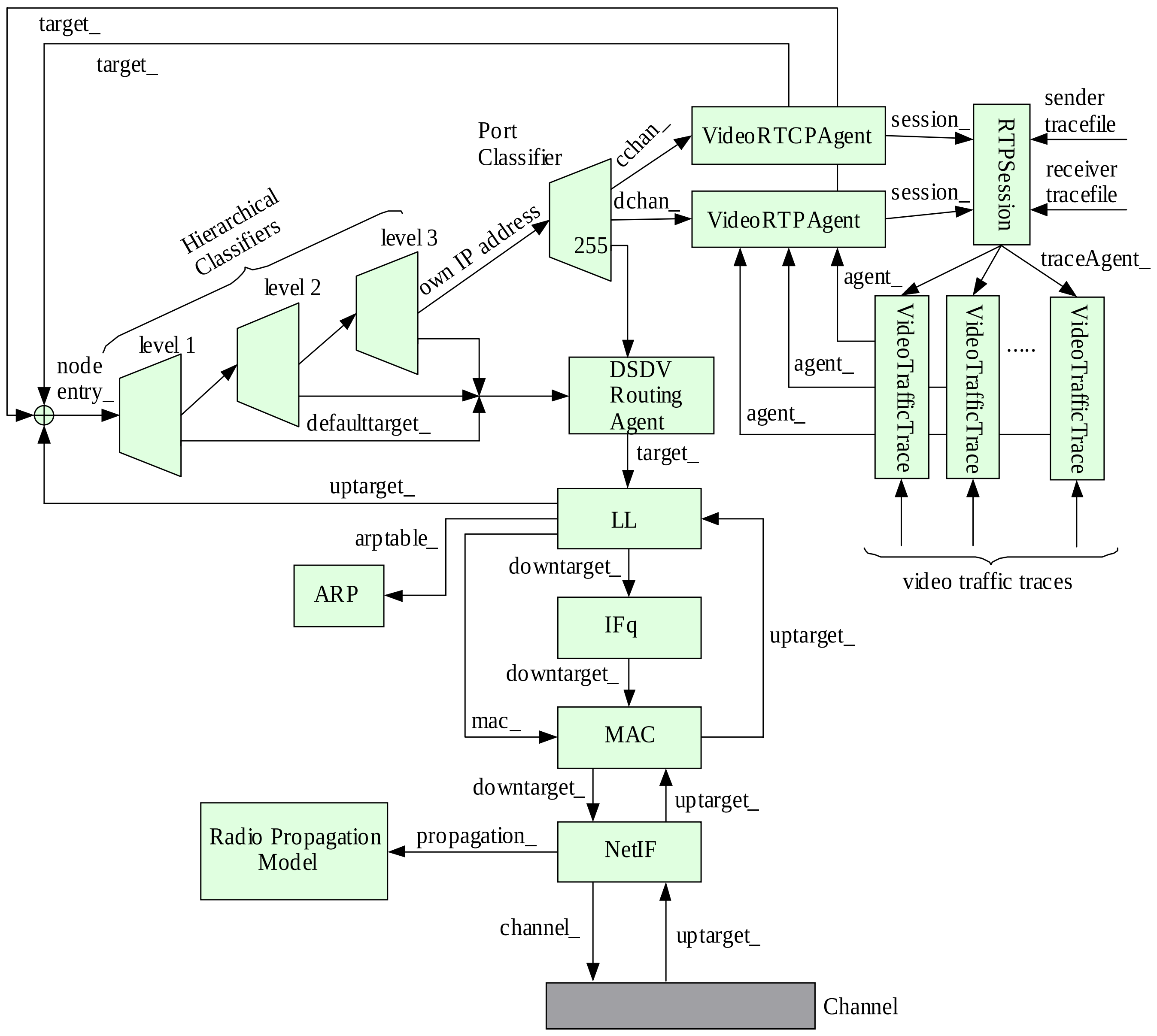

Fig. 5. Wired/Wireless Node Model in NS2.

The proposed NS2 environment is based on the existing agents. Four new classes were written or modified in C++ (VideoRTPAgent, VideoRTCPAgent, VideoTrafficTrace, RTPSession) and one modified in OTcl (Session/RTP) which were derived from similar existing classes.
RTPAgent in NS2 did not seem to have full functionality according to IETF standard. So we had to modify it in a way to meet our needs and requirements. **VideoRTPAgent** (see Fig. 5) is our new dedicated RTP agent which was built on the basis of the existing RTPAgent implementation. The new RTP agent monitors the

transmission of video streams in terms of packet processing. In NS2 simulator, the former RTPAgent was only able to generate CBR traffic but this was not the case for our scenarios because we wanted to simulate layered video transmission. Thus we implemented a new C++ object namely **VideoTrafficTrace** (see Fig. 5) that works in co-operation with VideoRTPAgent and RTPSession classes. This new class was based on the existing class TrafficTrace. Every new object of this class corresponds to a new layer. Every layer is actually a replicated instance of the raw video stream encoded (in MPEG4 format using FFmpeg) in different bit rate. The encoded video stream is parsed and the trace file created is attached on a new VideoTrafficTrace object (see Fig. 5).
RTPSession class was created on the basis of the existing RTPSession (actually is a modified version of the corresponding former class). The former RTPSession class implemented only the RTP session management but our modified version needs to monitor the layered video transmission mechanism. For this reason the decision algorithm (see Fig. 1) is entirely embedded in RTPSession and all the VideoTrafficTrace instances must be attached on it as shown in Fig. 5.
The RTCP protocol monitors and controls RTP session. Moreover it provides feedback on the quality of data distribution using report packets. Full functionality of this protocol was not provided by NS2. Thus we extended RTCPAgent in order to implement **VideoRTCPAgent**. In our NS2 environment, the RTCP protocol functionality is divided between two objects: the VideoRTCPAgent class and the modified RTPSession class. The VideoRTCPAgent class implements the RTCP packet processing, while the RTPSession class implements the RTP session management as mentioned before. The RTCP protocol behaviour is based on the periodic transmission of control packets. The packet is prepared by the RTPSession in response to the VideoRTCPAgent request. The VideoRTCPAgent class simulates the transmission and reception of the RTCP packets.

2.1.3 Description and analysis of the new agents in NS2

As mentioned above four new classes were implemented in C++ namely VideoRTPAgent, VideoRTCPAgent, RTPSession and VideoTrafficTrace. Furthermore an OTcl object namely Session/RTP was designed in order to operate in cooperation with its mirror C++ class. Below we provide an in depth description of the aforementioned classes and object:

VideoRTPAgent

As shown in Fig. 5 VideoRTPAgent monitors the transmission of video streams in terms of packet processing. The lines shown in the following figure are executed prior the transmission of every packet.

```
p = allocpkt();
hdr_cmn::access(p)->size() = size_;
//RTP HEADER
hdr_rtp* rh = hdr_rtp::access(p);
rh->flags() = 0;
rh->seqno() = seqno_++;
rh->srcid() = session_ ? session_->srcid() : 0;
//ECN capable transport
hdr_flags* hf = hdr_flags::access(p);
hf->ect() = 1;
hdr_cmn::access(p)->timestamp() = (u_int32_t)(SAMPLERATE*local_time);
hdr_cmn::access(p)->sendtime_ = local_time;     // (smallko)
if(openfile!=0){
    hdr_cmn::access(p)->frame_pkt_id_ = id_++;
15     sprintf(buf, "%-16f id %-16d udp %-16d\n", local_time, hdr_cmn::access(p)->frame_pkt_id_, hdr_cmn::access(p)->size()-28);
16     fwrite(buf, strlen(buf), 1, BWFile);
17  }
18  target_->recv(p);
```

Fig. 6. Function sendmsg() of VideoRTPAgent.

Every packet received by VideoRTPAgent is being delivered to the RTPSession object for further processing of the data collected during the trip of the packet from the video streaming server to the video client (see Fig. 7).

```
1  void VideoRTPAgent::recv(Packet* p, Handler*)
2  {
3                  if (session_)
4                                  session_->recv(p, 0);
5                  else
6                                  Packet::free(p);
7  }
```

Fig. 7. Function recv() of VideoRTPAgent.

VideoRTCPAgent

This agent monitors and controls RTP session. Moreover it provides feedback on the quality of data distribution using report packets. In our NS2 environment, the RTCP protocol functionality is divided between the VideoRTCPAgent class and the modified RTPSession class.

The RTCP protocol behaviour is based on the periodic transmission of control packets. The following figure presents the function that dynamically calculates the interval between transmissions of RTCP packets according to RFC 3550. As you will see in line 9 of Fig. 8 the RTCP packet is prepared by the RTPSession in response to the VideoRTCPAgent request.

```
1  void VideoRTCPAgent::timeout(int)
2  {
3         if (running_) {
4                u_int32_t srcID[SRCNUM], extended[SRCNUM];
5                long int cumulative[SRCNUM];
6                int ecn[SRCNUM];
7                double meanE2E[SRCNUM];
8                //calculate the size of the RTCP packet which depends on the type of the packet (SR,RR)
9                size_ = session_->build_report(0,srcID,extended,cumulative,ecn,meanE2E); //a function of RTPSession is called
10               sendpkt(srcID,extended,cumulative,ecn,meanE2E);
11               double t = interval_;
12               if (random_)
13                      /* add some zero-mean white noise */
14                      t += interval_ * Random::uniform(-0.5, 0.5);
15               rtcp_timer_.resched(t);
16               Tcl::instance().evalf("%s rtcp_timeout", session_->name());
17        }
```

Fig. 8. Function timeout() of VideoRTCPAgent.

After the preparation of the packet by the RTPSession, VideoRTCPAgent implements the RTCP packet processing (see line 10 in Fig. 8) as shown in Fig. 9.

```
1  Packet* p = allocpkt();
2   hdr_rtcp* rh = hdr_rtcp::access(p);
3   /* Fill in srcid_ and seqno */
4   rh->seqno() = seqno_++;
5   rh->srcid() = session_->srcid();
6   for(int i=0; i<SRCNUM; i++) {
7   rh->remSrcid(i) = srcID[i];     //set the srcID
8              rh->cnpl(i) = cumulative[i];                    //set the cumulative number of packets received
9               rh->ehsnr(i) = extended[i];                    //set the extended highest sequence number received
```

```
        rh->time(i) = Scheduler::instance().clock();   //set the time the RTCP packet was sent
        rh->ecn(i) = ecn[i];                           //set the number of received ECN packets within the interval
        rh->meanE2E(i) = meanE2E[i];                   //set the mean end 2 end delay within the interval
  }
  target_->recv(p);
```

Fig. 9. Function sendpkt() of VideoRTCPAgent.

The VideoRTCPAgent class simulates the transmission and reception of the RTCP packets. Upon reception of every RTCP packet VideoRTCPAgent object informs the RTPSession object (see Fig. 10) about the statistics and measurements collected from the receiver or the core network within the predefined interval like the number of packets lost, the number of packets marked (due to ECN) and the mean end-to-end delay for every packet received from the video client within this interval (see Section 6.1.2 of the deliverable D2.2).

```
void VideoRTCPAgent::recv(Packet* p, Handler*)
{
    session_->recv_ctrl(p,info);
}
```

Fig. 10. Function recv() of VideoRTCPAgent.

RTPSession

RTPSession class implements the RTP session management and also monitors the layered video transmission mechanism. As you may see from the figure below, RTPAgent prepares the RTCP packet in response to VideoRTCPAgent call as presented in Fig. 8.

```
int RTPSession::build_report(int bye, u_int32_t *srcID, u_int32_t *extended, long int *cumulative, int *ecn, double *meanE2E)
{
    int nsrc = 0;
    int nrr = 0;
    int len = RTCP_HDRSIZE;
    int we_sent = 0;
    if (localsrc_->np() != last_np_) {
        last_np_ = localsrc_->np();
        we_sent = 1;
        len += RTCP_SR_SIZE;
    }
    //this version supports connections from one-to-one or one-to-many
    //i.e. there is only one RTPSource for each RTP receiver
    for (RTPSource* sp = allsrcs_; sp != 0; sp = sp->next) {
        ++nsrc;
        int received = sp->np() - sp->snp();
        if (received == 0) {
            continue;
        }
        sp->snp(sp->np());
        srcID[nrr] = sp->srcid();
        extended[nrr] = sp->ehsr();
        ecn[nrr] = sp->ecn();
        sp->ecn() = 0;  //initialize the number of received ECN packets
        cumulative[nrr] = sp->ehsr() - sp->np();
        meanE2E[nrr] = sp->totalE2E() / ((double)received);
        sp->totalE2E() = 0; //initialize the total end 2 end delay of the packets received within this RTCP interval
        len += RTCP_RR_SIZE;
        if (++nrr >= 31)
            break;
    }
    if (bye)
        len += build_bye();
    else
        len += build_sdes();
    Tcl::instance().evalf("%s adapt-timer %d %d %d", name(),
                    nsrc, nrr, we_sent);
    Tcl::instance().evalf("%s sample-size %d", name(), len);
    return (len);
}
```

Fig. 11. Function build_report of RTPSession.

Moreover this agent is responsible for the processing of the received RTP packets. RTPSession collects the data that are embedded in the header of the received RTP packet as depicted in Fig. 12(in response to VideoRTPAgent call shown in Fig. 7).

```
// process of received RTP packets
void RTPSession::recv(Packet* p, Handler*)
{
        hdr_cmn* hdr = hdr_cmn::access(p);          //get access to common header
        hdr_rtp *rh = hdr_rtp::access(p);
        hdr_flags *hf = hdr_flags::access(p);       //get access to ECN flag
        u_int32_t srcid = rh->srcid();
        //check whether i received again packets from this source
        RTPSource* s = lookup(srcid);
        //if not then add this source to the list of sources that this session receives RTP packets from
        if (s == 0) {
                Tcl& tcl = Tcl::instance();
                tcl.evalf("%s new-source %d", name(), srcid);
                s = (RTPSource*)TclObject::lookup(tcl.result());
        }
        //One packet received
        s->np(1);
        //Extended Highest Sequence number Received
        s->ehsr(rh->seqno());
        bytes_ += hdr->size()-28;
        //check if the ECN is enabled and ECN bit is set
        if(hf->ect() && hf->ce()) {
                s->ecn() += 1;
        }
        s->totalE2E() += Scheduler::instance().clock() - hdr->sendtime_;
       fprintf(tFile,"%-16f id %-16d udp %-16d\n", Scheduler::instance().clock(), hdr->frame_pkt_id_, hdr->size()-28);
        fprintf(mFile,"%-5d\t%-16f\n", rh->seqno(), Scheduler::instance().clock() - hdr->sendtime_);
        Packet::free(p);
}
```

Fig. 12. Function recv of RTPSession.

As mentioned before, RTPSession manipulates the received RTCP packets as well (Fig. 10). Upon reception of a single RTCP packet, RTPSession agent re-calculates some critical quantities like EHSR (extended highest sequence number received), CNPL (cumulative number of packet lost), ecnPV (explicit congestion notification percentage variation) and parameter a (used for the evaluation of the available bandwidth). For more information about these quantities please read sections 6.1.2 and 6.1.3 of deliverable D2.2.

```
//evaluate Loss Rate Per Second
cnpl_diff = rh->cnpl(i) - statistics.cnpl[rh->srcid()];
ehsnr_diff= rh->ehsnr(i) - statistics.ehsnr[rh->srcid()];
time_diff = rh->time(i) - statistics.time[rh->srcid()];
lf = (double)cnpl_diff/(double)ehsnr_diff;
lrps = lf/time_diff;
//evaluate percentage variation of ecn
if (statistics.ecn[rh->srcid()] != 0)
        ecnPV = double(((double)rh->ecn(i) / (double)statistics.ecn[rh->srcid()]) - 1.0);
else if ((statistics.ecn[rh->srcid()] == 0)&&(rh->ecn(i) != 0 )) ecnPV = 1.0;
        else ecnPV = 0.0;
if (ecnPV < -1.0) ecnPV = -1.0;
else if (ecnPV > 1.0) ecnPV = 1.0;
//evaluate percentage variation of mean end 2 end delay
if (statistics.meanE2E[rh->srcid()] != 0)
         meanE2EPV = double(((double)rh->meanE2E(i) / (double)statistics.meanE2E[rh->srcid()]) - 1.0);
```

```
else if ((statistics.meanE2E[rh->srcid()] == 0)&&(rh->meanE2E(i) != 0 )) meanE2EPV = 1.0;
else meanE2EPV = 0.0;
if (meanE2EPV < -1.0) meanE2EPV = -1.0;
else if (meanE2EPV > 1.0) meanE2EPV = 1.0;

printf("LF %f LRPS %f ECN %f MEANE2E %f\n",lf*100,lrps,ecnPV,meanE2EPV);
a=0.0;
videoIe(&ecnPV, &lrps, &a);
bandwidth = bandwidth*a;
```

Fig. 13. EHSR, CNPL, ecnPV and parameter a evaluations in RTPSession.

After the evaluation of these parameters, RTPSession implements the decision algorithm in order to infer the optimal number of layers of which overall transmission rate will not exceed the available bandwidth of the network path that was calculated before. This algorithm was presented in more detail in section 6.1.3 of the deliverable D2.2 and its source code is shown in Fig. 14.

```
/********************************
*      DECISION ALGORITHM          *
*********************************/
for(j=0; j<MAX_SCALE; j++)
        if((bandwidth<=layers[j])&&(knob==false))
        {
                if(j == rtpAgent_->getScale())
                break;
                knob = true;
                selected_layer = j;
        }
        else if(bandwidth<=layers[j])
        {
                knob = false;
                if(selected_layer < j)
                {
                        frame_index = traceAgent_[ rtpAgent_->getScale() ]->video_stop();
                        if(frame_index < 0)
                                break;
                        rtpAgent_->setScale(selected_layer);
                        traceAgent_[ rtpAgent_->getScale() ]->video_start(frame_index);
                        fprintf(info,"%d %d\n",rtpAgent_->getScale(),frame_index);
                        break;
                }
                else //if(selected_layer > j)
                {
                        temp = rtpAgent_->getScale();
                        frame_index = traceAgent_[ rtpAgent_->getScale() ]->video_stop();
                        if(frame_index < 0)
                                break;
                        rtpAgent_->setScale(j);
                        traceAgent_[ rtpAgent_->getScale() ]->video_start(frame_index);
                        if(temp != rtpAgent_->getScale())
                                fprintf(info,"%d %d\n",rtpAgent_->getScale(),frame_index);
                        break;
                }
        }
/********************************
*           END OF DECISION         *
*********************************/
```

Fig. 14. Decision Algorithm.

VideoTrafficTrace

In NS2 simulator, the former RTPAgent was only able to generate CBR traffic but this was not the case for our scenarios because we wanted to simulate layered video transmission. Thus we implemented a new C++ object namely **VideoTrafficTrace** (as shown in Fig. 5) that works in co-operation with VideoRTPAgent and RTPSession classes. This new class was based on the existing class TrafficTrace.

Every new object of this class corresponds to a new layer. Every layer is actually a replicated instance of the raw video stream encoded (in MPEG4 format using FFmpeg) in different bit rate. The encoded video stream is parsed and the trace file created is attached on a new VideoTrafficTrace object using the function shown in Fig. 15.

```
int videoTraceFile::setup()
{
        tracerec* t;
        struct stat buf;
        int i;
        unsigned long time, size, type, max;
        FILE *fp;
        if((fp = fopen(name_, "r")) == NULL) {
                printf("can't open file %s\n", name_);
                return -1;
        }
        nrec_ = 0;
        while (!feof(fp)){
                fscanf(fp, "%ld%ld%ld%ld", &time, &size, &type, &max);
                nrec_++;
        }
        nrec_=nrec_-2;
        printf("%d records\n", nrec_);
        rewind(fp);
        trace_ = new struct tracerec[nrec_];
        for (i = 0, t = trace_; i < nrec_; i++, t++){
                fscanf(fp, "%ld%ld%ld%ld", &time, &size, &type, &max);
                t->trec_time = time;
                t->trec_size = size;
                t->trec_type = type;
                t->trec_max = max;
        }
        return 0;
}
```

Fig. 15. Function videoTraceFile of VideoTrafficTrace.

Our application implements the transmission of layered video streams using several VideoTrafficTrace agents (each one corresponds to a unique layer) which are attached on the RTPSession agent. The latter agent monitors and controls the layered-like transmission of video streams and infers the number of layers sent through a single connection based on the decision algorithm.

2.1.4 Commands executed before and after conducting simulations

In this section we will see at a glance some steps that are followed before and after conducting simulations using the agents analyzed above.

1. Initially we have to encode the chosen raw video stream using FFmpeg tool [6]. The number of the MPEG4-encoded video streams depends on the number of the layers we intend to use. Each single stream will be encoded in a different bit rate using the command having the following generic syntax:

```
ffmpeg [[infile options][`-i' infile]]... {[outfile options] outfile}...
```

Here is the encoding procedure of a yuv sequence into MPEG4 data format. In this example, I use foreman_qcif.yuv as an example. This sequence has 400 frames.

```
ffmpeg -i foreman_qcif.yuv -b 128 -r 30 -s qcif -vcodec mpeg4 –bf 2 -4mv  -g 12
foreman_128kbps.m4v
```

2. We use MP4.exe (from Evalvid package) to record the sender's trace file (st). Each frame will be fragmented into 1000 bytes for transmission. (Maximun packet length will be 1028 bytes, including IP header (20bytes) and UDP header (8bytes).)

```
mp4.exe –send 224.1.2.3 5555 1000 foreman_qcif.m4v > st
```

The trace file that was created will be attached to a VideoTrafficTrace agent in the tcl script file that describes the scenario. Needless to say that the above procedure is being followed for every single encoded video stream that corresponds to a different layer.
3. We execute the NS2 scenario written in tcl scripting language. After simulation, NS2 will create a number of files concerning the packets sent and received. The half of these files is to record the sending time of each packet while the rest of them are used to record the received time of each packet.
4. Finally we execute a script file namely doIt that takes the aforementioned files as input in order to construct the received video stream that consists of frames correspond to different layers. Additionally this file evaluates the psnr value of the received video stream. One can simply execute this command as shown:

```
./doIt 1
```

2.2 RAF

Receiver-driven Adaptive Feedback algorithm was also implemented in NS2 simulator. This algorithm was based on the Goddard Streaming Media framework [8] which was previously used to simulate the transmission of video streams over heterogeneous networks. The rest of this section is organized as follows: Section 2.2.1 deals with operating system and required packages, section 2.2.1 investigates the Goddard Streaming Media system whereas section 2.2.2 presents useful commands that are executed before and after conducting simulations regarding this algorithm.

2.2.1 Operating System and Required Packages

Before getting into the insights of our source code let us mention the operating system options and requirements needed prior the installation of our simulation modules. All the simulations were conducted in a linux-based Fedora Core 2 [2] operating system. All the required packages are shown in the table below.

Required Packages	Description
NS2 [1]	NS is a discrete event simulator targeted at networking research. Ns provides substantial support for simulation of many protocols, routing, and multicast protocols over wired and wireless (local and satellite) networks. All our simulations were conducted under this framework.
Evalvid in NS2 [3], [4] [5]	EvalVid [5] is a complete framework and tool-set for evaluation of the quality of video transmitted over a real or simulated communication network. It was integrated into NS2 as shown in [3] and [4]. We used it for end-to-end delay, end-to-end jitter and PSNR evaluations in NS2-based simulations. The commands we used for

	these evaluations will be shown below.
MPEG4 Encoder/Decoder from [9]	This is an MPEG4 encoder/decoder that was implemented by the National Chiao-Tung University in Taiwan. It takes as input a dedicated file that contains all the parameters involved in the encoding/decoding procedure.
Goddard Streaming Media System [8]	An NS2-based streaming system (client and server) which was designed based on the behaviours of Real Networks streaming media and Windows Stream media.

Table 2. Operating System and required packages used in RAF.

2.2.2 Goddard Streaming Media System

Worcester Polytechnic Institute designed and implemented in NS a streaming system (client and server) called Goddard. Goddard is designed based on the behaviours of Real Networks streaming media and Windows Stream media. The Goddard streaming client and server use packet-pairs to estimate the bottleneck capacity and select an appropriate media encoding level before streaming. During streaming, Goddard client and server reselect the media to stream (i.e., perform media scaling) in response to network packet losses or re-buffering events that occur when the client playout buffer empties. Goddard also simulates frame playout of the received media at the client, allowing frame rate and jitter to be measured for performance evaluation.
As in commercial systems, the Goddard server supports multiple levels of encoded media that are configured by giving the frame size and the frame rate for each scale level. In addition, the Goddard server has an option for setting the maximum fragment size for fragmenting large media frames before transmission. Typically, the maximum fragment size would be set to the maximum transmission unit (MTU) of the underlying network. The Goddard client, also called Gplayer for Goddard Player, has the configuration parameters shown in Table 3.

Parameter	Default value	Description
pkp_timeout_interval	2 seconds	Packet-pair timeout interval
buf_factor	1.5	Buffering rate factor
play_buf_thresh	5 seconds	Threshold to start playout
loss_monitor_interval	5 seconds	Loss monitoring interval
downscale_frame_loss_rate	0.05	Down-scale frame loss rate
upscale_interval	60 seconds	Up-scale decision interval
upscale_frame_loss_rate	0.01	Up-scale frame loss rate
upscale_limit_time_factor	3	Up-scale limit time factor

Table 3. Goddard Client (Gplayer) parameters with default values.

The default parameter values are set based on observations. Similar to commercial streaming systems, the Goddard client and server use three communication channels for a streaming session: a control cannel using a TCP connection, a UDP packet-pair channel, and a media streaming channel that can be TCP, UDP or MTP. When setting up a streaming session, the Goddard server sends the list of supported media scale levels to the Gplayer using the control channel. Then, Gplayer sets a timer with pkp timeout interval and requests the server to send a pair of UDP packets to estimate the capacity of the network path. If any one of the packet-pairs is lost, the packet-pair timer expires and Gplayer will send a request for another packet-pair to the Goddard server. On successful reception of a packet-pair, the capacity of the network path is computed by dividing the packet size by the dispersion. Then, Gplayer selects the largest media scale level with a bit rate less than the computed capacity and notifies the server. Gplayer also notifies the server of the buf factor before starting streaming to determine how much the server should increase the transmission rate during media buffering periods.

The Goddard client and server operate in two modes: buffering or streaming. During buffering, the Goddard server transmits the chosen media frames at the rate of buf factor times the streaming bitrate, where buf factor used for commercial streams typically ranges from 1.5 to 4. Gplayer maintains a media playout buffer and a playout threshold (play buf thresh). When the Goddard server starts media transmission in buffering mode, the Gplayer buffers the frames received in the media buffer. When the media buffer size (given in playout time) reaches the play buf thresh, Gplayer tells the server to switch to streaming mode and starts playing the media according to the timing described for the current media scale level. If the media buffer runs out of frames, Gplayer stops media playout and switches back to buffering mode. At this time, Gplayer re-selects the largest media scale level with a bitrate less than the average received throughput for the previous control interval. Then, Gplayer tells the server to transmit frames of the new scale level at the buffering rate, that is the streaming bitrate times buf factor. When a Goddard streaming session uses UDP or MTP for the media channel, Gplayer can also use frame loss information to make media scaling decisions. In this case, Gplayer monitors the frame loss rate each time it receives a media frame. When it is at least loss monitor interval since the last scale adjustment decision was made and the frame loss rate is greater than downscale frame loss rate, Gplayer scales the media down one level if the current scale level is not already at the minimum. If the current scale is at the minimum, Gplayer maintains the current scale level. The default value for downscale frame loss rate is set to 0.05. Gplayer also makes decisions to scale the media up to a higher level, but does so slowly and gently. Gplayer increases the scale level by one if the frame loss rate of the stream is less than upscale frame loss rate for upscale interval since the last time scaling decision was made and the bitrate of the stream after the increase is less than or equal to the estimated network capacity. Also, in order to reduce the chance of playout interruption, Gplayer limits scaling up to one below the last scale level that caused media re-buffering. This limit on scaling up is heuristically relaxed by one scale level if the stream maintains good quality (i.e., no scale down events) for upscale limit time factor times the upscale interval. The default for upscale interval is set to 60 seconds, a value from the observed range (30 to 90 seconds) during the streaming measurement studies. Thus, Goddard simulates a realistic streaming video application that performs media scaling, buffering and playout. Implementations of support for video frame dependencies, selective retransmission or other media repair mechanisms were left as future work.

In the initial version of Goddard System every layer was simulated on the basis of a different CBR traffic pattern. In our version every layer corresponds to a different VBR traffic pattern and for this purpose every layer is simulated using real video traces.

2.2.3 Implementation of the proposed algorithm

At the beginning our algorithm requires some initial values to be assigned to the different parameters mentioned above. Thus the user has to define these values prior conducting simulations. A user can set the number of layers regarding each scenario and define the bit rate of each layer (namely max_scale_ and bitrate_#_ respectively).

To the best of our knowledge, the algorithm can operate in the absence of user-defined values by setting some default values for every parameter.
At the beginning of every simulation the video streaming server sends to the client a list of all supported layers. This enables every client to decide and change the number of layers that are transmitted from the streaming server upon request according to the conditions of the network path. For this reason a decision algorithm is implemented in client side that periodically track the changes in the available bandwidth of the network path. The algorithm evaluates the available bandwidth and decides the number of frames per second, the width and height of each frame, the quality of each frame (in terms of bit per pixel) of the video stream (according to Table 5 of the deliverable D2.2) in order to meet user's needs and requirements (comply with the initial values of parameters assigned by user a priori).
The decision algorithm determines the maximum possible bit rate and then checks the initial values assigned by the user. Values that have been initially set are given higher priority over non-defined parameters. First of all, the algorithm examines the frame width parameter (frm_width). It is beyond any doubt that this parameter plays a crucial role for the user perceived quality of the received video stream. The value of this parameter poses a critical constraint in the system especially if we consider users having handheld devices with small displays. Having defined the frame width it is now possible to evaluate frame height depending on the aspect ratio.
In case the frm_width is not defined by the user, the algorithm assumes that there is no constraint posed for this parameter. Therefore the next parameter that is being examined concerns the quality of the transmitted video stream namely bit per pixel (bpp). If this parameter is not set a priori our algorithm will set it to 0.225 which corresponds to high quality. This can be done because there is no constraint with respect to the frame width and if have no constraint with respect to the bit rate too then we will be able to provide high quality video stream. The next step is to determine the aspect ratio. If no value has been set for this parameter then the algorithm uses a default value of 4:3 or 1.33. Having defined all these values (bit rate, bpp, fps, aspect ratio) our algorithm is able to calculate both the frame width and height using the equations in section 6.2.2 of the deliverable D2.2.
Moreover our algorithm is striving to provide high quality video streaming in the presence of high error rates by calibrating the values of the different parameters.
In addition the bit rate is calculated using frame width, frame height, bit per pixel, and frames per second values taken by Table 5 of D2.2. If the value of bit rate is lower than the maximum possible transmission rate (available bandwidth) that was defined at the beginning then all these values are sent back to the video streaming server which acts accordingly by adding or dropping layers.
On the other hand if the calculated bit rate is higher than the available bandwidth, the algorithm has to re-consider the values of some parameters. The next best thing that the algorithm has to do is to reduce the dimensions of the frame while maintaining the same frame quality and the same transmission rate defined by the user. So it is possible to re-calculate the bit rate and check again whether is lower than the available bandwidth or not. We would like to point out that the size of the frame depends on the aspect ratio while there is a lower limit for both dimensions.
All the measures are taken into consideration whenever the dimensions of the frame are not defined by the user. But if we consider that these parameters are initially set by the user then are given higher priority amongst all the other. All the other parameters can be altered by the decision algorithm but the frame width and frame height are kept constant until all the other parameters reach their lower value while

the transmission of the video stream is still infeasible. This is only the case where the values of these parameters can be altered. Even in this rare scenario there is a lower limit for these values. The smaller dimensions of a video frame are 80x60 and 80x45 regarding aspect ratio of 4:3 and 16:9 respectively.
In the previous paragraphs we presented and analysed all the parameters that are taken into consideration in the implementation of this adaptive algorithm. Given the large number of combinations amongst the different values of the aforementioned parameters we will provide four different cases so as to present the algorithm's respond to the various input scenarios.
We would like to point out that the parameter max_possible_bitrate takes the smaller values among the variables max_bitrate, avail_bandwidth and the constant parameter MAX_POSSIBLE. The latter parameter corresponds to the bit rate of the upper layer (i.e. if the upper layer has mean bit rate of 768Kbps the value of MAX_POSSIBLE is 768000). For example consider a user connected to the network with a modem of 56Kbps, the video streaming server can support 5 layers with a maximum mean bit rate of 768 Kbps while the available bandwidth of the network path is 1Mbps, then the video stream will be transmitted using the smaller value namely 56Kbps.

CASE A

Frame Dimension (frm_width)	Bandwidth of the user connection (max_bitrate)	Frame Quality (bpp_quality)	Frame Rate (fps)	Available Bandwidth (avail_bandwidth)
Given	Given	Given	If not given by the user then the default value is 30 or 25 if the colour standard is NTSC or PAL respectively.	Given
Given	Given	Given		Not Given
Given	Not Given	Given		Given
Given	Not Given	Given		Not Given

Table 4. Combination of different parameters when user provides specific frame dimensions and frame quality.

Based on the chosen max_possible_bitrate the algorithm examines if the transmission of a video stream is feasible when frame dimension (frm_width), frame quality (bpp_quality) and number of frames per second (fps) are given by the user. If the transmission is not feasible due to limited bandwidth constraint then the parameter bpp_quality is being reduced down to a lower limit (see Table 5 in D2.2). The algorithm strives to transmit a video stream in the lowest but satisfied quality while the dimensions of the frames are kept constant. If this is not feasible, it is apparent tha the frame dimensions have to be reduced down to a lower bound as mentioned before while user perceived quality is maintained in acceptable levels.

CASE B

Frame Dimension (frm_width)	Bandwidth of the user connection (max_bitrate)	Frame Quality (bpp_quality)	Frame Rate (fps)	Available Bandwidth (avail_bandwidth)
Given	Given	Not Given	If not given by the user then the default value is 30 or 25 if the colour standard is NTSC or PAL respectively.	Given
Given	Given	Not Given		Not Given
Given	Not Given	Not Given		Given
Given	Not Given	Not Given		Not Given

Table 5. Combination of different parameters when user provides specific frame dimensions.

Based on the chosen max_possible_bitrate, the algorithm examines if the frame having dimensions given by frm_width is feasible to have the same quality as the first appearance of the fps parameter in Table 5 of D2.2 with the corresponding bpp_quality. If the transmission is still infeasible then the algorithm reduces the quality down to a lower bound of the Table 5 of D2.2 where is the quality is sustainable. If the transmission is again infeasible then the dimensions of the frame have to be reduced until the frame width reaches 80 pixels which is the lowest feasible value in order to provide satisfactory user perceived quality.

CASE C

Frame Dimension (frm_width)	Bandwidth of the user connection (max_bitrate)	Frame Quality (bpp_quality)	Frame Rate (fps)	Available Bandwidth (avail_bandwidth)
Not Given	Given	Given	If not given by the user then the default value is 30 or 25 if the colour standard is NTSC or PAL respectively.	Given
Not Given	Given	Given		Not Given
Not Given	Not Given	Given		Given
Not Given	Not Given	Given		Not Given

Table 6. Combination of different parameters when user provides specific frame quality.

Based on the chosen max_possible_bitrate parameter our algorithm calculates the dimensions of the frame using the quality parameter bpp_quality and the fps parameter so that the overall bit rate is lower than the pre-defined max_possible_bitrate. If it’s higher then our algorithm tries to make calculations without altering the requested video quality. In this case it decreases the dimensions of the frame down to the lower bound (80 pixels) based on the user-defined quality. If the transmission is not feasible the quality parameter has to be re-considered (quality can also be decreased down to a lower bound given by Table 5 in D2.2).

CASE D

Frame Dimension (frm_width)	Bandwidth of the user connection (max_bitrate)	Frame Quality (bpp_quality)	Frame Rate (fps)	Available Bandwidth (avail_bandwidth)
Not Given	Given	Not Given	If not given by the user then the default value is 30 or 25 if the colour standard is NTSC or PAL respectively.	Given
Not Given	Given	Not Given		Not Given
Not Given	Not Given	Not Given		Given
Not Given	Not Given	Not Given		Not Given

Table 7. Combination of different parameters when user does not provide neither specific frame dimensions nor frame quality.

Based on the chosen max_possible_bitrate our algorithm calculates the dimensions of the frame using the quality parameter (bpp) which corresponds to the first appearance of fps parameter (see Table 5, D2.2), so that the overall bit rate is lower that the max_possible_bitrate. If the overall bit rate is higher than max_possible_bitrate the algorithm maintains the chosen quality and reduces the dimesions (down to 80 pixels). If the video stream cannot be sent then we have to decrease the quality based on its lower bound.

In all four cases mentioned above if we reach the lower bounds with respect to frame width and frame quality then the algorithm deduces than the transmission of the video stream is not feasible because the stream cannot be sent with satisfactory user perceived quality. The algorithm re-calculates the transmission parameters as soon as the parameter max_possible_bitrate changes.

2.2.4 Commands executed before and after conducting simulations

Evalvid framework presented in sections 2.1.1 and 2.1.2 is also used here for quality evaluations. Before conducting simulations we have to create a designated file namely example.par which will contain the following parameters and their initially assigned values. This file will have the following syntax:

```
Source.Width
Source.Height
Source.LastFrame
Source.FilePrefix
Source.FrameRate[0]
RateControl.BitsPerSecond[0]
Scalability.Spatial.Width
Scalability.Spatial.Height
```

After completing this step we adopt the following steps:

1. The file example.par is used in conjunction with mpeg4encoder.exe application [9] that takes all the aforementioned parameters into consideration in order to compress the raw video stream as shown below:

```
mpeg4encoder.exe example.par
```

2. We use MP4.exe (from Evalvid package) to record the sender’s trace file (st) as we did in ADIVIS algorithm. Each frame will be fragmented into 1000 bytes for transmission. (Maximun packet length will be 1028 bytes, including IP header (20bytes) and UDP header (8bytes).)

```
mp4.exe –send 224.1.2.3 5555 1000 foreman_qcif.cmp > st
```

3. We execute the NS2 scenario written in tcl scripting language. After simulation, NS2 will create two files, sd_be and rd_be. The file sd_be is to record the sending time of each packet while the file rd_be is used to record the received time of each packet.

4. Using the tool et.exe (from Evalvid) to generate the received video (err.cmp):

```
et.exe sd_be rd_be st foreman_qcif.cmp err_be.cmp 1
```

5. Decode the received video to yuv format. I also put the status of decoding into a file df_be.

```
mpeg4decoder.exe err_be.cmp err_be  176  144  >  df_be
```

6. Use the program myfixyuv.exe to fix the decoded yuv sequence.

```
myfixyuv.exe  df_be  qcif  400  err_be.yuv  myfix_be.yuv
```

7. In order to evaluate the objective quality of service of the received video stream we have to compute the PSNR using the psnr.exe (from Evalvid).

```
psnr.exe  176  144  420  foreman_qcif.yuv  myfix_be.yuv > psnr_myfix_be
```

3. Conclusions

In this deliverable we analyzed the simulation models that will be used for the evaluation the performance of each algorithm. Simulation models are based on the open source simulator ns2.
ADIVIS evaluations are based on an enhanced ns2 node model presented in Fig. 5 that includes a video RTP agent, a video RTCP agent and an enhanced RTP Session module. Moreover we tried to attach real video traces in the simulation model so as to conduct more realistic scenarios. ADIVIS QoS assessment would not be possible without the Evalvid Framework which is a complete framework and tool-set for evaluation of the quality of video transmitted over a real simulated communication network.
RAF evaluations are based on a streaming system (client and server) called Goddard which was implemented in Worcester Polytechnic Institute. Goddard is designed

based on the behaviours of Real Networks streaming media and Windows Stream media. In the initial version of Goddard System every layer was simulated on the basis of a different CBR traffic pattern. In our version every layer corresponds to a different VBR traffic pattern and for this purpose every layer is simulated using real video traces.

All in all a combined wired and wireless network will be simulated and the proposed algorithms will compose a fully adaptive system which will be tested for its ability to support real time multimedia applications in various realistic or extreme scenarios. The results of these scenarios will be presented and analyzed in Deliverable 3.3.

References

[1] Network Simulator – ns-2 site. http://www.isi.edu/nsnam/ns/.

[2] Fedora Project site. http://fedora.redhat.com/.

[3] Evalvid in NS2. http://140.116.72.80/~smallko/ns2/Evalvid_in_NS2.rar.

[4] C.-H. Ke, C.-H. Lin, C.-K. Shieh, W.-S. Hwang, "A Novel Realistic Simulation Tool for Video Transmission over Wireless Network," The IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC2006), June 5-7, 2006, Taichung, Taiwan.

[5] J. Klaue, B. Rathke and A. Wolish, "Evalvid – A Framework for Video Transmission and Quality Evaluation," http://www.tkn.tu-berlin.de/publications/papers/evalvid.pdf.

[6] FFmpeg Multimedia System site. http://ffmpeg.mplayerhq.hu/.

[7] J. Klaue. Evalvid – http://www.tkn.tu-berlin.de/research/evalvid/fw.html.

[8] Worcester Polytechnic Institute. "Research related to performance of networks, specifically congestion control and multimedia systems," http://perform.wpi.edu

[9] MPEG4encoder site. http://www.megaera.ee.nctu.edu.tw/mpeg/ Department of Electronics Engineering National Chiao-Tung University, Taiwan.