Real-Time Video Streaming in Multi-hop Wireless Ad Hoc Networks: Simulator manual

Yaniv Fais

Supervisors: Prof. Guy Even and Moti Medina

Table of Contents

List of figures

List of tables

I. Abstract

The problem of transmitting real time video stream over ad-hoc wireless network poses a special challenge due to the demanding requirement of the data and the special network model.

The requirement to satisfy for real time video streaming is large throughput of data and minimal and stable end-to-end delay, this characteristic of the data is required since transmission of video requires large bandwidth even with modern encoders due to the growing demand for high quality and the fact that complex encoding/decoding can't be done in real-time.

The transmission of real time video, for usage such as video conference or live events broadcast for example, adds the requirement of minimal end-to-end delay which is required so that the transmission over the network would be negligible in order to create the effect of real-time on both ends.

The medium of an ad-hoc wireless network is mostly challenging due to the fact that though these networks are more common today they still have relatively low bandwidth and low transmission distance compared to wide-area networks or cellular base-stations, this requires special handling for multi-hop transmission and flow-control.

The work in this project is part of a research group designed to handle such scenario, this document described the simulator and the flow control which are part of the overall solution to the shown problem.

II. Introduction

1. Wireless network model

We consider a problem of multiple nodes on a surface carrying each a single transceiver able to either listen in one of several known frequencies (channels) or transmit in one channel to neighboring nodes in a known protocol (See section II/3).

In the above network we consider a set of demands for video-streams which is composed by several triplets which are the sending node (camera node), destination node and the requested rate for each stream, each of the nodes has a location *(x,y)* which may change over time in the surface.

A direct communication between two nodes in this network is possible if their distance is smaller than the transmission range for these nodes (subject to the communication protocol, See section II/3). An indirect (multi-hop) communication between two nodes is possible if there exists a path of nodes with direct communication between the source and destination.

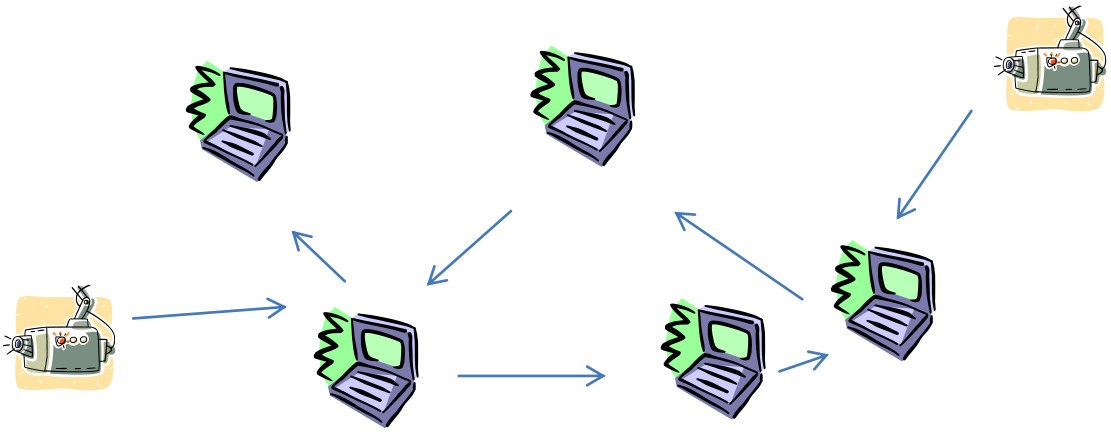


Figure II-1 Ad-Hoc Network model

Two nodes transmitting in the same channel interfere with each other and can cause error in packet receive, the protocol used (See section II/3) attempts to minimize the collision using control packets from both sender and receiver by setting only one transmitting node (this means interference is caused by transceivers from both sender and receiver) however errors may still occur due to cumulative interference from nodes, we can consider two models for such interference:

1. Graph model:

   In this model we consider two radii *r(v)* and *R(v)* which are associated with each transceiver , these two radii denote the *transmission range* and *interference range* of the node *v* respectively, the values of these radii is proportional to the transmission power of the transceiver in node *v* .

   A link *e = (v,u)* exists if the distance between the nodes *d(v,u)* is smaller than the transmission radius, *d(v,u)<r(v)* .

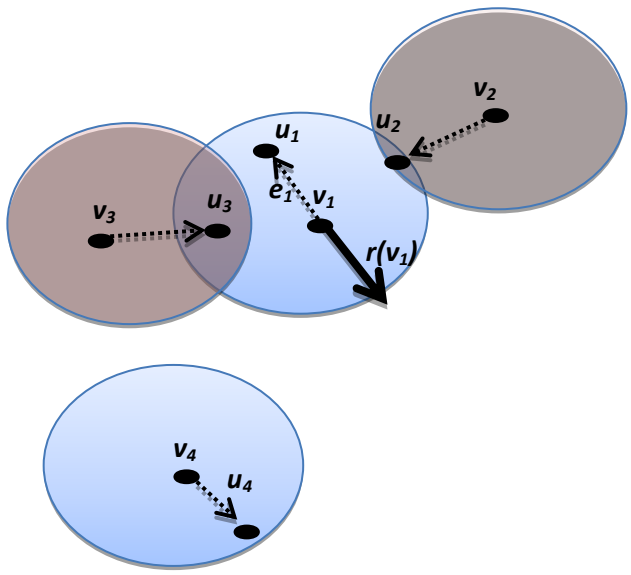


Figure II-2 Network disk graph model

The above figure illustrates a graph model of transmission from four nodes (v_1-v_4) to four receivers (u_1-u_4), for each transmitting node *v* a circle with radius *r(v)* shows the *communication range* of this node. We should note that when using the Wi-Fi protocol (See section II/3) both source and destination transmit packets thus they need to be able to communicate.

We say that node u can receive a message from v if $d(v,u)<r(v)$ and every other node x that transmits in the same time satisfies $d(x,v)>R$.

An interference is defined between two links (v_1,u_1) and (v_2,u_2) if $\min\{d(u_1,u_2), d(u_1,v_1), d(v_1,u_2), d(v_1,v_2)\} < R$.

The following figure shows the transmission between node v_1 along edge e_1 which is inside the transmission range $r(v_1)$, the edges e_2 and e_3 are inside the interference range $R(v_1)$ and thus they can't be used in the same time e_1 is used.

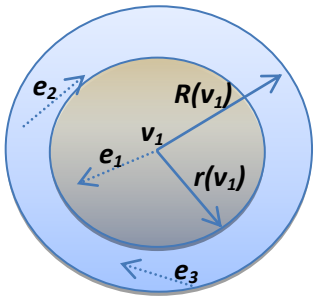


Figure II-3 Graph Interference model

2. Signal-to-Interference-plus-Noise-Ratio (SINR) model: ((Kumar, 2000), (T. Moscibroda, 2006)) , (Avin, Emek, Kantor, Lotker, Peleg, & Roditty, 2008) This model (a.k.a. physical interference model) which is considered more accurate than the graph model defines a successful communication between node u and v according to the SINR of the message which is received in u, considering that the node set S_t transmit in the same time, the SINR is defined:

$$SINR(\mathrm{u,v,S_t}) = \frac{P/d(u,v)^{\propto}}{N + \sum_{x \in St \setminus u} P/d(x,v)^{\propto}}$$

Where P denotes the transmission power and $\propto$ the path loss exponent.

We now say that a message is transmitted successfully if $\mathrm{SINR}(u,v,S_t) > \beta$ where β is the minimum SINR threshold for the communication scheme.

The full definition for the interference model which is used is more subtle (See full article (Even, Fais, Medina, Shahar, & Zadorojniy, 2011)) however it is important to note the major difference between these two models, especially in relation to the usage of the graph model in the scheduling algorithm (See section II/4) and the usage of the SINR model in the simulator (See section II/3).

3. Wireless protocol

The 802.11g IEEE protocol (See (IEEE, 2003)) is designed for Carrier-Sense-Multiple-Access with Collision-Avoidance (CSMA/CA) networks where each node "listens" to the channel prior to attempting to send a packet such that it doesn't interfere with existing usage of the medium, if the medium is "busy" then a random delay (commonly exponentially growing) is used until a next attempt to transmit is done.

In order to avoid problems caused by "hidden node" which are outside the range of one of the two nodes which desire to communicate special control packets have been added:

Request-To-Send (RTS) control packet is transmitted by the source whenever it desires to send a new frame

Clear-To-Send (CTS) packet is being transmitted by the destination node if it isn't busy and it senses that the channel is not busy

When the source receives the CTS packet it sends the payload packet to the destination

When the destination has completed receiving the payload packet it sends to the source an acknowledge (ACK) packet

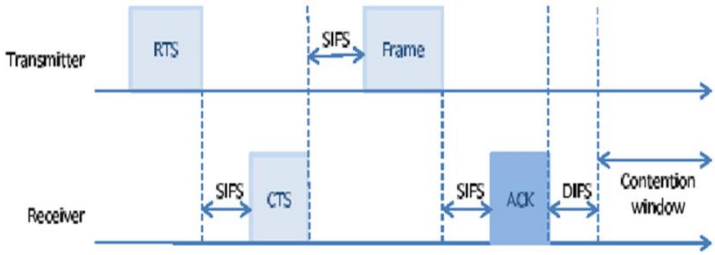


Figure II-4 802.11 Protocol timing diagram

Source (Zadorojniy, Even, & Shahar, Frequency and Time Slot Assignment Algorithm, 2009)

The above protocol, using a simple global (not per neighbor) state machine is able to safely perform communication in such network of un-synchronized and non-collaborating nodes and can avoid deadlock and collision in the network.

This protocol however causes a severe degradation on actual performance compared to the theoretical rate enabled by the coding, moreover this protocol doesn't deploy any priority on message transmission and thus doesn't guarantees fairness or any quality-of-service on either throughput or end-to-end delay.

The 802.11g physical layer allows transmission in 8 Modulation Coding Scheme (MCS), each MCS performs different bit to symbol coding in the frequency domain and can achieve different bit-rate from 6Mbps to 54 Mbps, the payload packet can be send in any such MCS however higher bit-rate suffers from higher coding errors and thus can sustain only higher signal-to-noise (SNR) ratio. If we consider the case of fixed communication parameters such as power, noise and etc. then the SNR is lower as the distance between the sender and receiver increases, this means in greater distance only small bit-rate (and small MCS) can be used, this fact is being used by the scheduling algorithm (See sections VI/4 , VI/5).

4. Scheduling algorithm

In order to accommodate the requirements a scheduling algorithms has been developed by the research group, this algorithm accepts as input the nodes $\{V\}_{i=1,N}$ and their surface locations x_i, y_i and a set of requests $\{r_k\}_{k=1,K}$ triplets (source,destination,demand) (See section VI / 7) .

The algorithm outputs a scheduling table which consists of a list of rows where each row has the following columns (See section VI/ 8):

Time-Slot , Stream Number , Sender, Receiver , MCS , Frequency channel , Data Flow

This table is a valid schedule in terms that it has no interfering edges according to the graph model as defined above, moreover this algorithm performs routing along multi-hop paths and provides valid scheduling for all the nodes such that transmission of requests would be satisfied and attempts to achieve low end-to-end.

The algorithm and its tradeoffs are defined in the full article (See (Even, Fais, Medina, Shahar, & Zadorojniy, 2011)) however it is important to note that this algorithm uses the graph interference model as an approximation for the SINR model , this makes the checking of the validity of this algorithm in a real environment challenging since the scenario may make some instance of the problems work as expected and satisfy the video stream transmission request as the algorithm outputs properly and other instances may fail to do so due to differences between the different models.

5. Video coding

Video coding aims at achieving the following goals:

- Reduce data size of video (compression) but allow a balance between quality and compression
- Resilience to errors in order to allow transmission by unreliable networks
- Synchronization of audio/video and data flow
- Tractable computational complexity
- User interaction such as time search and etc.
- Enablement of different input quality (frame resolution, frame per second rate and etc.)

Compression of the video data is extremely important when transmitting over wireless media, as the data rate is much higher than can be transmitted, the following table summarizes the uncompressed data rate for a video camera at a given frame resolution and rate:

Table 1 Camera data rate

Source **(Scoblete, A Beginner's Guide to Camcorder Bit Rates)**

Camera source type	*Data rate [Mbits/sec]*
320x240 (VGA/4) at 20 frames/sec	36.8
352x288 (CIF) at 30 frames/sec	73
640x480 (VGA) at 30 frames/sec	221.18
1920x1080 (Full-HD) at 30 frames/sec	1232

The following table lists the data rate achieved by several wireless standards:

Table 2 Network speed

Source (zytrax)

Network protocol	*Actual data rate (average) [Mbits/sec]*
Wi-Fi 802.11g	26
GPRS (2G)	0.03
WCDMA (3G)	0.384
ADSL2 (non-wireless)	12

It is obvious from the above tables that good compression must be achieved in order to transmit video over a wireless network and while the rate achieved by wireless networks is constantly increasing so are the common video quality which are often designed for bit rates for local networks and mediums such as DVD.

In order to achieve these goals many different schemes of video coding have been developed, most notably by the Moving Pictures Experts Group (MPEG).

This groups has developed many different layers of algorithms aimed to achieve the above, the coding is composed of an encoder which takes the raw data and

compresses the data often in a lossy way (i.e. not storing all the original data) such that a simple decoder would be able to display the video and achieve good quality.

The encoding has various coding features (also known as layers or profiles) , where each such feature adds different algorithm which can further increase the quality or compression rate. While deploying all these features can achieve the best results it may also be compute intense therefore the encoder is free to use only parts of these layers. This is especially important for real-time video where deploying a full-feature encoder may be difficult to achieve or may increase end-to-end delay due to the complexity of the encoder.

The encoder used by MPEG uses several important characteristics in-order to perform the compressions:

- Encoding of pictures is done similar to that of JPEG format by splitting the frames into smaller 2D blocks and using the smaller entropy in each 2D-DCT of the YUV colors of the block for coding.
- Encoding of time domain is done by detecting moving vectors of smaller blocks in respect to a reference frame.

The video coding is then split into several frame types which hold the encoded data:

- I-frame (intra-coded): coded without reference to any other frame (picture)
- P-frame (predictive-coded): coded with reference to previous frame (either I or P)
- B-frame (Bi-directional predictive coding): coded in reference to either past or future I or P frames

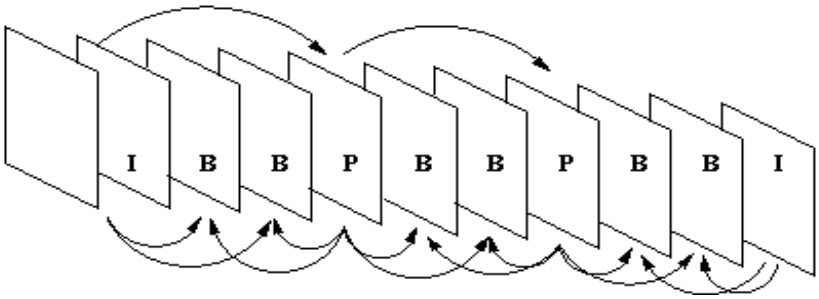


Figure II-5 MPEG Group Of Pictures

source (John G., Wai- tian, & Susie J., 2002)

Such consecutive frames are then grouped into a group of pictures (GOP) which can be coded without reference to other frames, GOP is usually 12-15 frames where one can perform random access to the first I-frame of the group.

This partitioning into groups allows easy navigation in time and is mostly effective in mediums such as DVD , when transmitting real-time video over network we use only frame references back in time in order not to add delay in the encoder and decoder.

Audio is commonly coded in a MPEG1 layer 3 (MP3) format which compresses the sampled audio data by storing only frequencies which are hearable by humans.

The video and audio are then multiplexed and synchronized in a container format.

The importance of the coding into groups of pictures is important since when transmitting over a wireless network it is required to partition the data into packets however some of these packets may be lost during the transmission, by using the

group and the reference frames the decoder may be able to compensate such errors and still continue to decode the video and provide good quality.

Newer coding standards are being developed, most of them are based on the above concepts but add additional control packets or enhanced entropy encoding which can increase the compression further but the concept of motion detection and partitioning to packets which are mostly important for the network transmission remains.

One of the most important concepts of real-time encoding however is the ability to change the rate of transmission through reduction of the quality of the video through reduction of the frame rate or reduction of the bit rate of the frames (number of colors, motion sensitivity and etc.), this allows an adaptive system according to available bandwidth.

6. Video network protocols

Transmission of data over network requires several layers of protocols for handing the communication from the physical layer to the application layer, one of the most crucial layer is the transmission layer.

A very common protocol used in the transmission layer is the Transmission Control Protocol (TCP), TCP is used very widely over the internet as an end-to-end protocol in various application, this protocol provides guaranteed transmission with low overhead bandwidth however this protocol isn't suitable for real-time video as the quality of service or the end-to-end isn't guaranteed and is often too high for video streaming. While even common services for video transmission use TCP/IP , this usage however is not for streaming and it simply attempts to send the file to the destination where it is buffered and displayed when suitable, while this method works well for low-bandwidth non-real time video over high bandwidth (LAN/WAN) network , it is not suitable for real time video streaming in wireless network.

For the purpose of video streaming protocols such as User Datagram Protocol (UDP) are used, this protocol employs a simple message which contains only the source, destination and the data (+checksum) and doesn't require handshake, as such it is less reliable but offers good end-to-end delay.

One of the most common application layer protocols for real time video streaming is Real-time Transmission Protocol (RTP) (IETF), this protocol is designed for low overhead end-to-end, real-time carrying of video and audio data. This protocol defines the packet format, the most important fields are the following:

Payload type - type of data carries in the data field according to specific types defined, for example H264 or MP3 .

Sequence number – number which identified a packet in the stream, increases by 1 for every packet which is send from the source

Timestamp – a number which is used by the receiver in order to play the data in a specified interval (the clock resolution is defined according to the profile of application), for example this number can be the micro second in which to play the audio.

Data – actual data in the message according to the payload type

The applications on both sides can then use this format in order to communicate and send video/audio data such as defined above (H264/MP3) in RTP packets and encode the sequence numbers and timestamps as appropriate. The different MPEG packets can then be stored in one RTP packet , multiple RTP packet (for big MPEG packets such as I-frame) or even small multiple MPEG packets in one RTP packet.

The RTP protocol is commonly used with the Real-time Control Protocol (RTCP), this protocol adds additional information between the source and destination in the form of "report" packets, these reports contain statistical information such as inter-arrival jitter, number of packets send and lost, this data is send between every destination and the source.

Since there can be multiple receivers this control data rate is adjusted according to the number of receivers and is limited to less than 5% than the overall bandwidth.

These reports are than used in order to perform the flow control and congestion control as well as rate adjustment in the source.

In this project we don't use RTCP but we developed a flow control algorithm which is more suitable for this application as described in chapter III .

7. Simulator (OMNET++/MiXiM)

The simulator framework which is being used is a discrete event based network simulator (OMNET++),

The nature of this simulator is that it allows modeling of dynamic network topology with possibly moving nodes which can communicate with one another in the form of packets, every such packet which is being send passes through the various Open System Interconnection (OSI) (Zimmermann, 1980) network layers in both sender and receiver where events are created and extensible components can be used to add functionality, this functionality can be defined using small C++ modules and the hierarchy can be described using a textual language (called NED) to represent the different components of the system (such as NIC and its MAC).

The simulator itself provides a parametric textual configuration which allows to easily define different runs according to changing parameters such as transmission power, thermal noise and etc.

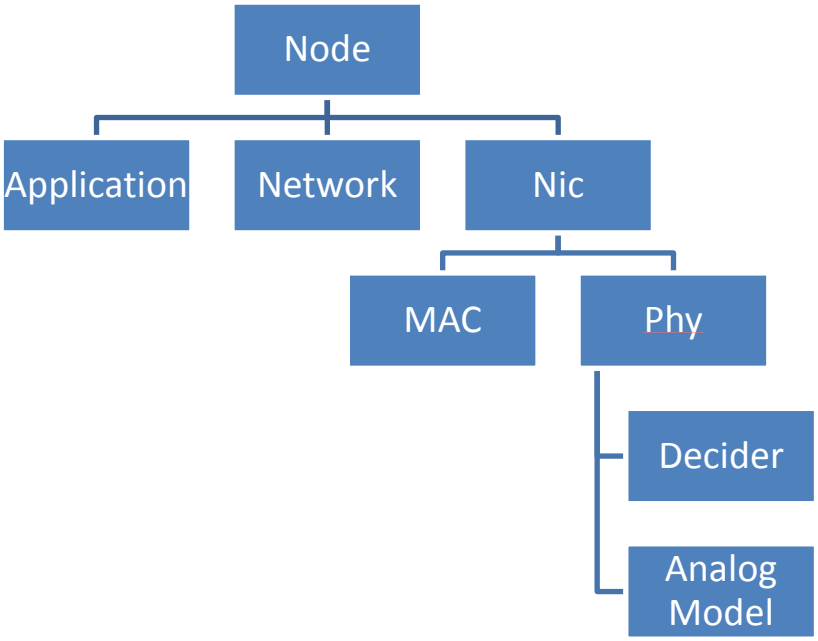


Figure II-6 Network simulator node hierarchy

The wireless network can be modeled using the MiXiM library (See (MiXiM)) , this library allows to model wireless network and protocols , it adds the following important components:

- Mobility of nodes in a 2D/3D world based on events of movement .
- Connection modeling between different nodes operating in the same frequency.
- Channel modeling of signal attenuation in air and obstacles (such as path-loss, fading and shadowing) and noise introduction.
- Physical layer modeling in the node.

The heart of the modeling is the physical layer, this layer receives the signal which is being send from any source, this signal carries the information of time, frequency and space, once this information is available the signal-to-noise (SNR) ratio is calculated after the attenuation and noise is added, most importantly this ratio includes as noise also the interference from other signals which are in the air (making it effectively the signal to noise and interference ratio (SINR)).

Once the SNR per such packet is calculated a Decider module is used in order to make a per-node decision whether the signal can be used to decode a packet or isn't strong enough for this purpose, since there can be many different events which would have different SNR's throughout the duration of the packet (in time or frequency) the minimum SNR is used.

This SNR is used by the decider in a module which models a packet loss however in a real environment the data is being coded into symbols possibly carrying multiple bits according to a modulation of the transmission (in a transmission scheme/bit rate), since each symbol may have errors which can cause errors for the entire packet (and later to a movie frame) a random model depending on packet different parts (such as header , data and tail) and different modulation is created and models whether a whole packet is received or not, this model is used in order to hide the complexity of bit based coding, transmission and decoding (and error correction) (See (Cocorada, OMNET++/MiXiM 802.11g)).

III. Flow control

While the scheduler assumes a steady state in a real environment such steady state is difficult to reach due to the following reasons:

- Noise is random and thus some packet loss would occur which changes the data rate.
- Scheduling is computed based on static locations however nodes may perform movements which can alter the data rate.
- The scheduling is computed based on constant rate however when working with the 802.11g protocol the collision avoidance used in the protocol causes random quiet periods, these random periods change the data rate.
- The scheduler is computed based on a model of noise and signal loss , the real environment may not be fully accurate to this model and these difference will cause data rate changes (it should be noted the simulator doesn't model this since it uses the same model as the scheduler).

The changes in data rate cause fluctuations in the rate of data flow in and out of nodes, these positive differences are accumulates in the nodes queues , it is well known in queuing theory that such unsteady state causes queues to grow bigger and bigger over time, this can be seen in the experiments below (See section III/2) , for this reason a flow control mechanism has been developed in order to keep the system in a steady state according to the scheduler.

The multi-flow table computed by the scheduling algorithm determines the number of packets *mf* (e, s) that should be sent along each link *e* for stream *s* during each period. Each node *v* monitors the following information for each outgoing link *e* from *v* , $e \in E_{out}(v)$:

1. *P(e, s, t)* - the number of packets belonging to stream *s* sent along the link *e* during the period t.
2. $P^{+}(e,s,t)$ - the maximum number of packets belonging to stream *s* that can be sent along the link *e* during the period *t*.

   Note that $P^{+}(e,s,t) \geq P(e, s, t)$; inequality may happen if the queue *Q(e, s)* is empty when a packet is scheduled to be transmitted along the link *e*. Note that if *e* is not planned to deliver packets of stream *s*, then $P^{+}(e,s,t) = 0$.

We remark that a node *v* can also monitor *P(e, s, t)* for an incoming link *e* to *v* , $e \in E_{in}(v)$. However, the value $P^{+}(e,s,t)$ for a link $e \in E_{in}(v)$ must be sent to *v* (e.g., by appending it to one of the delivered packets).

The Flow-Control algorithm is executed locally by all the nodes in the network. Let *e = (u, v,m)* denote a link from *u* to *v* in channel *m*, and let *s* denote a stream. Each node executes a separate instance per stream. In the end of each period *t*, each node *u* "forwards" the value of $P_{+}(e, s, t)$ to node *v*. In addition, in the end of each period *t*, node *v* sends "backwards" the value *R(e, s)* to *u*. The value *R(e, s)* specifies the number of packets from stream *s* that *v* is willing to receive along the link *e* in the next period *t + 1*.

Algorithm Flow-Control (v, s) - computed on each node v locally for managing the local queue and incoming rate for stream s.

1. Initialize: for all $e \in E_{in}(v)$, $R(e, s) \leftarrow mf(e, s)$.
2. For $t = 1$ to ∞ do
   a) Measure $P(e, s, t)$ for every $e \in E(v)$, and $P_+(e, s, t)$ for every $e \in E_{out}(v)$.
   b) Receive $P_+(e, s, t)$ for every $e \in E_{in}(v)$, and $R(e, s)$ for every $e \in E_{out}(v)$.
   c) $R_{in} \leftarrow min\{\sum_{e \in E_{out}(v)} R(e, s), \sum_{e \in E_{out}(v)} P_+(e, s, t), \sum_{e \in E_{in}(v)} P_+(e, s, t)\}$.
   d) $for\ every\ e \in E_{in}(v): R(e, s) \leftarrow Rin * \frac{P^+(e,s,t)}{\sum_{e` \in E_{in}(v)} P^+(e`,s,t)}$
   e) Drop oldest packets from $Q(v, s)$, if needed, so that $|Q(v, s)| \leq R_{in}$.

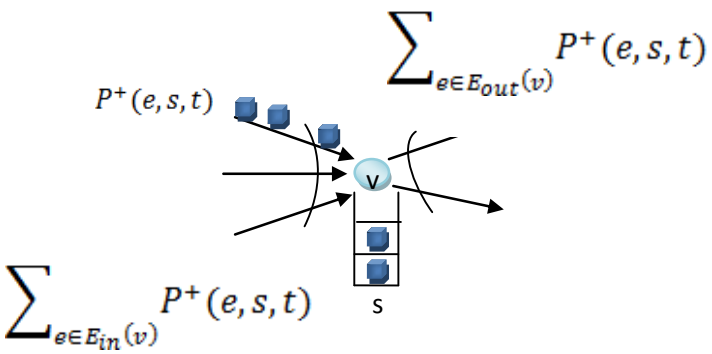


Figure III-1 Node flow control

The above figure illustrates the flow control operation in node *v* when operating on stream *s* , the packets enter on incoming edges into the stream's queue and the flow control summarizes $P+(e,s,t)$ for all incoming and outgoing edges and acts accordingly

The Flow-Control algorithm is used in order to equalize the incoming and outgoing packet-rates in intermediate nodes as follows: the requested packet-rate $R(e, s)$ is initialized to be the value $mf(e, s)$ derived from the scheduling table. This algorithm is activated in the end of each period of the scheduling table and it uses the values $P(e, s, t)$ and $P_+(e, s, t)$ for every link *e* incident to *v*.

Some of these values are computed locally and some sent by the neighbors but the computation is done in each node with neighbors information only and there is no end-to-end information transmission.

The incoming packet-rate R_{in} is computed in line *2c*, this rate is divided fairly among the incoming links in line *2d*. Excess packets in the queue $Q(v, s)$ are dropped so that the number of packets in $Q(v, s)$ is at most R_{in}. The rational is that, in the next period, at most R_{in} packets will be delivered, and hence, excess packets might as well be dropped, we drop oldest packets since those have less chance to be used by a real time video decoder.

In the above algorithm the source and destination nodes require special handling, the destination doesn't have any outgoing edges therefore we set the "incoming" rate to be the scheduling result, i.e. $R(e, s) \leftarrow mf(e, s)$. It should be noted we can also set this

rate to the stream data request and this would allow exceeding the schedule results (by using only the slots allocated by the scheduler thus not adding more interference to other nodes) however the schedule is designed to guarantee stable queues and minimal end-to-end delay therefore it is more reliable to use this metric.

The source node sets the video encoder packet-rate to R_{in} .

1. Flow control implementation

The flow control algorithm is implemented in the following method, it performs the flow control inside a node for a specific queue of a stream, it receives the nodeID of the node it is working on and the round number:

```
void VideoStreamAppl::Stream::flowControl(int nodeID,int
round)
{

  if (++flowControlMessagesReceivedInRound <
outEdges.size())
    return; // wait till we get all the flow control
messages from out edges, skip if not all edges are done

  flowControlMessagesReceivedInRound = 0; // reset flow
control for node
  unsigned long int roundOut=0, // edges out total
transmitted rate (P+)
                totalIn=0, // edges in total transmitted
              rate (P+)
                rateOut=0; // edges out transmitted rate (P)

   if (outEdges.empty()) { // destination node
     ..
     // set roundOut,rateOut and maxRateOut to be the
request rate per this stream in bits per second
   }
   else // intermediate node or source
     // go over all outgoing edges for this stream
     FOR_EACH(OutEdgesCollection,eIter,outEdges) {
         // add edge total transmitted rate in round and
ghosts (P+)
        roundOut += eIter->second.getTotalRoundRate();
         // add edge total existing rate (R(e,s))
        rateOut += eIter->second.rate;
     }

   if (inEdges.empty())// if source node then special case
where camera is a feeding edge
        totalIn = rateOut;
   else // not a source - go over all incoming edges
     FOR_EACH(InEdgesCollection,inIter,inEdges)
       // add to total input count the flow over the input
      edge (P+)
       totalIn += inIter-
>second.getTotalRoundRate(nodeID,streamNumber);

    // compute 2.c in algorithm: minimum over P+ of in,out
and current R
    unsigned long rateIn = rateOut;
    if (rateIn > roundOut)
        rateIn = roundOut;
    if (rateIn > totalIn)
        rateIn = totalIn;
```

```
FOR_EACH(IntSet,it,incomingTasks) { // go over all incoming
node ID's
      int sendingToMeNodeID = *it;
      if (sendingToMeNodeID==nodeID) // if camera node then
skip
        continue;
      VideoStreamAppl * sendingToMeAppl =
VideoStreamAppl::apps[sendingToMeNodeID];

      unsigned long actual = sendingToMeAppl-
>streams[streamNumber]->outEdges[nodeID].roundOut; // bits
transmitted to me : P()
      unsigned long nodeIn = sendingToMeAppl-
>streams[streamNumber]-
>outEdges[nodeID].getTotalRoundRate();// bits transmitted to
me with phantoms: P+()
      float weight = (((float)nodeIn)/totalIn); // edge
weight from all incoming
      unsigned long newRate = weight*rateIn;
      unsigned long oldRate = sendingToMeAppl-
>streams[streamNumber]->outEdges[nodeID].rate;

      sendingToMeAppl->streams[streamNumber]-
>outEdges[nodeID].rate = newRate; // update new rate of node
      if (newRate != (unsigned long)oldRate) { // if changed
rate
        // change RTP video pool if a camera node and call
flow control of origin
        sendingToMeAppl->setRtpPool(streamNumber);
        sendingToMeAppl->streams[streamNumber]-
>flowControl(sendingToMeNodeID,round);
      }
    }

   if (!outEdges.empty())// call the queue drop according to
new rate if not destination
        queueDrop(nodeID,rateIn);
}
```

```
// perform queue drop from node's stream queue according to
rate (in Bytes)
void VideoStreamAppl::Stream::queueDrop(int nodeID,unsigned
long rate)
{
  if (queueSize*8<rate) // skip if meeting rate (queueSize
in bit)
        return;
  int reduce = (queueSize*8-rate)/8; // compute bytes to
reduce,out in bits,queue in bytes
  int newSize =  queueSize - reduce; // bits transmitted :
P()
  unsigned removeSize = 0;
  while ((unsigned)newSize < queueSize) { // while still
didn't remove all
   ApplPkt * last = incomingMessages.front();// get first
packet in queue
    AdHocWiFiMsg msgDetails(last->getName());
        FOR_EACH(IntList,idsIter,msgDetails.IDs)  {// mark
as dropped

AdHocWiFiApplLayer::messagesStatistics[*idsIter].lastRecieve
= PKT_DROPPED;
          msgDetails.recordLog(AdHocWiFiMsg::Drop); //
record to log
        }
   int packetSize = last->getBitLength()/8;
   delete last;
   queueSize -= packetSize; // update queue size
   removeSize += packetSize; // update size removed
   incomingMessages.pop_front();// actual remove of packet
from queue
  }
}
```

2. Flow control experimental results

The following figures illustrate the effect of applying the flow control algorithm, in the following series of figures the figures on the left depict the run result with the flow control and the figures on the right depicts the results without applying the flow control algorithm, these figures are the result of an execution of a random grid scenario of 7x7 nodes spread equally across 1000mX1000m with 12 requests over a duration of 5 minutes as described in the experimental results of the article (See (Even, Fais, Medina, Shahar, & Zadorojniy, 2011)).

The first two figures (Figure III-2, Figure III-3) show a histogram of the end-to-end delay of all the packets of one of the streams in this scenario (stream #10), it can be seen that the flow control reduces the end-to-end delay from a maximum of 20 seconds to 1.6 seconds.

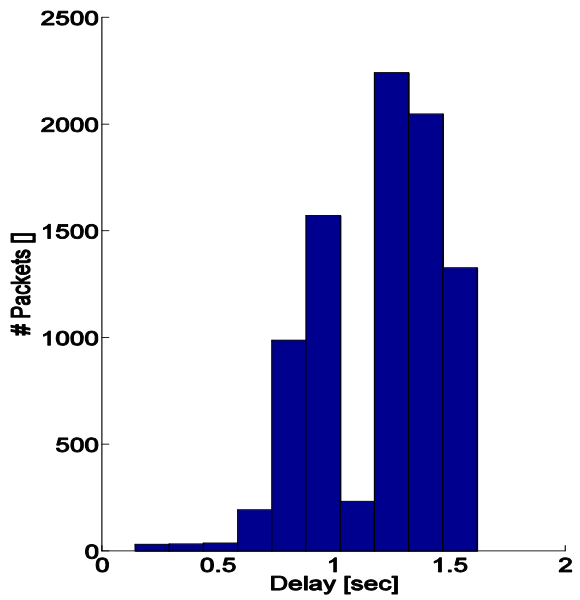


Figure III-2 End-to-end delay histogram (flow control)

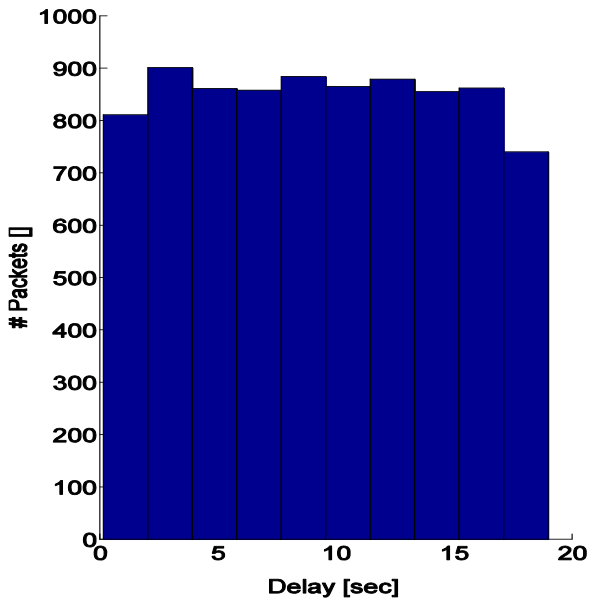


Figure III-3 End-to-end delay histogram (no flow control)

The following graphs (Figure III-4 , Figure III-5) show the end-to-end delay of each packet over all of the packets sent along this stream, it can be seen on the left that when using the flow control the end-to-end delay remains mostly the same (with fast ramp-up until queues in the system fill-up) but when not using the flow control on the right the end-to-end delay grows linearly.

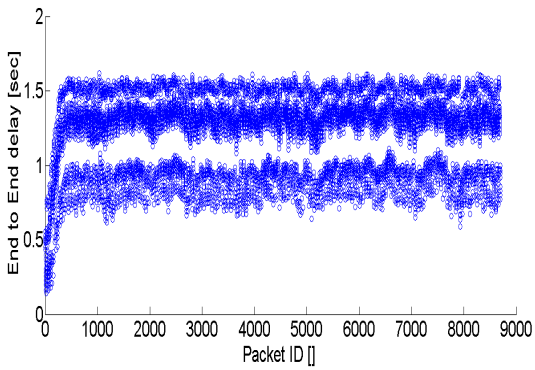


Figure III-5 End-to-end delay per packet (flow control)

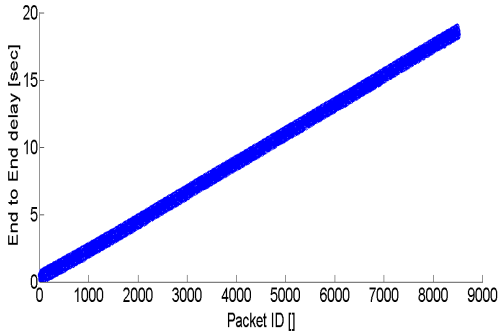


Figure III-4 End-to-end delay per packet (no flow control)

The following figures (Figure III-6 , Figure III-7) depict the queue size in number of packets over time for this stream, a very disturbing but expected phenomenon appears on the right when not using the flow control, the queue size grows linear with time, this is the source of the growing end-to-end delay. When using the flow control (on the left) the queue sizes make small fluctuations over some small constant number of packets making the queue bounded and stable.

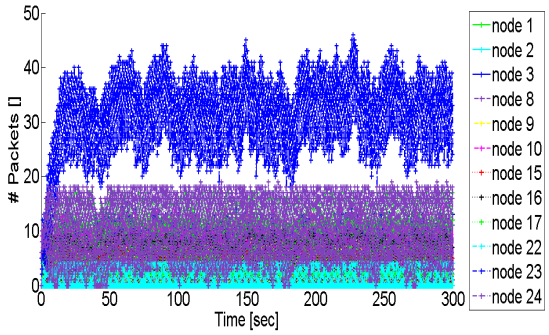


Figure III-7 Stream #10 Queue sizes (flow control)

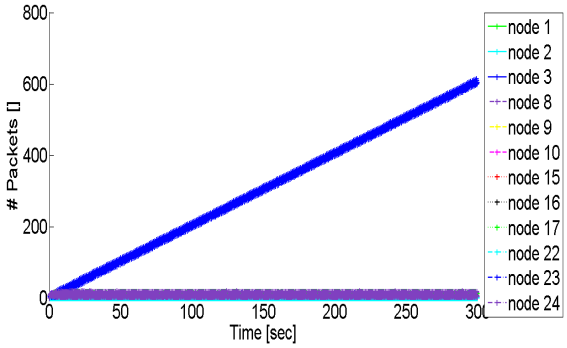


Figure III-6 Stream #10 Queue sizes (no flow control)

The following graphs (Figure III-8 , Figure III-7) shows the packet drop rate of stream #10 when using the flow control, it can be seen that the drop rate is very low (below 5%, the average drop rate is 0.7%), obviously when not using the flow control the drop rate is 0%.

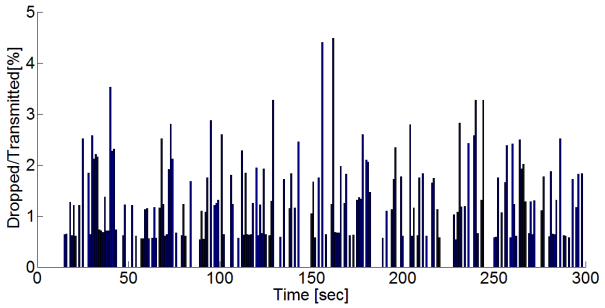


Figure III-8 Stream #10 packet drop rate

The throughput of the two runs is shown in the following figures (Figure III-9 , Figure III-10) , it is interesting to see that the flow-control managed to increase the received ratio in this case and that the system is stable in terms of throughput in both cases, as expected.

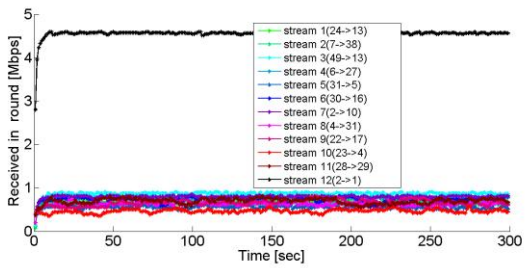


Figure III-10 Throughput (flow control)

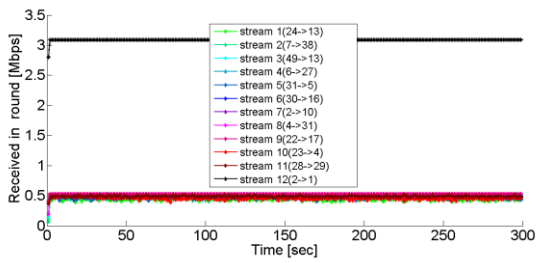


Figure III-9 Throughput (no flow control)

IV. System Overview

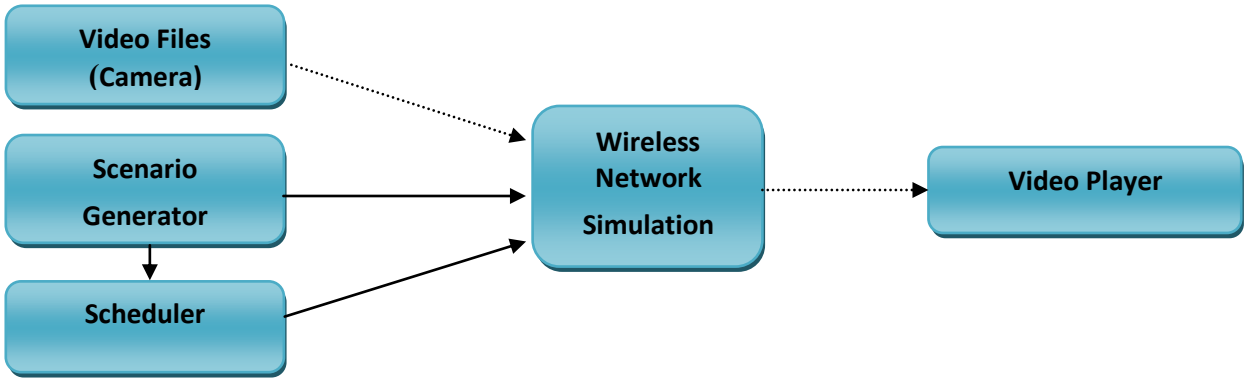


Figure IV-1 System overview

The project system is composed of a scenario generator, a scheduling algorithm and a network simulator, The first two components are external and are considered inputs to the simulator.

The output of the simulator is the actual network traffic, this output can be used to compose a video from input video stream traffic, and this video data can be then played using a video player.

The simulator can also be used for transmission of non-video packets through a parameter setting (see section VI / 9)VI .

1. Design Challenges

Trying to solve the specific project problem over common tool sets meets several challenges.

The following section would describe such challenges and solutions taken:

a) Encoding/Transmitting/Playing real-time video streams over simulated network

As the video should be generated in real time (possibly with different encoding schemes) and should also be played and transmitted in real time as a stream there is a problem of running all these components in real-time, this is specifically not possible since the network simulator doesn't work in real time due to the complexity of simulating all the events of multiple nodes of the network.

This requires bridging between the real-time and non-real time components of the system, for this purpose the encoding is done in pre-processing step by preparing a video with multiple rates, the playback of the transmitted video which achieves the destination node is done in post-processing as the network simulator only saves the "trace" of the arriving packets.

b) Multiple channels simulation

MiXiM base classes require a Phy/Mac/Application layering per node whereas the project requirements are for a node to be able to communicate over the network in multiple channels in multiple frequencies.

In order to enable this higher layer of high level video application was added such that this upper layer would be unique per-node and would communicate with the lower layer application, each node has be several lower application layers (and corresponding Mac/Phy) per node where each such component communicates with one channel (frequency).

Apart from creating the above layering there is the need to create a connection manager per channel in the simulated "world" and to connect the appropriate low level applications (per node) to this connection manager which is responsible for one channel.

c) Support for 802.11g model

OMNET++ MiXiM wireless network simulation component comes built in with support only for 802.11a/b however version 'g' was required.

Previous projects have used the predecessor of MiXiM where support for 802.11g was already available in open source, in order to enable such support inside MiXiM this code was used and modified to suit on top of MiXiM base classes.

The above required creating another class inheriting from MiXiM base MAC80211 class, in this class the state machines and parameters are modified to suite the 802.11g behavior. (See (Cocorada, OMNET++/MiXiM 802.11g) , (Cocorada, An IEEE 802.11g simulation model with extended debug capabilities, 2008) , this includes code snippets which were used exactly in the simulator as shows below)

The implementation of the 802.11g is done in two components, the first is the MAC layer and the second is the decider.

The MAC is enhanced by having the Mac80211g which derives from the Mac80211 class of MiXiM and the packetDuration method which overloads the base class method as following (File Mac80211g.cpp,note the method of the base class of MiXiM was changed to be virtual):

```
/**
 * @brief computes the duration of a transmission
 *      over the physical channel, given a certain bitrate
 * @param bits number of bits in packet
 * @param bt bitrate of transmission
 */

simtime_t Mac80211g::packetDuration(double bits, double br)
{
  static const double PHY_HEADER_TIME = 28e-6;
  double duration  = ceil((16+bits+6)/(br/1e6))*1e-6 +
PHY_HEADER_TIME;
  return duration;
}
```

Note this calculation follows the theoretical duration as in (Zadorojniy, Even, & Moni, Frequency and Time Slot Assignment Algorithm, 2009).

The second enhancement in order to support 802.11g is done in the Decider module which supports the decision whether a signal which is being sensed in a node is a packet or noise.

The implementation is done in the AdHocWiFiDecider module (File AdHocWiFiDecider.cpp).

```
/**
 * @brief computes whether a signal is OK (packet)
 * @param snrMap signal and noise power over time/frequency
domains to check
 * @param frame packet transmitted wireless
 */
DeciderResult* AdHocWiFiDecider::checkIfSignalOk(Mapping*
snrMap, AirFrame* frame)
{
// get the decider result
      DeciderResult80211 * dr =

dynamic_cast<DeciderResult80211*>(checkIfSignalOkBase(snrMap,fr
ame));


      // extract application packet (can be NULL if control
packets)
      cPacket * msgPhy = frame->getEncapsulatedMsg();
      cPacket * msgMac = msgPhy->getEncapsulatedMsg();
      if (msgMac==NULL) return dr;
      cPacket * msgApp = msgMac->getEncapsulatedMsg();
      ApplPkt *msg = dynamic_cast<ApplPkt *>(msgApp);

…
      // if message was transmitted to this node (by address)
then add the computed SINR to application statistics
      if (myIndex==msg->getDestAddr())
        VideoStreamAppl::addSINR(msg,dr);

      return dr;
}


/**
 * @brief computes whether a signal is OK (packet)
 * @param snrMap signal and noise power over time/frequency
domains to check
 * @param frame packet transmitted wireless
 */
DeciderResult* AdHocWiFiDecider::checkIfSignalOkBase(Mapping*
snrMap, AirFrame* frame)
{
       …

      Signal& s = frame->getSignal();// get signal strength
      simtime_t start = s.getSignalStart();// get signal start
time
      simtime_t end = start + s.getSignalLength();//get signal
end time
      DeciderResult80211* result = 0;

      ConstMappingIterator* bitrateIt = s.getBitrate()-
>createConstIterator();
```

```
            bitrateIt->next(); //iterate to payload bitrate indicator
            double payloadBitrate = bitrateIt->getValue();
            delete bitrateIt;
            double sinrMin;
      …
            start = start + RED_PHY_HEADER_DURATION; //its ok if the
phy  header is received only
            Argument min(DimensionSet::timeFreqDomain);
            min.setTime(start);
            min.setArgValue(Dimension::frequency_static(),
      centerFrequency - 11e6);
            Argument max(DimensionSet::timeFreqDomain);
            max.setTime(end);
            max.setArgValue(Dimension::frequency_static(),
      centerFrequency + 11e6);

           // find minimum SINR from start time to end time
            sinrMin = MappingUtils::findMin(*snrMap, min, max);

            if (sinrMin > snrThreshold) { //if above basic threshold
                  bool berPacket = false;
                  berPacket = packetOkSorin(sinrMin, frame-
      >getBitLength() - (int)PHY_HEADER_LENGTH, payloadBitrate);
                  if (berPacket) { // no errors !!
                              result = new DeciderResult80211(OK,
      payloadBitrate, sinrMin);
                  } else { // got errors
                        EV << "Packet has BIT ERRORS! It is lost!\n";
                        result = new DeciderResult80211(false,
      payloadBitrate, sinrMin);
                  }
            }
            else { // three sections : header and FCS-any bit bad
      then all bad,for payload bad-bits/total-bits < p then ok
                  Argument min(DimensionSet::timeFreqDomain);
                  min.setTime(start);
                  min.setArgValue(Dimension::frequency_static(),
      centerFrequency - 11e6);
                  Argument max(DimensionSet::timeFreqDomain);

      max.setTime(start+RED_PHY_HEADER_DURATION+16/payloadBitrate);
      // header+service
                  max.setArgValue(Dimension::frequency_static(),
      centerFrequency + 11e6);

                 // compute minimum SINR in header
                  sinrMin = MappingUtils::findMin(*snrMap, min, max);
                  if (sinrMin > snrThreshold) { // header good
                        min.setTime(end-6/payloadBitrate); // trailer
                        max.setTime(end);
                        // compute minimum SINR in trailer
                        sinrMin = MappingUtils::findMin(*snrMap, min,
      max);
                        if (sinrMin > snrThreshold) { // trailer good

            min.setTime(start+RED_PHY_HEADER_DURATION+16/payloadBitra
      te); // payload
                              max.setTime(end-6/payloadBitrate);
                              if (countBelowThBer(*snrMap, min,
      max,snrThreshold*3,payloadBitrate)<payloadBerThreshold) // have
      enough good bits , results is OK
```

```
                                result = new
DeciderResult80211(OK, payloadBitrate, sinrMin);
                            else // too many errors in payload,
fail
                                result = new
DeciderResult80211(false, payloadBitrate, sinrMin);
                        }
                }
        }

        if (result==NULL) { // errors , lost packet
                EV << "Packet has ERRORS! It is lost! " << frame-
>getDisplayString() << " " << frame->getId() << endl;
                result = new DeciderResult80211(false,
payloadBitrate, sinrMin);
        }

        return result;
}

/**
  * calculate bit error rate for BPSK modulation
  * @param sinr SINR to check
  * @param bandwidth frequency of transmission
  * @param bitrate transmission bitrate
  * @param channelModel AWGN/Rayleigh model
  **/
double ber_bpsk(double sinr, double bandwidth, double bitrate,
char channelModel){
  // … implementation formula follows article …
}

/**
  * calculate bit error rate for QPSK modulation
  * @param sinr SINR to check
  * @param bandwidth frequency of transmission
  * @param bitrate transmission bitrate
  * @param channelModel AWGN/Rayleigh model
  **/
double ber_qpsk(double sinr, double bandwidth, double bitrate,
char channelModel){
  // … implementation formula follows article …
}
/**
  * calculate bit error rate for 16-QAM modulation
  * @param sinr SINR to check
  * @param bandwidth frequency of transmission
  * @param bitrate transmission bitrate
  * @param channelModel AWGN/Rayleigh model
  **/
double ber_16qam(double sinr, double bandwidth, double bitrate,
char channelModel){
  // … implementation formula follows article …
}

/**
  * calculate bit error rate for 64-QAM modulation
  * @param sinr SINR to check
  * @param bandwidth frequency of transmission
  * @param bitrate transmission bitrate
  * @param channelModel AWGN/Rayleigh model
```

```
  **/
double ber_64qam(double sinr, double bandwidth, double bitrate,
char channelModel){
  // … implementation formula follows article …
}



/**
  * return true if packet is OK
  * @param sinrMin minimum SINR in packet duration
  * @param lengthMPDU length of MPDU (Phy header) of packet
  * @param bitrate transmission bitrate
  **/
bool AdHocWiFiDecider::packetOkSorin(double sinrMin, int
lengthMPDU, double bitrate)
{
    double berHeader, berMPDU;

…

        //PLCP Header 24bits, BPSK, r=1/2, 6Mbps
        berHeader=ber_bpsk(sinrMin, BANDWIDTH , 6E+6,
channelModel);
        berHeader=Pb(1, berHeader);


        switch((int)bitrate){ // calculate BER in payload based
on MCS
            case (int)(6E+6)://6Mbps, r=1/2, BPSK
                  berMPDU=ber_bpsk(sinrMin, BANDWIDTH ,
bitrate, channelModel);
                  berMPDU=Pb(1, berMPDU);
            break;
            case (int)(9E+6)://9Mbps, r=3/4, BPSK
                  berMPDU=ber_bpsk(sinrMin, BANDWIDTH ,
bitrate, channelModel);
                  berMPDU=Pb(3, berMPDU);
            break;

            case (int)(12E+6)://12Mbps, r=1/2, QPSK
                  berMPDU=ber_qpsk(sinrMin, BANDWIDTH ,
bitrate, channelModel);
                  berMPDU=Pb(1, berMPDU);
            break;
            case (int)(18E+6)://18Mbps, r=3/4, QPSK
                  berMPDU=ber_qpsk(sinrMin, BANDWIDTH ,
bitrate, channelModel);
                  berMPDU=Pb(3, berMPDU);
            break;

            case (int)(24E+6)://24Mbps, r=1/2, 16QAM
                  berMPDU=ber_16qam(sinrMin, BANDWIDTH ,
bitrate, channelModel);
                  berMPDU=Pb(1, berMPDU);
            break;
            case (int)(36E+6)://36Mbps, r=3/4, 16QAM
                  berMPDU=ber_16qam(sinrMin, BANDWIDTH ,
bitrate, channelModel);
                  berMPDU=Pb(3, berMPDU);
            break;
```

```
            case (int)(48E+6)://48Mbps, r=2/3, 64QAM
                  berMPDU=ber_64qam(sinrMin, BANDWIDTH ,
bitrate, channelModel);
                  berMPDU=Pb(2, berMPDU);
            break;

            case (int)(54E+6)://54Mbps, r=3/4, 64QAM
                  berMPDU=ber_64qam(sinrMin, BANDWIDTH ,
bitrate, channelModel);
                  berMPDU=Pb(3, berMPDU);
            break;
            default:
                  berMPDU=0;
            }


      }

      if((berHeader > 1.0 || berMPDU > 1.0))
          return false;// error in MPDU
    // probability of no bit error in the PLCP header
      double headerNoError;
      if (phyOpMode=='g')
            headerNoError = pow(1.0 - berHeader, 24);//PLCP
Header 24bit(without SERVICE), 6Mbps
      else
      headerNoError = pow(1.0 - berHeader,
HEADER_WITHOUT_PREAMBLE);

    // probability of no bit error in the MPDU
    double MpduNoError = pow(1.0 - berMPDU, lengthMPDU);

    double rand = dblrand();

    if (rand > headerNoError)
        return false; // error in header
    else if (dblrand() > MpduNoError)
        return false;  // error in MPDU
    else
        return true; // no error
}
```

2. Detailed system architecture

The network simulator system is partitioned into several components according to input/output processing and common network layers modeling.

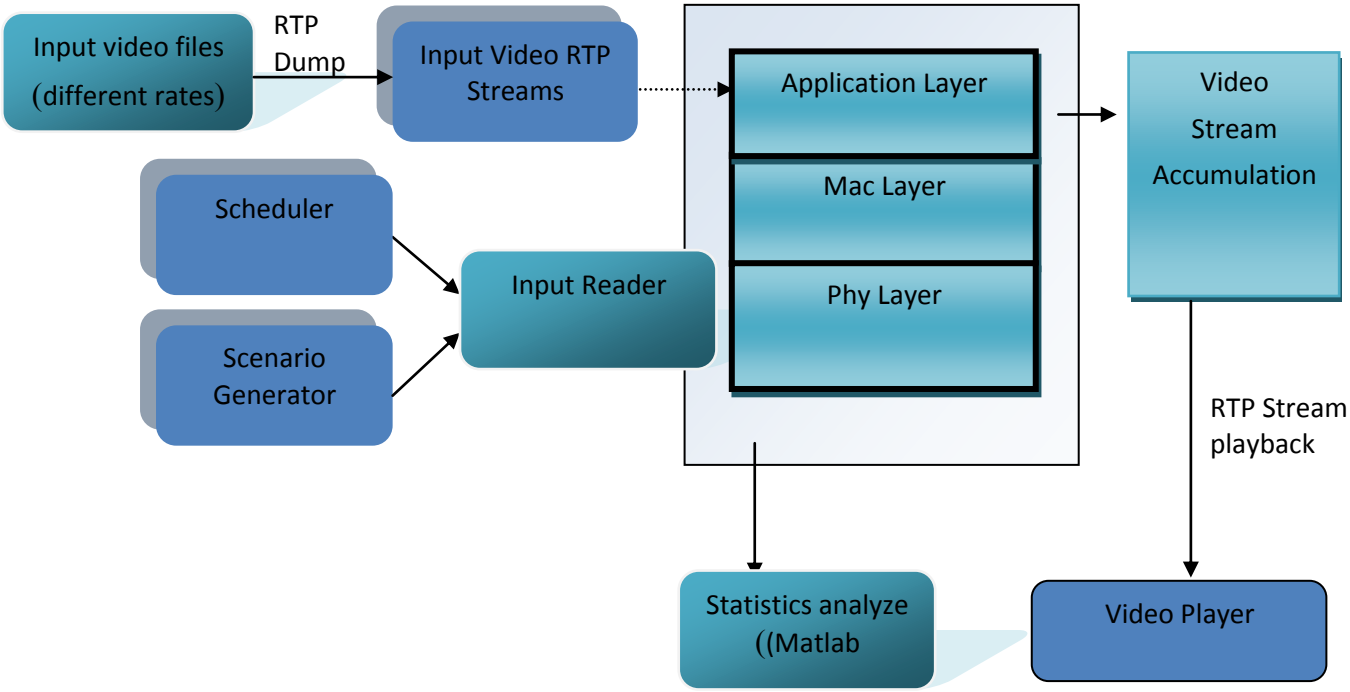


Figure IV-2 Detailed implementation architecture

1. <u>RTP Dump</u>
   In order to enable dynamically changing rate of the camera node a video file is being encoded in multiple data rates by changing the frame-per-second and encoder bit-rate (quality) , in this way multiple video stream files are created with RTP formatted data.
2. <u>Input scenario reader</u>
   The input of the simulator is the scenario generator and the scheduling table. The scenario generator should provide parameters to the network model whereas the scheduler should provide the transmission table to nodes.
   The input scenario uses reader from parameter files similar to the scheduler where possible, in this way a configuration is statically created from the scenario parameters file and converted to the OMNET++ configuration files.
3. <u>Input video reader</u>
   The RTP dump data is being read and the packets meta data (data-rate,unique ID and time) are used in order to transmit the simulated packets in the network, it should be noted this is used only in a video-streaming mode, in normal mode fixed size packets with no video-information are being sent.
4. <u>Network Physical (PHY) Layer and Analogue Model</u>
   As described in the Phy and Analogue Models are responsible for simulating the attenuation (like shadowing, fading and path loss) of a received signal. The Radio module simulates the physical characteristics of the radio, like switching times and simplex / duplex capabilities and the Decider is

responsible for evaluation (classification as noise or signal) and demodulation (bit error calculation) of the received messages. Additionally, it provides channel sensing information to the MAC layer. These models implement the specific path loss model and a decider based on SINR model , implement BER rate according to modulation.

5. Network Media Access Control (MAC) Layer
   The MAC layer simulates the 802.11g media access control protocol, this includes carrier sensing, request to send and etc.
6. Node
   Each node implements:
   a. scheduled transmission according to scheduler table of video data, including transmission of RTP data packets from dump files as described in section II / 4 .
   b. Receive and accumulate of packets to compose a video frame and dump to meta-file with video stream data (holding only RTP packet ID's number and time).
   c. Node movement according to scenario generator.
   d. Collecting of statistics and saving to OMNET++ format.
   e. Flow control algorithm.
7. RTP dump playback
   Enable playback of simulation results meta-video-data file and video RTP file into video player over local network
8. Statistics processing
   Simulation results are processed for needed statistics using Matlab and needed graphs are created from files which are created during simulation run.

3. Simulator output

OMNET++ statistics data format is used in order to save any statistics of the network simulation, this output textual data format is then processed in order to create summary tables and graphs.

The OMNET++ scavetool is used in order to convert the data to CSV format or Matlab in order to be processed.

There are three file types which are generated from the simulator:

1. The statistics which is stored in OMNET++ format includes the following data (file name: <configuration.csv>):

Initial_Positions X per node

Initial_Positions Y per node

PlaygroundSize

Rounds

Slots

Source ID's

Destination ID's

TX_POWER [mW]

PathLossAlpha

PathLossBeta

Noise [dBm]

Scheduling Table From node ID

Scheduling Table To node ID

Scheduling Table Slot

Scheduling Table Channel

Scheduling Table Stream

Throughput per stream (x # streams)

Queue length (max/mean) per node per stream [#messages]

Packet's TX time per stream [sec] for each packet

Packet's RX time per stream [sec] for each packet

Packet's End-to-End delay per stream for each packet [sec]

[Note] some of these fields exist in the scenario generator files however they are transferred to the CSV file in order to simplify the Matlab Processing script

2. Detailed log about the message transmission in the following format (File: <configuration>_log.csv)
   Message ID,from,to,stream,slot,channel,SNIR,Event ID(0=Recv,1=Send,2=Lost,3=Drop),Time

   All the messages transmitted or received from each node to node in the simulation would appear in this log, in addition every event of packet loss (by the decider) and an event of packet drop by flow control would appear as an entry in this log.
   This log can be used to gather comprehensive statistics about simulation trace state and is used for analysis in the matlab script (See section IV / 2).

3. RTP Dump for every destination node and stream (if working with Video) (File: node<destination>_<stream>.txt)
   Format:
   #<video-rate-ID>_<Frame-Per-Seconds>
   <Packet ID> @<Time>
   Any of the above lines can appear multiple times in the text file, the first indicating the current video RTP file and second indicating a packet ID arrival time.
   This output is used when replaying a video using rtpreplay (See section IV / 2)
   This file is generated only when running in "video" mode.

4. Analysis module output

The statistics graphs which are generated include:

a. Throughput per stream per scheduling round
b. Throughput over last round compared to expected from scheduler
c. Packet drop per stream
d. Packet drop over time
e. Queue sizes over time (max, mean) sampled several times in each round
f. End-to-End delay of streams histogram
g. End-to-End delay of streams per packet
h. Number of hops per stream
i. Scenario surface layout
j. Packet error rate per stream
k. SINR/SNR comparison per scheduling table entry (slot x stream x channel) for highest , median and lowest differences

See explanation of result files in VI/1) .

5. Detailed System Design

a) *RTP Dump*

In order to enable dynamically changing rate of the camera node a video file is being encoded in multiple data rates by changing the frame-per-second and encoder bit-rate (quality) , in this way multiple video stream files are created with RTP formatted data.

The RTP Dump uses the "rtpdump" utility from (Schulzrinne, RTP Tools) , the encoding and changing of video rates is done using a video encoding utility (See (erightsoft, 2011)) and is then played and dumped as RTP file.

b) *Input scenario reader*

The input of the simulator is the scenario generator and the scheduling table. The scenario generator provides parameters to the network model such as nodes locations and transmission parameters and channel fading parameters whereas the scheduler provides the transmission table to nodes.

The input scenario uses reader from parameter files similar to the scheduler wherever possible, in this way a configuration is statically created from the scenario parameters file and converted to the OMNET++ configuration files. Though the simulator reads some files of the scheduler/scenario generator directly (Such as scheduling table, requests and etc.) some information is needed before simulation starts such as number of nodes.

This is implemented under the IniGen directory.

c) *Input video reader*

The RTP dump data is being read and the packets meta data (data-rate, unique ID and time) are used in order to transmit the simulated packets in the network, it should be noted this is used only in a video-streaming mode, in normal mode fixed size packets with no video-information are being sent. Moreover only the meta-data is used in the network simulator and not all of the video data in order to avoid allocating large memory for simulation, there is no importance for carrying actual data since the packets are not being encoded but only for the destination.

This is implemented in the RtpDump class.

d) *Network Physical (PHY) Layer and Analogue Model*

The Phy and Analogue models are responsible for simulating the attenuation (like shadowing, fading and path loss) of a received signal. The Radio module simulates the physical characteristics of the radio, like switching times and simplex / duplex capabilities and the Decider is responsible for evaluation (classification as noise or signal) and demodulation (bit error calculation) of the received messages. Additionally, it provides channel sensing information to the MAC layer. These models implement the specific path loss model and a decider based on SINR model, implement BER rate according to modulation. (See section II/1) and (Cocorada, An IEEE 802.11g simulation model with extended debug capabilities, 2008))

This is implemented in the AdHocWiFiPhyLayer, AdHocWiFiDecider and AdHocWiFiPathLossModel classes.

e) *Network Media Access Control (MAC) Layer*

The MAC layer simulates the 802.11g media access control protocol, this includes carrier sensing, request to send and etc, this later keeps track on message encoding (MCS) rate and coordinates with the application layer for gathering statistics such as SINR.
This is implemented in the Mac80211g and AdHocWiFiPhyMac classes.

f) *Node*

Each node implements:

a. Scheduled transmission according to scheduler table of video data in different channels.
b. Receive and accumulate of packets into queues.
c. Dump of video meta-data at destination nodes in order to compose a video frame and (holding only RTP packet ID's number and time).
d. Flow control and queue drop.
e. Node movement according to scenario generator.
f. Handling of application/Mac addressing.
g. Collecting of statistics and messages log and saving to OMNET++ format.

This is implemented in the AdHocWiFiApplLayer, VideoStreamAppl and AdHocWiFiMobility classes.

g) *RTP dump playback*

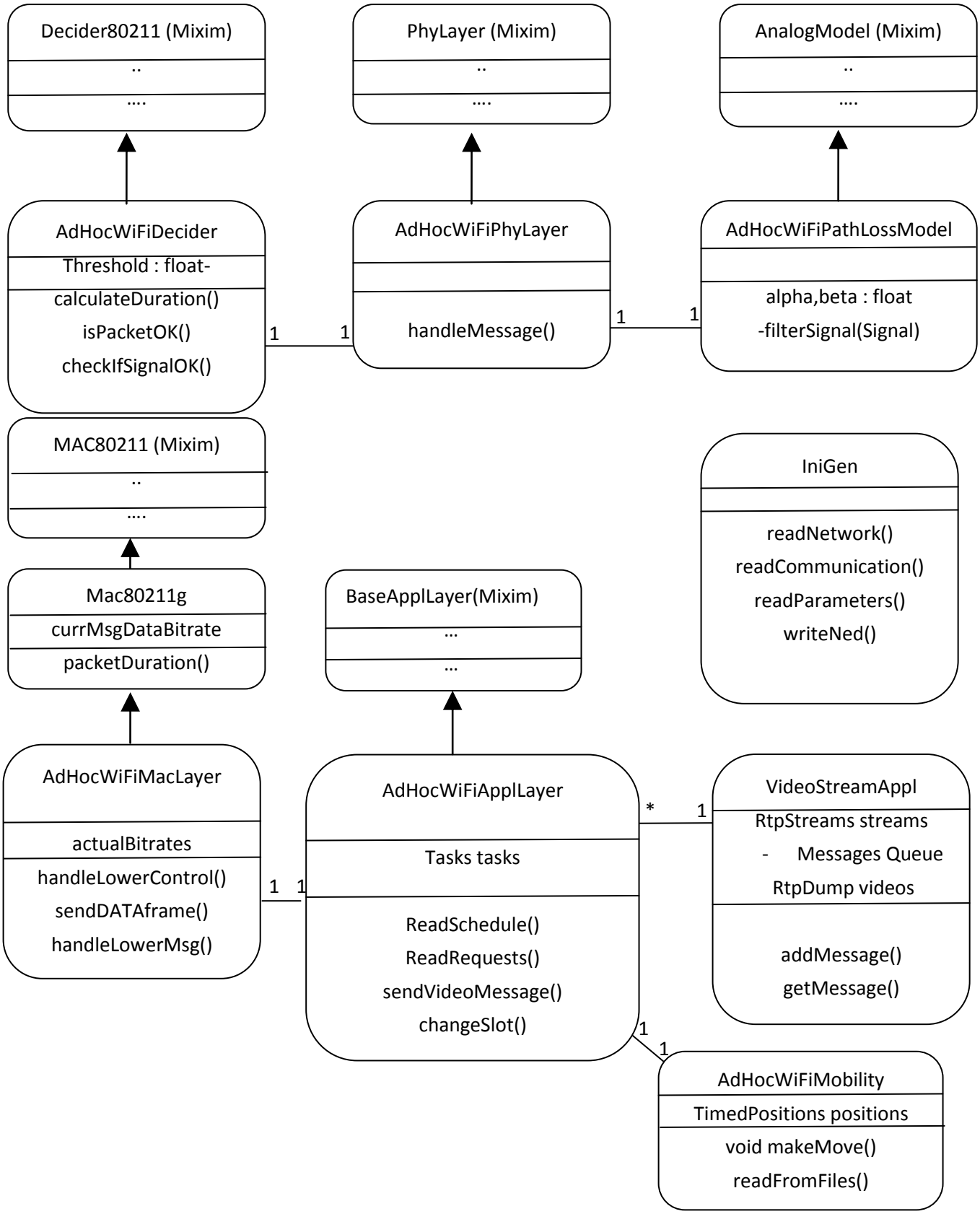
Enable playback of simulation results meta-video-data file and video RTP file into video player over local network.
This is implemented under the rtpreplay directory.

h) *Statistics processing*

Simulation results are processed for needed statistics using Matlab and needed graphs are created from files which are created during simulation run, more specifically the simulation output files (See section IV/3) are processed in order to gain the output graphs (See section IV / 2 / 4)
This is implemented under the analyze directory/AdHocWiFi.m Matlab file.

Figure IV-3 Simulator UML class diagram

(see (UML, 2011))



Classes documentation

Note : Full documentation and UML diagrams for every class are available in HTML format under the file doxygen/index.html

AdHocWiFiApplLayer class – (File AdHocWiFiApplLayer.)*

Application layer (per channel) of node

Performs send/receive to channel and communicates with high-level node application which stores messages in queues per stream.

Methods:

```
void initialize(…) - initialize object with stage of initialization
void finish() – finish run , delete objects and store end to end statistics
void setCurrentTask(const ScheduleItemSPtr task) – set current active transmission task of node
void txOver() – mark transmission of last message as over
unsigned getInitiatorMessage(int stream, int slot, long bits) - return an initiator (camera) message
void cancelLastInitiatorMessage() - cancel last message requested to send
unsigned int getMsgLength() const - return message length
int getNodeID() const - return ID of node
ScheduleItemSPtr getArbitrarlyTask() - perform weighted arbitration on current tasks
void transmitMessage() - sent a message from active tasks
void transmitMessage(ScheduleItemSPtr task) - sent a specific message per task
virtual void handleSelfMsg(cMessage* msg) - handle control message
virtual void handleLowerMsg(cMessage* msg) - handle MAC message
void changeSlot() - change slot in scheduling table
static void changeRate(int stream, int rate, int fps) - change rate per stream
static int getMessageSizeBits() - return message size in bits
static double getSlotPart() - return part of slot remaining from current time
```

void finishGlobal() - finish simulation and process statistics

Members:

bool active - is current application active (sending) currently

bool hasTask - has transmission task in current slot

ActiveTasks currentTasks - current scheduling task, if multiple then can perform weighted arbitration (non normal mode), holds pair of sent messages per task and the task itself

int channel, nodeID - WiFi channel of node and node id

int currSlot - current slot

int queueSampleOffset - offset (part of slot) for sampling queue size statistics in one slot mode

int currRound - current scheduling iterative round

cMessage *slotMsg, *transmitMsg - internal control messages - change slot and route message

Int2MessageCollection inputControlMessages - internal control message for getting input (camera) message

static map< **int**,map< **int**, **int** > > networkAddresses - network addresses map from node Index,channel to address

ScheduleItemCollection::Stream2SchedulesMap tasks - tasks of node/channel

IntSet initiatorStreams - streams which are initiators (sources) for this node

static ApplicationsMap applications - collection of high level applications map from id

int cnt, prevSlotCnt, idleSlots - cnt is last message id of transmit,prevSlotCnt is sample of cnt and idleSlots is number of slots in which there was no send

static unsigned long msgID - message id overall counter

static map<int,ofstream*> finalTimes - final times per stream map output file (for video playback)

static cOutVector * schedFromVec,* schedToVec,* schedSlotVec,* schedChannelVec,* schedStreamVec,*schedDataVec,*schedMcsVec - vectors for storing schedule table

static vector<MessagesStatistics> messagesStatistics - statistics of all messages(SINR,send time, receive time)

static RequestsCollection *requestsReader – requests collection for all stream

static Int2IntMap streamDestinations, streamSources - collection of stream destinations and sources

static double slotMaxTime - max time of any slot

static int slots - number of slots in schedule

static double roundTime - time of round, usually slotMaxTime*slots however can be different in special mode

static double slotSamplePart - slot sample part for queue size if in one slot

static int rounds - number of total rounds to repeat scheduling table iteratively

static int channels - number of frequency channels

static ScheduleItemCollection *schedule - schedule table

static bool rtpStatistics - if true store RTP statistics

static bool enableArbitration - if true enable arbitration between queues when sending

static bool oneSlot - if true all entries in table would be scheduled every slot

static Mode mode - mode of transmission (raw,video)

static StatisticsMode statisticsMode - mode of statistics (part/full)

static int msgSizeBits - message size in bits

AdHocWiFiApplLayer /AdHocWiFiMsg struct – a meta video message

Members:

```
int stream - stream message belongs to
int channel - channel message was send on
int rate - MCS of message
int from - source node ID
int to - destination node ID
int slot - slot number in scheduling table in which
message was send
int cnt - ID of message
static ofstream log - log for messages
```

Methods:

```
void recordLog(…) const - record log for message
```

VideoStreamAppl (File : VideoStreamAppl.)*

Top application layer for node.

Contains the sending AdHocWiFiApplLayer objects which are used for communication per channel (frequency).

Manages messages in the queues per stream and flow control.

Manages the video meta-data.

Methods:

```
void  addChannel(…) -  gathers messages to queues
and stores statistics,add a channel lower
application
void  checkAndSetTask(…) - route message in node
void  initialize() – initialize,called from
AdHocWiFiApplLayer
void  report(…) - report in end of round
void  reportStreams(…) - report in end of round for
all streams
void  endOfRound(…) - perform statistics gathering
and flow control
void  initStartRound(…) - initialize parameters for
start of round
void  finish() - called upon finish of simulation -
store statistics and perform cleanup
void  setRtpFiles() - define the names  of the RTP
file base bane and number
void  addMessage() – add a received message to
applications queue
ApplPkt * getMessage(…)  - get message from queue
int getNodeID(...) const - return id of node
void  finishSerial(...)  - finish simulation for
node, release buffers and collect statistics
static void addSINR(…) – add SINR statistics for
message
void  sampleQueueLength(…) – sample the queue length
for statistics
void  addIncomingTask(…) - add an incoming task
which node is supposed to receive data from
void  checkAndCreateStream(…)- check and create a
stream struct
```

Members:

```
map< int, AdHocWiFiApplLayer*> channels - map
between channel number and channel-application
object
int nodeID - ID of node
Int2TasksMap nodeTasks - map between slot number and
task
static vector<VideoStreamAppl*> apps - all video-
stream applications collection (by nodeID index)
RtpPerRate * initiatorData - pointer to initiator
video RTP data
unsigned currInitRate -  current initiator bit rate
unsigned currVidMsg - current video message id in
RTP stream
unsigned currInitFPSindex - current frame per second
for RTP video index
Streams streams - Streams collection data per this
node/application
```

VideoStreamAppl /RtpStream struct - RTP stream data for sources (camera)

Methods:

```
int getMsg() – return a new message
void  advanceTime(…) - advance time of streaming
pool
```

Members:

```
RtpData * video - video data
double bpsRate  - bit per second of video
double totalTime - total time of video
double totalSize - total size of video
static InitiatorsData initiatorsData - initiators
(camera) data for applications per video bps/fps
rates
```

VideoStreamAppl /Stream struct - represent a stream member of application, holds queue

Methods:

```
void  addMessage(…) - add a message to incoming queue
ApplPkt * getMessage(…) – get a message from queue
unsigned long phantomRequests(…) - return number of remaining phantom messages to send if queues were full in this slot (i.e. P+ )
void  flowControl(…) - activate flow control tasks and send back flow control message
void  queueDrop(…) - drop from queue according to rate
void  finishSerial(…) – end simulation , release memory and gather statistics
void  addSINR(…) – add SINR data for packet statistics
void  changeRate(…)  - change transmission rate from queue – adjust video camera rate
```

Members:

```
InEdgesCollection inEdges – edges incoming to node in stream
OutEdgesCollection outEdges – edges outgoing from node in stream
unsigned streamNumber – number of stream
MessagesDeque incomingMessages – queue of messages in stream
```

VideoStreamAppl /Stream/OutEdge struct - An edge in the out stream of node

Members:

```
int to – destination node of edge
unsigned int rate - bits out allowed on edge according to scheduler/flow control
unsigned int maxRate - bits out allowed from scheduler
unsigned int roundOut - rate in the current round already transmitted
```

Methods:

```
unsigned long getTotalRoundRate() - return round output rate including phantoms
```

VideoStreamAppl /Stream/InEdge struct - An edge in the in stream of node

Members:

```
int from – source of edge
```

Methods:

```
unsigned long getTotalInRate()- return total incoming data rate in round
```

Mac80211g class – enhancement for MiXiM to support 802.11g (File Mac80211g.) , see (Cocorada, OMNET++/MiXiM 802.11g)*

Methods:

```
virtual void initialize(…) – initialize model with parameters
simtime_t packetDuration(…) - computes the duration of a transmission over the physical channel, given a certain bitrate
```

Members:

```
double currMsgDataBitrate - current message data bitrate
```

AdHocWiFiMac class – Media Access Layer representation (File AdHocWiFiMac.)*

Members:

```
static long long bitrates[9] – bit rates of 802.11g
AdHocWiFiApplLayer * appl – application pointer
```

Methods:

```
virtual void initialize(…) – initialize object
static unsigned int getActualBitrate(…) - return
actual bitrate for PER=0 and given MCS and packet
length for 802.11g
double getCurrMsgDataBitrate() - return data bitrate
for current message
virtual void  sendDATAframe(…) - send data frame
according to protocol
virtual void  handleUpperMsg(…) - handle control
message from application/network
virtual void  handleLowerMsg(…) -  handle message
from Phy
virtual void  handleLowerControl(…) - handle lower
control message from Phy
virtual void  beginNewCycle() – begin new message
cycle
```

AdHocWiFiPhy class – Physical layer representation (File AdHocWiFiPhy.)*

Members:

```
virtual void  initialize(…) – initialize object with
parameter
virtual AnalogueModel* getAnalogueModelFromName(…) –
set Analogue (channel) model
virtual Decider* getDeciderFromName(…) – get decider
```

AdHocWiFiPathLossModel class – Channel path loss representation (File AdHocWiFiPathLossModel.)*

Implements the project path loss model (See section 0II / 2)

Members:

```
Move* hostMove – location and direction of the transmitter
double alpha,beta - model parameters
double elbit_c,elbit_htr,elbit_hrc - Elbit parameters
double elbit_additive - computation temporaries of Elbit macrocell formula
double elbit_dist_gain - computation temporaries of Elbit macrocell formula
```

Methods:

```
virtual void  filterSignal(…) - filter signal according to path loss model
virtual void  initialize(…) – initialize object with parameter
```

AdHocWiFiPathLossModelMapping class – Channel path loss representation for specific signal (File AdHocWiFiPathLossModel.)*

Members:

```
AdHocWiFiPathLossModel* model - Pointer to the model to get parameters from
double distance – distance between source and destination
Signal& signal – signal of transmission
```

Methods:

```
virtual double getValue(…) const – get strengths of signal after path loss
```

AdHocWiFiDecider class – model deciding if packet can be received (File AdHocWiFiDecider.)*

Methods:

```
virtual DeciderResult* checkIfSignalOk(…) – check if signal is OK according to SINR random model (See (Cocorada, OMNET++/MiXiM 802.11g))
double calculateDuration(…) - calculate duration of frame on air in seconds
```

AdHocWiFiMobility class – implementing node location and mobility over time (File AdHocWiFiMobility.)*

Methods:

```
virtual void  initialize(int stage);
static double getUpdateInterval() - return global
update interval (schedule round repeat time)
virtual void  fixIfHostGetsOutside() – fix location
if node gets outside playground
virtual void  makeMove() – make a move of node
bool readFromFile(…) - read locations from file for
a certain time
static bool readFromFiles(…) – read locations from
files for all times
```

Members:

```
int index - index in timed positions
int ID - ID of host
static TimedPositions timedPositions – all nodes
positions over time
static double sUpdateInterval - update interval -
schedule round time
```

RtpData class – implementing Real Time Protocol file reader and packet handler (File RtpData.) – see (Schulzrinne, RTP Tools)*

Members:

```
 FILE *in - input file to read from
 int first - time offset of first packet
 vector<RD_buffer_t*> buffers – RTP buffers
(packets) collection
 vector< int > sizes – sizes of packets
 vector< int > times – transmission times of packets
(mili-seconds)
 unsigned long totalSize – total data size of stream
 bool buffersLoad – whether to load actual data or
only size and times
```

Methods:

```
RtpData(…) – read from RTP dump file
int playHandler(…) – read next packet from file and
add to buffers
int getSizes(int i) – get packet size in index i
int getSizes() – return total number of packets
int getTimes(int i) – return transmission time of
packet i
int getTotalSize() – return total data size in
stream
void  addBuffer(RD_buffer_t * buffer) - add packet
data to stream
```

V. Bibliography

(n.d.). Retrieved from UML: http://www.uml.org

Avin, C., Emek, Y., Kantor, E., Lotker, Z., Peleg, D., & Roditty, L. (2008). SINR Diagrams:Towards Algorithmically Usable SINR Models of Wireless Networks. Tel Aviv, Isreal.

Cocorada, S. (2008). An IEEE 802.11g simulation model with extended debug capabilities. *1st international conference on Simulation tools and techniques for communications, networks and systems & workshops.* Brussels.

Cocorada, S. (n.d.). *OMNET++/MiXiM 802.11g.* Retrieved from http://vega.unitbv.ro/~sorin.cocorada/omnetpp/

erightsoft. (2011, January 1). *SUPER ©*. Retrieved from erightsoft: http://www.erightsoft.com/SUPER.html

Even, G., Fais, Y., Medina, M., Shahar, S. (., & Zadorojniy, A. (2011). *Real-Time Video Streaming in Multi-hop Wireless Static Ad Hoc Networks.* Tel Aviv University, Faculty of Engineering, Tel Aviv.

IEEE. (2003). *802.11g-2003*. Retrieved from IEEE Standards: http://standards.ieee.org/getieee802/download/802.11g-2003.pdf

IETF. (n.d.). *RFC 2250,3550,3551,3016,3640,3984*. Retrieved from IETF RFC: http://tools.ietf.org/html/

John G., A., Wai- tian, T., & Susie J., W. (2002, September 18). *Video Streaming: Concepts, Algorithms.* Retrieved from HP: http://www.hpl.hp.com/techreports/2002/HPL-2002-260.pdf

Köpke, A., Swigulski, M., Wessel, K., & Willkomm, D. (2008). Simulating Wireless and Mobile Networks in OMNeT++ - The MiXiM Vision. *1st international conference on Simulation tools and techniques for communications, networks and systems & workshops.*

Kumar, P. G. (2000, March). The capacity of wireless networks. *IEEE Transactions on Information Theory* , pp. 46(2):388–404.

live555. (2010, January 1). *LIVE555 Streaming Media*. Retrieved from LIVE555: http://www.live555.com/liveMedia/

MiXiM. (n.d.). Retrieved from MiXiM: http://mixim.sourcefourge.org

OMNET++. (n.d.). Retrieved from OMNET++: http://www.omnetpp.org

OMNET++ project publications. (n.d.). Retrieved from OMNET++ project publications: http://www.omnet-workshop.org

Perl. (2011, 04 01). *Perl*. Retrieved from The Perl Programming Language: www.perl.org

Ramesh, C. D. (2009, March 11). Capacity Characterization of Multi-Hop Wireless Networks-A Cross Layer Approach. Blacksburg, Virginia, U.S.

Schulzrinne, H. (n.d.). *RTP News*. Retrieved March 10, 2011, from Columbia university: http://www.cs.columbia.edu/~hgs/rtp/

Schulzrinne, H. (n.d.). *RTP Tools*. Retrieved March 10, 2011, from Columbia university: http://www.cs.columbia.edu/irt/software/rtptools/

Scoblete, G. (n.d.). Retrieved from A Beginner's Guide to Camcorder Bit Rates: http://camcorders.about.com/od/camcorders101/a/guide_to_bit_rates.htm

Scoblete, G. (n.d.). Retrieved from A Beginner's Guide to Camcorder Bit Rates: http://camcorders.about.com/od/camcorders101/a/guide_to_bit_rates.htm

Sommer, C., Dietrich, I., & Dressle, F. (2009). Simulation of Ad Hoc Routing Protocols using OMNeT++ - A Case Study for the DYMO Protocol. *Mobile Networks and Applications , 15* (6), pp. 786-801.

T. Moscibroda, R. W. (2006, November). Protocol design beyond graph-based models. *ACM SIGCOMM Workshop on HotNets* .

UML. (2011, January 1). *UML*. Retrieved from UML: http://www.uml.org

VLC. (n.d.). Retrieved from Video Lan: http://www.videolan.org

VLC. (2011, January 1). *VLC*. Retrieved from Video Lan: http://www.videolan.org

Wessel, K., Swigulski, M., Köpke, A., & Willkomm, D. (2009). MiXiM – The Physical Layer An Architecture Overview. *2nd International workshop on OMNeT++.* Rome.

Zadorojniy, A., Even, G., & Moni, S. (2009). *Frequency and Time Slot Assignment Algorithm.* Tel Aviv University, Tel Aviv.

Zadorojniy, A., Even, G., & Shahar, M. (2009). *Frequency and Time Slot Assignment Algorithm.* Tel Aviv University, Tel Aviv.

Zimmermann, H. (1980). OSI reference model–The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications* , pp. 28(4):425–432.

zytrax. (n.d.). Retrieved from Telecom and Network Speeds: http://www.zytrax.com/tech/data_rates.htm

VI. Appendix

1) Tools/environment build

In OMENT++ the simulation is configured using a file named “omnetpp.ini”, since there are additional configurations needed from the Scheduler and Mobility-Generator application then a small application called “IniGen” is created on-order to create a configuration for the simulator.

This IniGen creates a file named omnetpp.ini.include which is then included from the main omnetpp.ini , this file contains the network structure and related parameters.

- ### IniGen compilation

The IniGen is a simple project for creating network topology and initial parameters for the simulator.

The IniGen uses the same source files as the Scheduler project.

This project is placed inside the IniGen directory and in order to build it one can use the “make” command from shell.

The IniGen directory should be placed under the following directory:

<OMNET-BASE-DIR>/samples/MiXiM/examples/AdHocWiFi

- ### First time OMNET++/MiXiM environment build instructions

Checkout the repository omnetpp tar and unpack it to a directory and move into this directory (this is referred later as <OMNET-BASE-DIR>) in a command line shell

Run mingw.env from command-line

Run: “`./configure`” and then “`make`”, this would build OMNET++ libraries (may take several minutes)

Place the MiXiM directory from the repository into the <OMNET-BASE-DIR>/samples/MiXiM/ directory and move into this directory.

Run: “`make -f makemakefiles CONFIGNAME=gcc-debug`” and then “`make all`”, this would build the MiXiM libraries (may take several minutes)

Change into the <OMNET-BASE-DIR>/samples/MiXiM/examples/ directory, run: “`AdHocWiFi/gen_make_top.sh`”

Change into the <OMNET-BASE-DIR>/samples/MiXiM/examples/AdHocWiFi directory and run: “`./gen_make.sh`”

Run from shell: “`make`”

Change into the <OMNET-BASE-DIR>/samples/MiXiM/examples/AdHocWiFi/IniGen directory, start a cygwin shell and run: “make” (this step isn’t required since Windows compiled executable is included in the repository unless you want to compile this executable which isn’t part of MiXiM).

- ## AdHocWiFi Simulator compilation

The simulator AdHocWiFi directory should be placed under:

<OMNET-BASE-DIR>/samples/MiXiM/examples

Notice the OMNET++ make files are directory full-path dependent therefore one needs to regenerate them upon switching paths.

There are two scripts created for this:

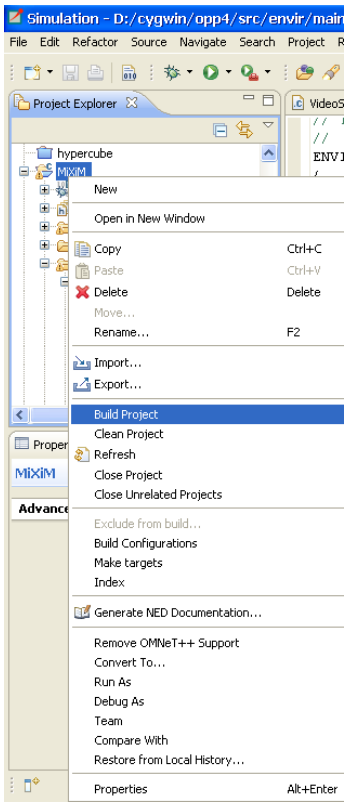
Run: “`AdHocWiFi/gen_make_top.sh`” from the <OMNET-BASE-DIR>/samples/MiXiM/examples directory

Run: “`gen_make.sh`” from the <OMNET-BASE-DIR>/samples/MiXiM/examples/AdHocWiFi directory

The simulator can then be compiled either inside OMNET++ (Eclipse) environment by using the “build” command or using “make” from shell inside the AdHocWiFi directory.

To open the simulator start a command line shell and change into the OMNET++ installation directory <OMNET-BASE-DIR>,run "`mingwenv.cmd`" , the shell would switch to MinGW shell (Linux like), now enter "`omnetpp` &" – The GUI should now start , you can confirm the location of the default workspace.

For building inside Eclipse right-click on the MiXiM project tree on the left panel and then select "Build project" as shown below:



- ## Rtp Tools compilation

Extract the rtptools*.gzip file and run "make" from Cygwin shell.

- ## Rtp Replay compilation

run "make" from Cygwin shell inside the AdHocWiFi/rtpreplay directory.

Note the above tools are available as compiled executables however you may need to compile them, also note the rest of the components (ScenarioGenerator,ScheduleCalculator) are also provided as compiled files however one can compile them from a VisualStudio project "build" command.

2) Scenario preparation

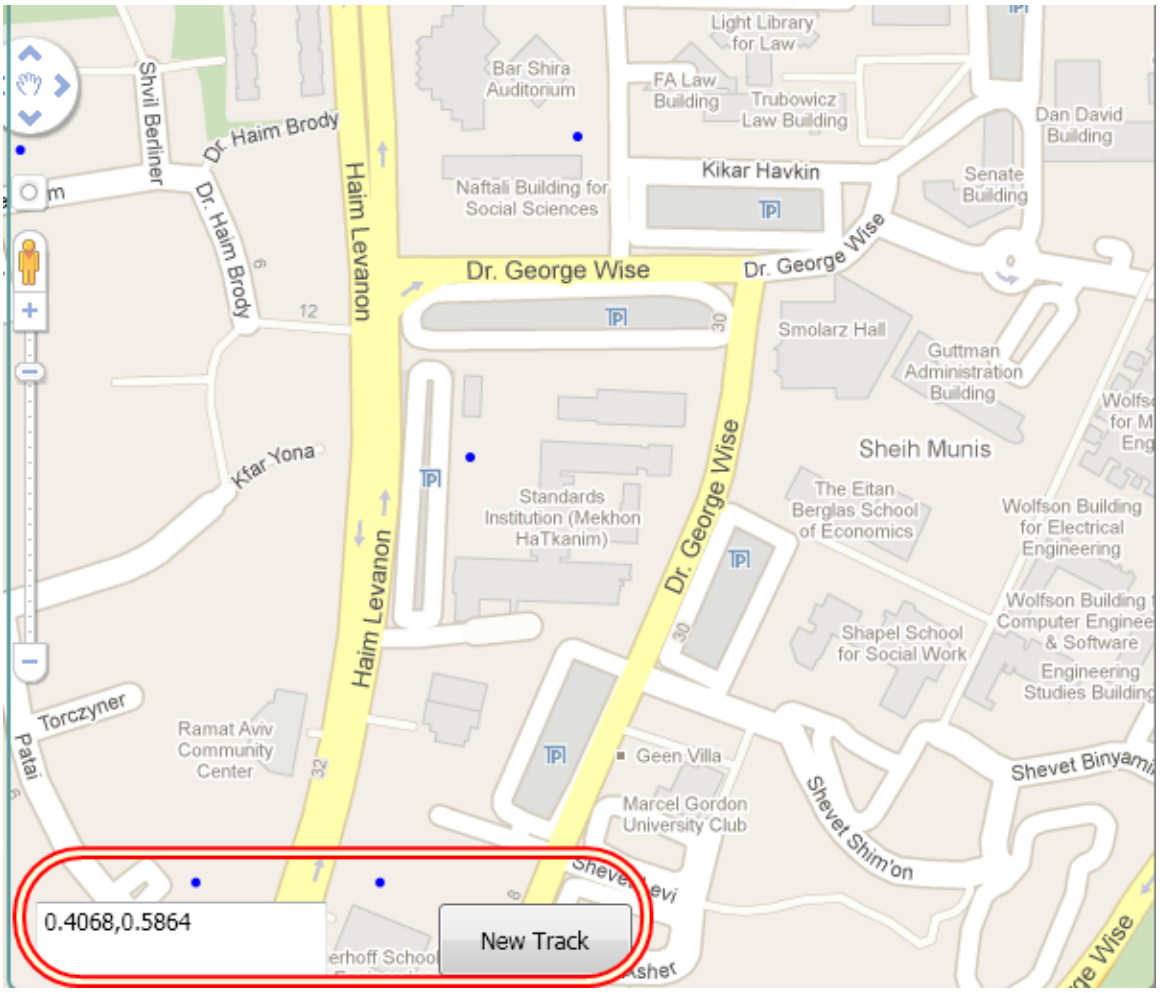
The simulator takes as input a scenario and schedule table, to prepare such scenario take the parameters text file as described in chapter VI / 7) and edit the parameters according to the comments in the file.

Then run the scenario mobility generator and scenario generator and the schedule calculator as described below.

- ### Running the scenario mobility generator

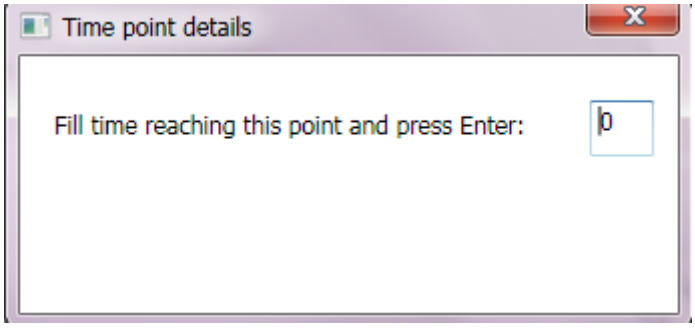
Run the executable: `WpfApp1.exe`

You should see an opaque window with the label "New Track" and coordinates bar in the bottom. You should see something similar to the following screen (with background according to your MS Windows background , it is therefore convenient to place a map behind this window as in the example):

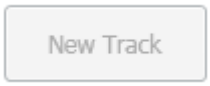


You can then click on a point in this window where every point means a location of a group of nodes in a certain time point, clicking on such multiple points creates a track, the coordinates are relative to the SCALE_IN_METERS parameter.

Whenever clicking on screen the following window would open:

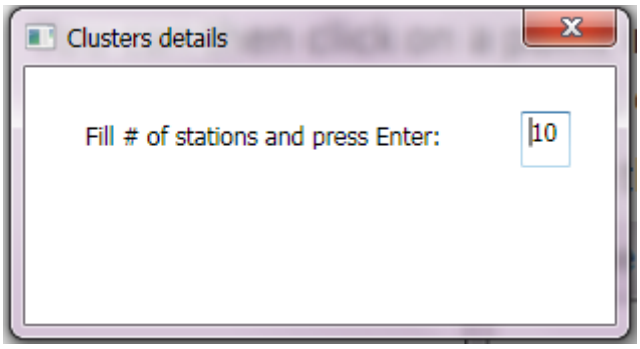


Enter the time in seconds where the nodes arrive to the neighborhood of this point.

Whenever finished with a track click on the 

button.

You should see a window as following:



You can then fill the number of nodes which participate in this track, this generator would place them randomly across this track.

You can fill additional tracks and when done close this application (By selecting "Close" after right clicking on the application from the task bar usually placed in the bottom of MS Windows screen).

The file points_metadata.txt would be created as output ,this file is used by the scenario generator and is used when placed in the same directory whenever running the scenario generator.

- ## Running scenario generator

The scenario generator is run using the command:

```
ScenarioGenerator.exe <parameters-file>
```

For example:

```
ScenarioGenerator.exe params1.txt
```

Where the parameters file is as specified in VI / 7) .

- ## Running scheduler

The scheduler calculator is run using the command:

```
SchedulerCalculator.exe <parameters-file>
```

For example:

```
SchedulerCalculator.exe params1.txt
```

Where the parameters file is as specified in VI / 7) .

3) Simulator usage

The simulator uses OMNET++/MiXiM standard components and builds the 802.11g and needed application specific layers.

Notice running the entire flow can be done simply by running the "benchmark" flow as described below (See Running benchmark) ,the following describes running individually the different steps.

Note: The implementation defines the AdHocWiFi NED package, this prefix is also used for all other implementation files, for simplicity running this package can be done by placing it inside MiXiM examples directory.

- ### IniGen run

Running the IniGen is done by running inside the IniGen directory the bin/gen.exe executable and passing the name of the main parameters file which is used by the scheduler (commonly called params1.txt)

The output of this process is a file named omnetpp.ini.include

For example run: `bin/gen.exe params1.txt`

Another option (recommended) is to add also a parameter for the base directory of the scenario files, in this case the tools assumed the file names inside the parameters files are inside this directory, for example if the scenario and schedule files are inside a directory named scenario1 one directory above the IniGen directory then run:

```
bin/gen.exe params1.txt ../scenario1
```

This would enable having the scenario and schedule files in one directory without any copying of files.

- ### Running the simulator basic test

The basic test contains two nodes where node "1" sends to node "2" 8 streams each in different MCS for 1/10 of a second, during this time node "2" gets farther from node "1" in 5meter/sec.

This test is under <OMNET-BASE-DIR>/samples/MiXiM/examples/scen/line8 and it should be run exactly as in the above instructions where TEST=../scen/line8 in the make command.

Moreover to generate the graphs change into the ../scen/line8 directory and run from MatLab/Octave the command:

per2snr('Main.csv','Main_log.csv')

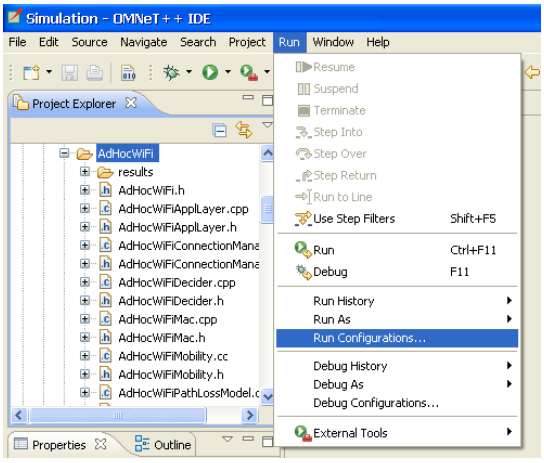
The basic test should show a graph of Throughput per MCS in the simulation compared to the analysis and also a graph of PER to SNR for different MCS.

- ## Simulator run

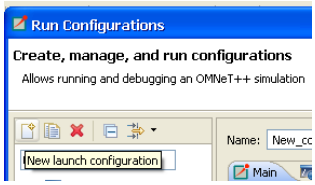
The simulation can be run either inside OMNET++ (Eclipse) environment by using the “run” command over the AdHocWiFi configurations or by command line using the “run.sh” from shell.

To setup and run from the Eclipse GUI:

Select "Run "Configurations…" from the "Run" menu



A new Dialog titled "Run Configurations" would open, click on the top-left button ("New launch configuration")



Fill in the details as following:

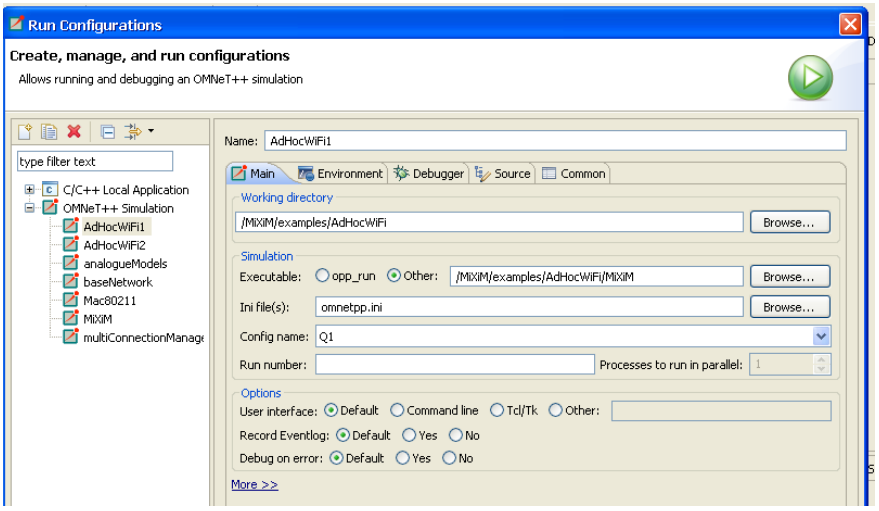
Name: AdHocWiFi1

Working directory: /MiXim/examples/AdHocWiFi

Executable: /MiXiM/examples/AdHocWiFi/MiXiM

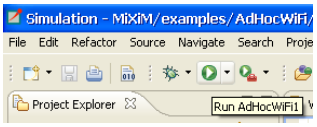
Ini File(s): omnetpp.ini

Config name: Main

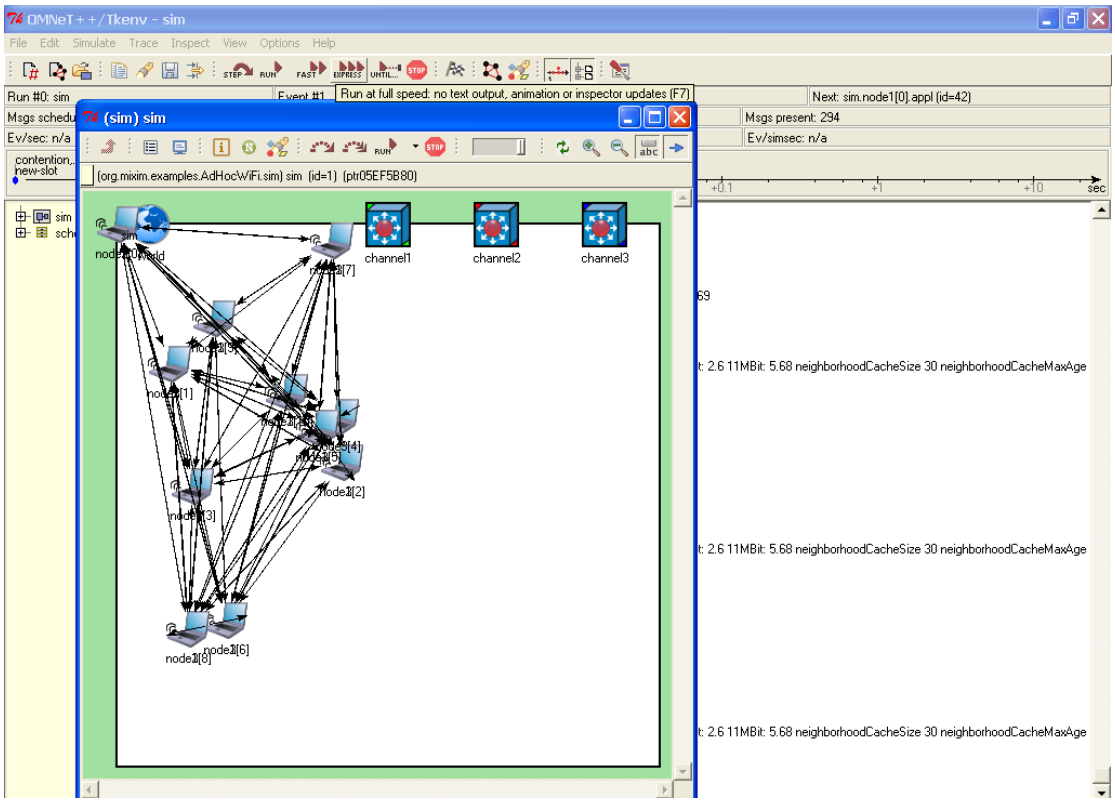


Click on the bottom-right "Run" button to run this configuration

Following runs can be done by clicking on the "Run" icon:



After starting the run the simulator GUI window would open similar to the following:



You can press the "Run" button for interactive mode but notice it may be slow, press the "Express" mode for full running speed.

Once simulation completes a pop-up message would show up, after closing the simulator windows the output files would be generated.

- ## Statistics collection

The statistics is gathered using the OMNET++ standard vector/scalar file format, this data is later processed using OMNET++ scavetool to produce the needed graphs, specifically comma-separated CSV text format is exported which later can be imported by spreadsheet tools to produce graphs, Additional graphing tools can be used to produce graphs from the OMNET++ format.

In order to generate this statistics use the "`make -C results gen_csv`" from the base directory (<OMNET-BASE-DIR>/samples/MiXiM/examples/AdHocWiFi) , the <CONFIG-NAME>.csv file would be generated inside the results directory.

This .csv file can be imported into common tools such as MS Excel by simply dragging it to the Excel window.

Additionally there are Matlab/Octave tools which process this data and described later on.

There are two .csv files which are generated for example for the main configuration:

results/Main.csv – general parameters

results/Main_log.csv – messages events log

- ## Command line full running flow for one scenario

The following assumes the first environment build is complete

Create a new directory under the <OMNET-BASE-DIR>/samples/MiXiM/examples , for example "scen/scenario1"

Run the scenario generator and scheduler in this directory.

The schedule, requests and coordinates file should be created.

Use a file named “params” as the parameter file for the scenario (as in chapter VI / 9).

Start MinGW command line shell by running mingwenv.cmd inside the <OMNET-BASE-DIR> from either command line or windows explorer.

Change into the AdHocWiFi/analyze directory (<OMNET-BASE-DIR>/samples/MiXiM/examples/AdHocWiFi/analyze) and run:

"`make -C ../results TEST=../scen/scenario1/ COPY=1`"

This would run the simulation in batch mode and produce a file named Main.csv and Main_log.csv copied to test directory (../scen/scenario1)

Create the graphs by running from within Matlab/Octave inside the <OMNET-BASE-DIR>/samples/MiXiM/examples/AdHocWiFi/analyze directory: (you can point to the full path of the .csv files)

“AdHocWiFi(‘../results/Main.csv’,'../results/Main_log.csv')

This step would create image files with the figures.

- ## Running benchmark

A benchmark utility was created for running the full flow with different scenarios and different configurations, overall this is the simplest way for running the entire flow.

Configuring this run is done inside the "AdHocWiFi/analyze" directory in a file named "benchmark.prl" (with Perl syntax, see (Perl, 2011)), this file defines only two important variables: "scenarios" and "configurations" and is included by the main run script ("run_benchmark.prl").

The "scenarios" variable is an array where each element is a hash with the following fields: Dir , Generator, Scheduler, Simulator and Analyze.

"Dir" is a string pointing to a directory where the "params" file which defines the simulation parameters exists (See section VI / 9) and the other parameters correspond to the flow steps according to their name and each such field variable holds an integer value where any value other than "1" causes ignore of step and "1" means running the step.

Note the ability to run only some steps is useful for example for manually "generating" scenarios or for running the analysis step on computers which don't have Matlab.

The following example code defines two scenarios where all the steps are being run:

```
@scenarios = (
   {
   Dir => "../../experiments/circle_25_1000_41_5_6d",
   Generator => 1,
   Scheduler => 1,
   Simulation => 1,
   Analyze => 1,
   }
   ,
   {
   Dir => "../../experiments/grid_12_1000_41_5_9d",
   Generator => 1,
   Scheduler => 1,
   Simulation => 1,
   Analyze => 1,
   }
 );
```

The "configurations" variable is also an array of hashes holding two variables:

"Name" is the name of the configuration, this name corresponds to a simulation configuration in the omnetpp.ini (See section VI/9).

The "Params" variable is a string which is concatenated to the "params" file and defines the settings of the configuration.

For example the following code defines two configurations corresponding to the Main (MF-I-S) and ShortP simulation configurations.

```
@configurations  = (
   {
   Name => "Main",
   Params => "USE_SMART_SCHEDULER, int, 1\nROUTER_TYPE,
int, 4\nALLOW_ADDING_SLOTS, int, 0\n",
   }
   ,
   {
     Name => "ShortestPath",
     Params => "USE_SMART_SCHEDULER, int,
0\nROUTER_TYPE, int,    2\nALLOW_ADDING_SLOTS, int,
0\n",
   }
   );
```

After setting the benchmark parameters execute from shell (from the AdHocWiFi/analyze directory):

```
./run_benchmark.prl
```

Running this step may take even several hours (depending on the overall number of scenarios x configurations) and in the end it would generate the .csv raw statistics results and the graphs in picture formats of .jpg , .eps and .fig (Matlab).

The results would be placed inside the scenario directory where a new directory would be created for each configuration and separately for the different image formats.

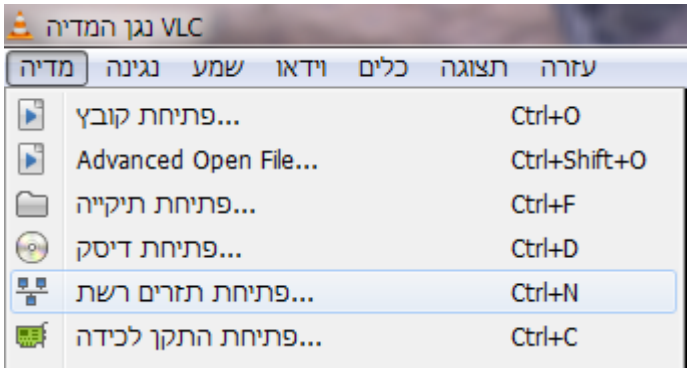
Moreover log files (with .log suffix) would be created for every step in the run flow, for example for the above benchmark the following structure would be created: (underlines files meaning they are part of the input), notice file formats are specified in VI8).

- CommGraph.txt – text file with the network graph of edges , output of the generator
- Coords_000.000.txt – text file with coordinates of nodes for time 0 sec
- Coords_001.000.txt - text file with coordinates of nodes for time 1 sec
- ..<Coords file for each second>..
- debug_comm_graph.csv – All edges in communication graph with SNR between nodes , output of generator
- generator.log – log file of generator run
- InterfGraph.txt – Interference graph between edges , output of generator
- params1.txt – original parameters file for benchmark
- params_gen.txt – parameters file used for running the generator
- points_metadata.txt – location of nodes in meta data (result of scenario locations generator, optional – if doesn't exist then created by generator)
- Requests_tmp.txt – Requests of scenario – optional (if not exists then used randomly by generator)
- Requests.txt – Requests of scenario (either a copy of Requests_tmp.txt or created by generator)
- snr2per.txt – *SNR2PER table used by generator*
- Main/ShortestPath – directory per configuration name
  - flow_summary.txt – scheduler results for with results of flow per stream
  - Main.csv.tgz – statistics results of simulator run (after compression with gZip utility)
  - Main_log.csv.tgz – messages log results of simulator (after compression with gZip utility)
  - omnetpp.ini.include – additional parameters for simulators from scenario , created by IniGen
  - params – parameters for scenario/configuration used by scheduler (created by benchmark)
  - queues_summary.txt – queue lengths as calculated by scheduler
  - scheduler.log – log of running the scheduler
  - schedule.txt – scheduler table text file
  - simulator.log – log of running the simulator

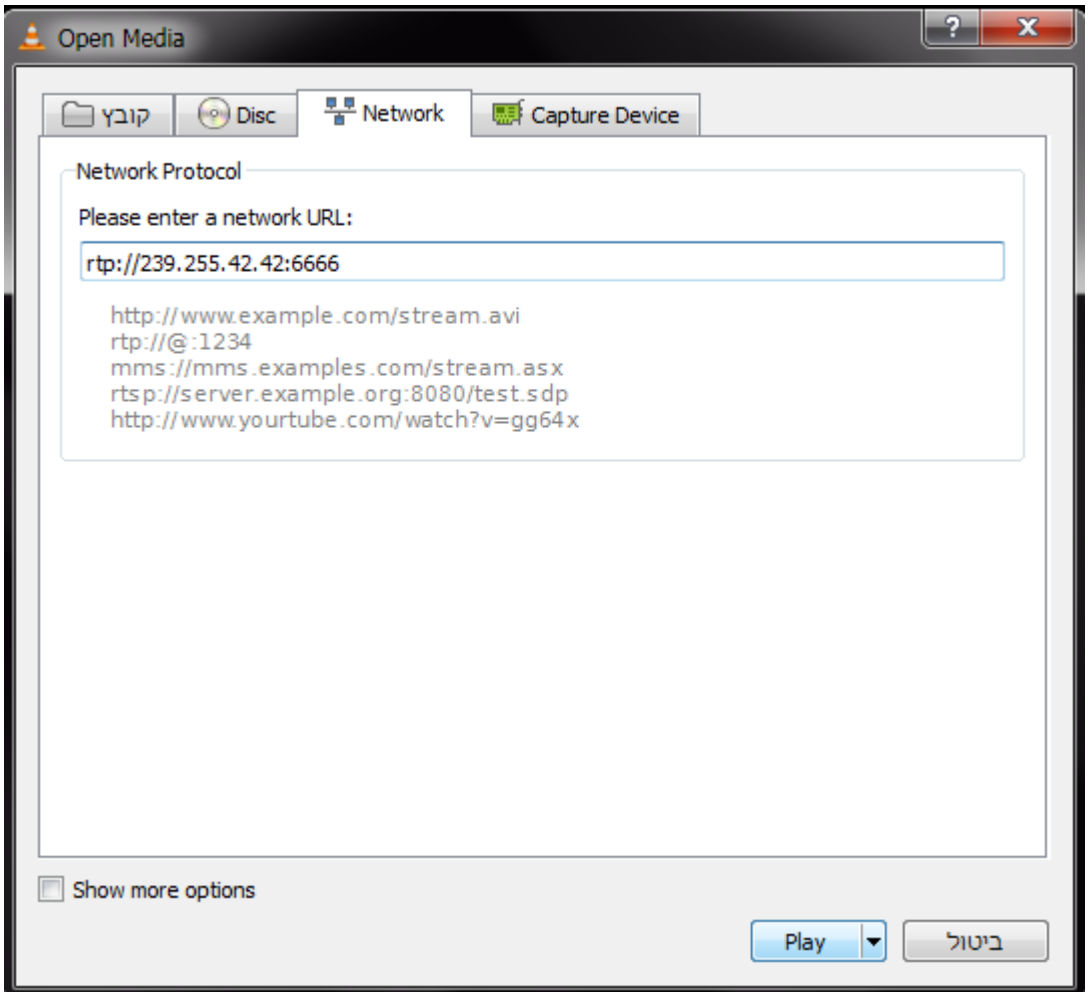
- jpg/eps/fig – directory per photo format name (all files in directory would be with this suffix)
  - dropped – dropped ratio per stream
  - e2e_delay_hist_multi_* – end to end delay histogram per stream (3 streams in one image)
  - e2e_delay_multi_* - end to end delay per stream per packet (3 streams in one image)
  - hops_number – number of hops per stream
  - max_drop_in_time – drop in time of stream with maximal drop percentage
  - per – packet error rate per stream
  - q_len_max_multi_* - max queue length size value per node/stream over time (3 streams per image)
  - q_len_mean_multi_* - average queue length size value per node/stream over time (3 streams per image)
  - scenario – scenario nodes location and routing
  - sinr_loss_high – (SNR - SINR) value per scheduling table entry (10 high values)
  - sinr_loss_lower - (SNR - SINR) value per scheduling table entry (10 low values)
  - sinr_loss_median - (SNR - SINR) value per scheduling table entry (10 median values)
  - table_PER – packet error rate for each scheduling table entry
  - throughput – throughput per stream over time
  - tp_vs_sched – throughput per stream compared to scheduler results

Running the video results

Run the VLC client, and select Media->Open media from the menu (מדיה>פתח תזרים רשת):



Select the IP / port to play from as following:



Click the "Play" button.

Run the rtpreplay utility using the simulation results (available only when working in "video" mode) and the RTP dump files, change into the AdHocWiFi/rtpreplay directory and run from Cygwin shell:

```
./rtpplay -f <RTP-file-prefix> -t <simulation-video-result-file-per-stream> 239.255.42.42/6666
```

For example:

```
./rtpplay -f ../vids/wildlife -t ../results/node31_str8.txt 239.255.42.42/6666
```

You should now see the video playing in the VLC client, for example:

This can be compared to the original video, for example:

Dependencies:

OMNET++

The simulator depends on OMNET++ and was tested with version 4.1

To download follow http://www.omnetpp.org/ and check out the installation guide.

After extracting to a local directory (referred above as <OMNET-BASE-DIR>) you need to change to this directory and run from a command line run:

```
./configure
make
```

MIXIM++

The wireless model library used is MiXiM version 1.1, To download follow http://mixim.sourceforge.net/ and check the installation guide.

Extract the downloaded file into the <OMNET-BASE-DIR>/samples directory

MiXiM would be built upon built of the simulator the first time.

Note: In order to extend MiXiM to 802.11g model the following file needs to be changed:

<OMNET-BASE-DIR>/samples/MiXiM/modules/mac/Mac80211.h

<OMNET-BASE-DIR>/samples/MiXiM/modules/mac/Mac80211.cc

<OMNET-BASE-DIR>/samples/MiXiM/modules/utility/Consts80211.h

These file should be updated according to the files released to affect changes and updates done in order to support 802.11g rates suitable for our model (only 802.11g nodes).

Matlab

Matlab can be used when analyzing simulation results.

Matlab is a commercial tool from MathWorks.

Cygwin/Octave

Cygwin is required in order to compile on windows shell, it isn't required for running

Octave can be used to generate graphs of simulation results instead of matlab.

To install Cygwin follow the instructions from http://cygwin.com , when installing include also the packages: make,gcc,xinit,xterm,octave

When running octave run within the X-Cygwin terminal

Changing simulation parameters:

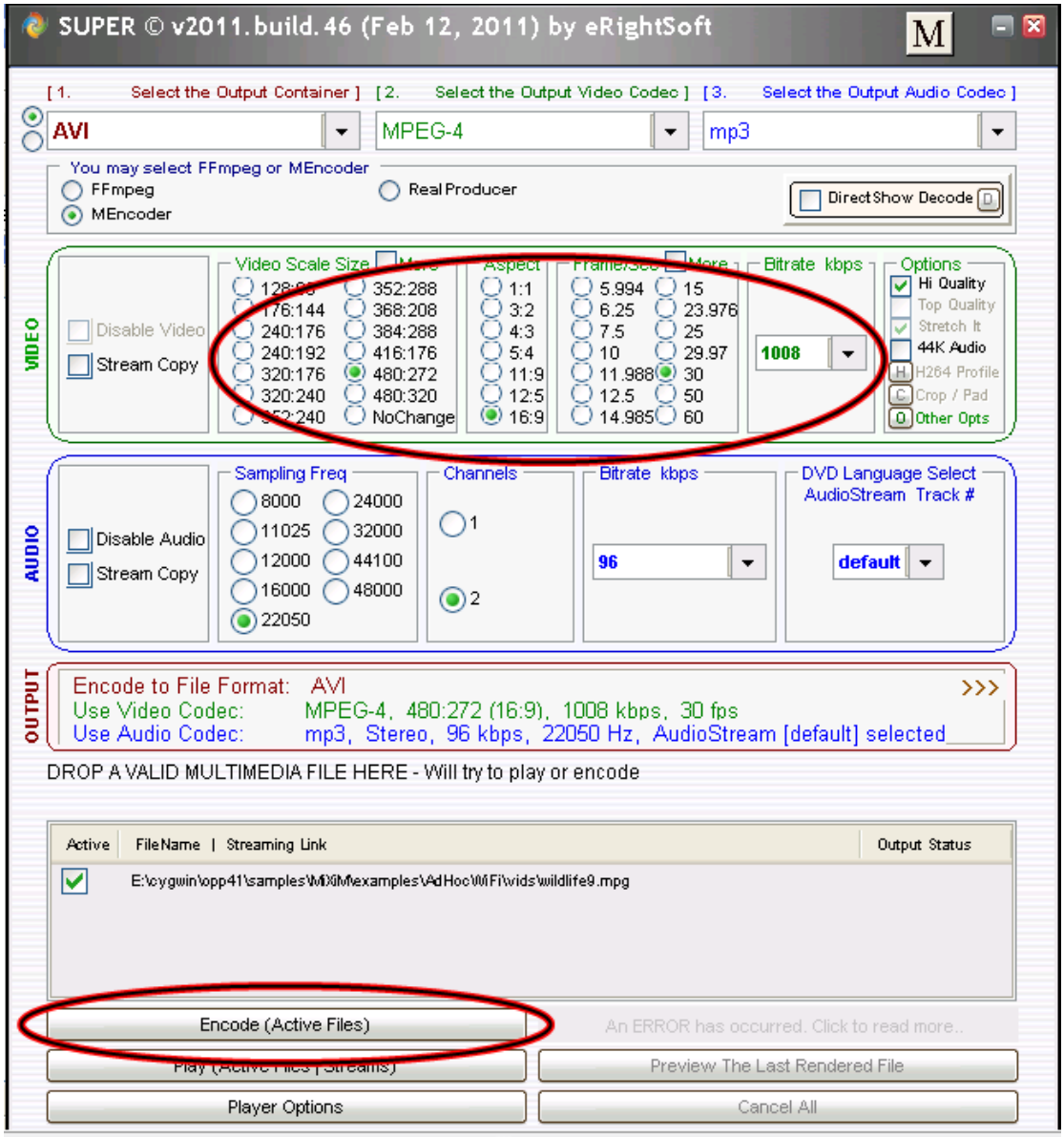
Some parameters are related to the simulation only and are places inside the omnetpp.ini file inside the AdHocWiFi directory.

For example changing between transmission of video data or raw data is changed by setting the parameter "mode" of the application between "raw" and "video" (See section VI / 9).

Notice the parameters of the scenario are processed by the IniGen and are placed inside the IniGen/omnetpp.ini.include file.

Encoding a video with different bit rates

Start Super and drag to the window the video file from the file manager, you should have a window screen which looks as following:



Change the encoder settings (Frame rate, Bit rate) for both video and audio and press the "Encode (Active Files)" button.

The encoding should start and may take several minutes, repeat this process several times while changing the encoder settings.

You should have a different *.avi file in the output directory.

Creating RTP file from Video file

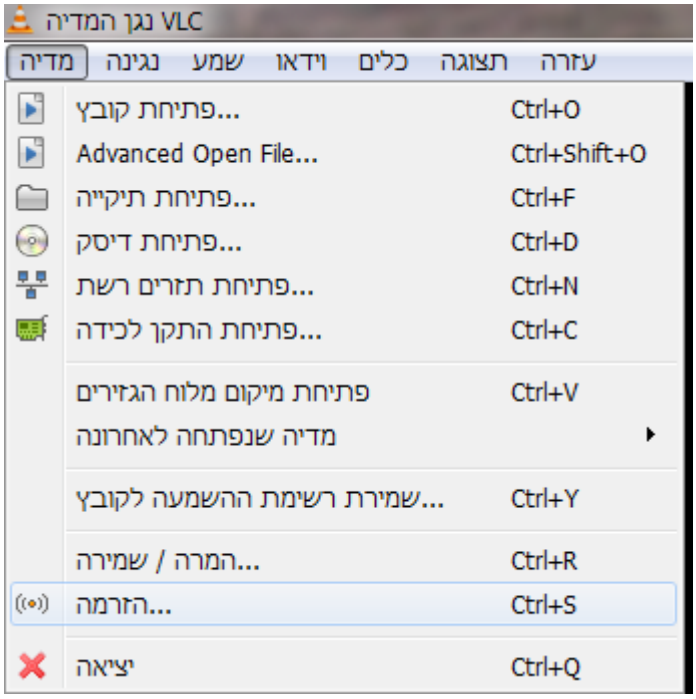
Run the rtpdump utility from the RTP Tools (See (Schulzrinne, RTP Tools)) using the following command line:

```
./rtpdump  -F dump -o <RTP-output-file>
239.255.42.42/6666
```

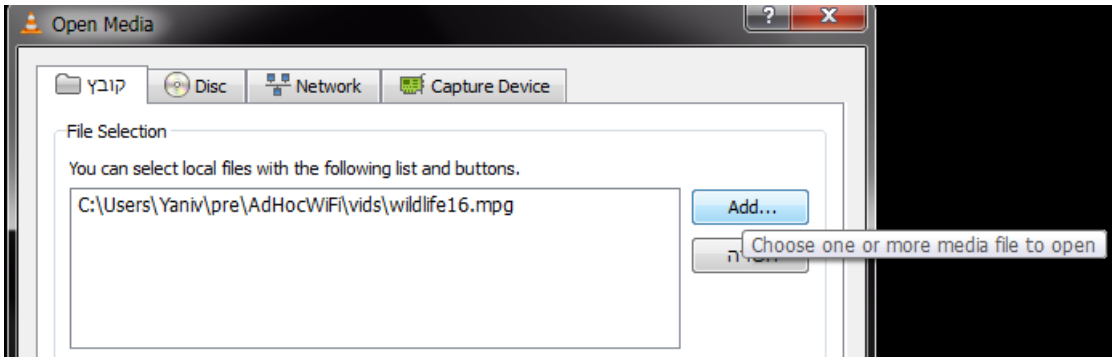
For example:

```
./rtpdump  -F dump -o wildlife1.rtp 239.255.42.42/6666
```

Start VideoLan (VLC) tool (See (VLC, 2011)) , select Media->Stream (מדיה-> הזרמה) from the menu as following:

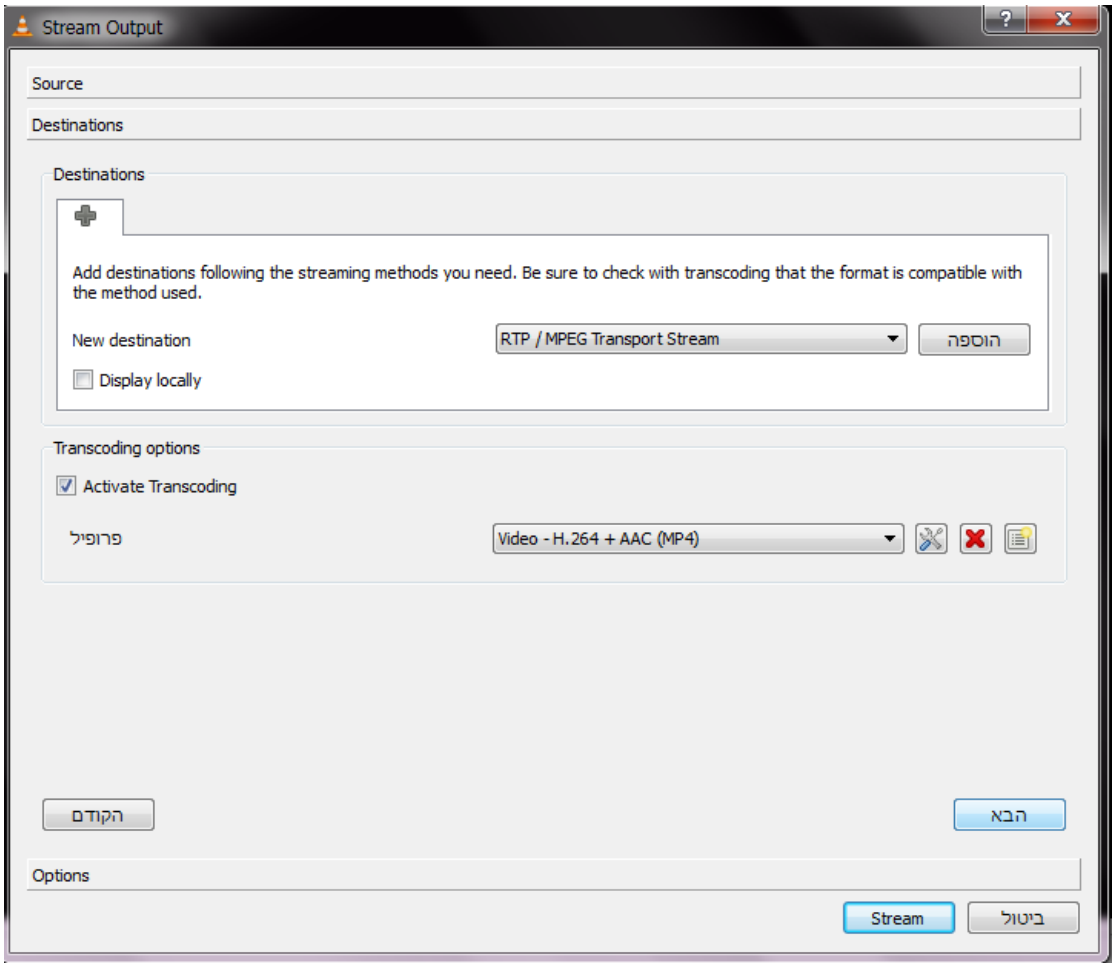


Choose the video file which was created in previous step (encoding) by clicking "Add" and selecting the file from the file manager:

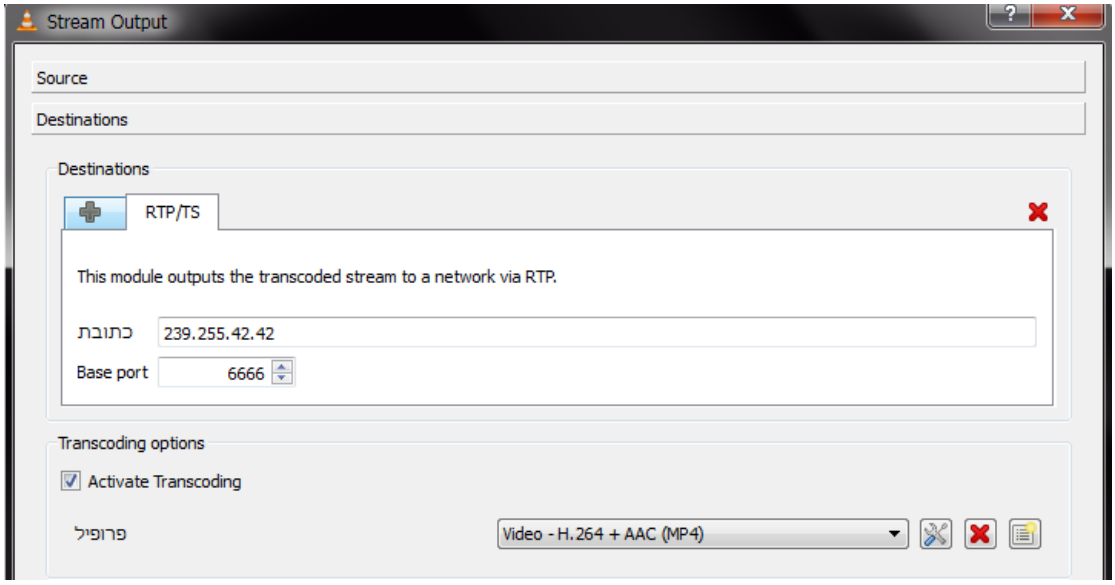


Click on the "Stream" button in the bottom of the page:

Select the stream options as shown in the following screen: RTP/MPEG Transport Stream and Video-H.264+AAC(MP4):



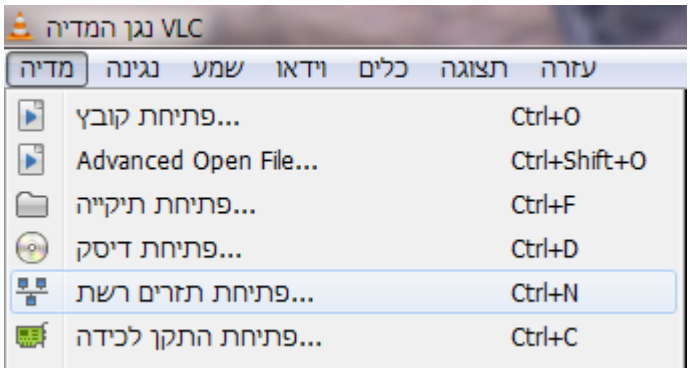
Click on the "Add" and set the IP address and port as following:



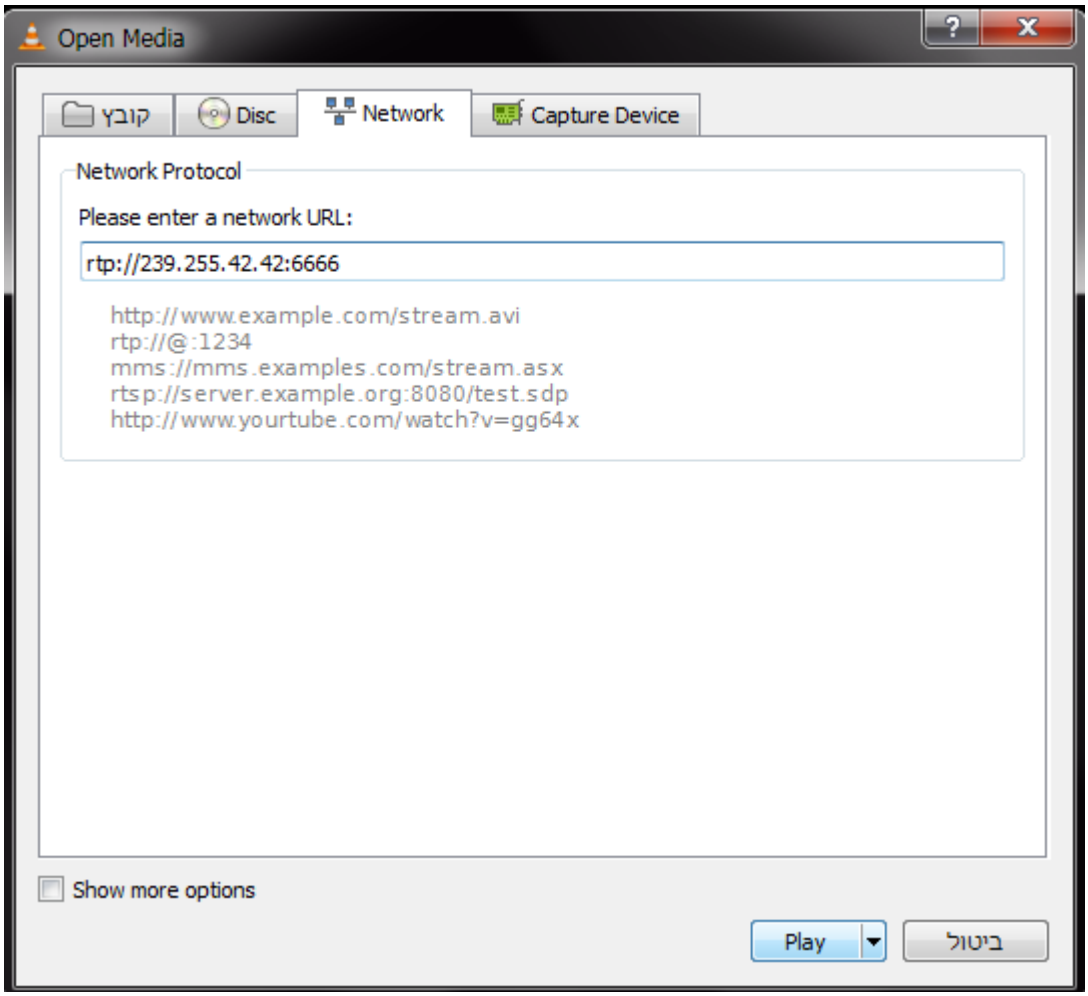
Click on "Stream" , you should see VLC progressing the video time, once complete you can stop the "rtpdump" (by Ctrl-C) , you can see the output RTP file being created.

Checking the RTP file created correctly

Run the VLC client, and select Media->Open media (מדיה->פתיחת תזרים רשת) from the menu:



Select the IP / port to play from as following:



Click the "Play" button.

Run the rtpplay utility on the created RTP dump file created in previous step as following:

```
./rtpplay -f <RTP-file> 239.255.42.42/6666
```

For example:

```
./rtpplay -f wildlife1.rtp 239.255.42.42/6666
```

You should now see the VLC playing the video file from the network stream according to the RTP file which was dumped.

4) Simulator log results example:

The following log shows an example of a wireless network in the 802.11 protocol with 10 nodes and a packet which is being send from node 1 to node 4.

```
** Event #1780  T=3.841289221357  sim.host[1].net (BaseNetwLayer, id=26), on
`BROADCAST_REPLY_MESSAGE' (ApplPkt, id=495)
in encaps...
CInfo removed, netw addr=50
 netw 26 sending packet
sendDown: get the MAC address
host[0]::BaseArp: for host[4]: netwAddr 50; MAC address 51
 pkt encapsulated
** Event #1781  T=3.841289221357  sim.host[1].nic.mac (Mac80211, id=28), on
`BROADCAST_REPLY_MESSAGE' (NetwPkt, id=1037)
Mac80211::handleUpperMsg BROADCAST_REPLY_MESSAGE
CInfo removed, mac addr=51
pkt encapsulated, length: 1328
packet (NetwPkt)BROADCAST_REPLY_MESSAGE received from higher layer,
dest=51, encapsulated
state IDLE --> CONTEND
[Host 1] - PhyLayer(Decider): Creating RSSI map.
3.841289221357 do contention: medium = [idle with rssi of 1e-11], backoff = 0
** Event #1782  T=3.841289221357  sim.host[1].nic.phy (PhyLayer, id=29), on
`contention' (ChannelSenseRequest, id=8)
** Event #1783  T=3.841317221357  sim.host[1].nic.phy (PhyLayer, id=29), on
selfmsg `contention' (ChannelSenseRequest, id=8)
[Host 1] - PhyLayer(Decider): Creating RSSI map.
** Event #1784  T=3.841317221357  sim.host[1].nic.mac (Mac80211, id=28), on
`contention' (ChannelSenseRequest, id=8)
3.841317221357 handleLowerControl contention
 Mac80211::sendRTSframe duration: 0.000058666666 br: 6e
 Mac80211::timeOut RTS 0.000119334332
state CONTEND --> WFCTS
** Event #1785  T=3.841317221357  sim.host[1].nic.mac (Mac80211, id=28), on
`{Radio switching over}' (cMessage, id=1039)
3.841317221357 handleLowerControl Radio switching over
** Event #1786  T=3.841317221357  sim.host[1].nic.phy (PhyLayer, id=29), on `wlan-
rts' (Mac80211Pkt, id=1040)
AirFrame encapsulated, length: 352
host[1]::PhyLayer: sendToChannel: sending to gates
** Event #1787  T=3.841317221357  sim.host[2].nic.phy (PhyLayer, id=37), on `{}'
(AirFrame, id=1042)
host[2]::PhyLayer: Received new AirFrame with ID 59 from chann
PhyLayer(SimplePathlossModel): sqrdistance is: 18500
PhyLayer(SimplePathlossModel): wavelength is: 0.124292
PhyLayer(SimplePathlossModel): distance factor is: 1.85028e-11
```

Node 1 sending message: network layer,MAC and Phy

Node 1 : sending RTS frame, waiting for CTS

Node 2 : receiving frame , path loss is computed and signal is strong enough

[Host 2] - PhyLayer(Decider): Processing AirFrame...
[Host 2] - PhyLayer(Decider): Signal is strong enough (3.14739e-11 > 1.12202e-12) -> Trying to receive AirFrame.
…….
** Event #1794 T=3.841317221357 sim.host[9].nic.phy (PhyLayer, id=93), on `{}' (AirFrame, id=1041)
host[9]::PhyLayer: Received new AirFrame with ID 59 from channel
PhyLayer(SimplePathlossModel): sqrdistance is: 39200
PhyLayer(SimplePathlossModel): wavelength is: 0.124292
PhyLayer(SimplePathlossModel): distance factor is: 4.12105e-12
[Host 9] - PhyLayer(Decider): Processing AirFrame...
[Host 9] - PhyLayer(Decider): Signal is strong enough (7.01006e-12 > 1.12202e-12) -> Trying to receive AirFrame.
host[9]::PhyLayer: Handed AirFrame with ID 59 to Decider. Next handling in 0.000058666666s.
**** Event #1795 T=3.841375888023 sim.host[1].nic.phy (PhyLayer, id=29), on selfmsg `{transmission over}' (cMessage, id=53)**
**** Event #1796 T=3.841375888023 sim.host[1].nic.mac (Mac80211, id=28), on `{Transmission over}' (cMessage, id=1049)**
3.841375888023 handleLowerControl Transmission over
PHY indicated transmission over
transmission of packet is over

Node 1 : sending RTS frame complete

** Event #1797 T=3.841375888023 sim.host[1].nic.mac (Mac80211, id=28), on `{Radio switching over}' (cMessage, id=1050)
3.841375888023 handleLowerControl Radio switching over
** Event #1798 T=3.841375888023 sim.host[2].nic.phy (PhyLayer, id=37), on selfmsg `{}' (AirFrame, id=1042)
[Host 2] - PhyLayer(Decider): Processing AirFrame...
[Host 2] - PhyLayer(Decider): Creating RSSI map excluding AirFrame with id 59
snrMin: 3.11888
berHeader: 1.52733e-05 berMPDU: 2.11603e-10
packet was received correctly, it is now handed to upper layer...
host[2]::PhyLayer: Decapsulating MacPacket from Airframe with ID up to MAC.
host[2]::PhyLayer: End of Airframe with ID 59.

Node 2 : SINR is computed, random decode model : packet is received correctly but destination isn't node 2

**** Event #1799 T=3.841375888023 sim.host[2].nic.mac (Mac80211, id=36), on `wlan-rts' (Mac80211Pkt, id=1051)**
handleLowerMsg frame (Mac80211Pkt)wlan-rts received
updated information for neighbor: 27 snr: 3.11888 bitrate: 1.2e+07
handle msg not for me wlan-rts
NAV timer started, not QUIET: 0.000384666665
state IDLE --> QUIET
cannot beginNewCycle until NAV expires at t 3.841760554688
** Event #1800 T=3.841375888023 sim.host[3].nic.phy (PhyLayer, id=45), on selfmsg `{}' (AirFrame, id=1043)
[Host 3] - PhyLayer(Decider): Processing AirFrame...
[Host 3] - PhyLayer(Decider): Creating RSSI map excluding AirFrame with id 59

snrMin: 1.7079
berHeader: 0.00168475 berMPDU: 3.42015e-06
packet was received correctly, it is now handed to upper layer...
host[3]::PhyLayer: Decapsulating MacPacket from Airframe with ID 59 and sending it up to MAC.
host[3]::PhyLayer: End of Airframe with ID 59.
** Event #1801 T=3.841375888023 sim.host[3].nic.mac (Mac80211, id=44), on `wlan-rts' (Mac80211Pkt, id=1052)
handleLowerMsg frame (Mac80211Pkt)wlan-rts received
updated information for neighbor: 27 snr: 1.7079 bitrate: 9e+06
handle msg not for me wlan-rts
NAV timer started, not QUIET: 0.000384666665
state IDLE --> QUIET
cannot beginNewCycle until NAV expires at t 3.841760554688
** Event #1802 T=3.841375888023 sim.host[4].nic.phy (PhyLayer, id=53), on selfmsg `{}' (AirFrame, id=1044)
[Host 4] - PhyLayer(Decider): Processing AirFrame...
[Host 4] - PhyLayer(Decider): Creating RSSI map excluding AirFrame with id 59
snrMin: 4.80805
berHeader: 5.47776e-08 berMPDU: 2.20915e-15
packet was received correctly, it is now handed to upper layer...
host[4]::PhyLayer: Decapsulating MacPacket from Airframe with ID 59 and sending it up to MAC.
host[4]::PhyLayer: End of Airframe with ID 59.
**** Event #1803 T=3.841375888023 sim.host[4].nic.mac (Mac80211, id=52), on `wlan-rts' (Mac80211Pkt, id=1053)**
handleLowerMsg frame (Mac80211Pkt)wlan-rts received
updated information for neighbor: 27 snr: 4.80805 bitrate: 1.8e+07
handle msg for me wlan-rts in IDLE

Node 4 receives RTS packet

** Event #1804 T=3.841375888023 sim.host[4].nic.phy (PhyLayer, id=53), on `{end SIFS}' (ChannelSenseRequest, id=24)
** Event #1805 T=3.841375888023 sim.host[5].nic.phy (PhyLayer, id=61), on selfmsg `{}' (AirFrame, id=1045)
[Host 5] - PhyLayer(Decider): Processing AirFrame...
[Host 5] - PhyLayer(Decider): Creating RSSI map excluding AirFrame with id 59
snrMin: 0.407194
berHeader: 0.128676 berMPDU: 0.0371289
Packet has BIT ERRORS! It is lost!
packet was not received correctly, sending it as control messag
host[5]::PhyLayer: End of Airframe with ID 59.

Node 5 : SINR is computed, random decode model : packet is not received correctly (SINR too small)

** Event #1806 T=3.841375888023 sim.host[5].nic.mac (Mac80211, id=60), on `ERROR' **(Mac80211Pkt, id=1054)**
3.841375888023 handleLowerControl ERROR
handle msg not for me ERROR
staying in state IDLE
** Event #1807 T=3.841375888023 sim.host[6].nic.phy (PhyLayer, id=69), on selfmsg `{}' (AirFrame, id=1046)

……..
** Event #1815 T=3.841385888023 sim.host[4].nic.phy (PhyLayer, id=53), on selfmsg `{end SIFS}' (ChannelSenseRequest, id=24)
[Host 4] - PhyLayer(Decider): Creating RSSI map.
**** Event #1816 T=3.841385888023 sim.host[4].nic.mac (Mac80211, id=52), on `{end SIFS}' (ChannelSenseRequest, id=24)**
3.841385888023 handleLowerControl end SIFS
Mac80211::sendCTSframe duration: 0.000050666666 br: 6e+
state IDLE --> WFDATA

Node 4 : sending CTS

** Event #1817 T=3.841385888023 sim.host[4].nic.mac (Mac80211, id=52), on `{Radio switching over}' (cMessage, id=1058)
3.841385888023 handleLowerControl Radio switching over
** Event #1818 T=3.841385888023 sim.host[4].nic.phy (PhyLayer, id=53), on `wlan-cts' (Mac80211Pkt, id=1059)
AirFrame encapsulated, length: 304
host[4]::PhyLayer: sendToChannel: sending to gates
** Event #1819 T=3.841385888023 sim.host[1].nic.phy (PhyLayer, id=29), on `{}' (AirFrame, id=1061)
host[1]::PhyLayer: Received new AirFrame with ID 60 from channel
PhyLayer(SimplePathlossModel): sqrdistance is: 14900
PhyLayer(SimplePathlossModel): wavelength is: 0.124292
PhyLayer(SimplePathlossModel): distance factor is: 2.85238e-11
[Host 1] - PhyLayer(Decider): Processing AirFrame...
[Host 1] - PhyLayer(Decider): Signal is strong enough (4.85201e-11 > 1.12202e-12) -> Trying to receive AirFrame.
host[1]::PhyLayer: Handed AirFrame with ID 60 to Decider. Next handling in 0.000050666666s.
** Event #1820 T=3.841385888023 sim.host[0].nic.phy (PhyLayer, id=21), on `{}' (AirFrame, id=1062)
host[0]::PhyLayer: Received new AirFrame with ID 60 from channel
PhyLayer(SimplePathlossModel): sqrdistance is: 36325
PhyLayer(SimplePathlossModel): wavelength is: 0.124292
PhyLayer(SimplePathlossModel): distance factor is: 4.7992e-12
……..
3.841436554689 handleLowerControl Radio switching over
** Event #1830 T=3.841436554689 sim.host[1].nic.phy (PhyLayer, id=29), on selfmsg `{}' (AirFrame, id=1061)
[Host 1] - PhyLayer(Decider): Processing AirFrame...
[Host 1] - PhyLayer(Decider): Creating RSSI map excluding AirFrame with id 60
snrMin: 4.80805
berHeader: 5.47776e-08 berMPDU: 2.20915e-15
packet was received correctly, it is now handed to upper layer...
host[1]::PhyLayer: Decapsulating MacPacket from Airframe with ID 60 and sending it up to MAC.
host[1]::PhyLayer: End of Airframe with ID 60.
**** Event #1831 T=3.841436554689 sim.host[1].nic.mac (Mac80211, id=28), on `wlan-cts' (Mac80211Pkt, id=1070)**

Node 1 : received CTS

handleLowerMsg frame (Mac80211Pkt)wlan-cts received
updated information for neighbor: 51 snr: 4.80805 bitrate: 1.8e+07
handle msg for me wlan-cts in WFCTS
** Event #1832 T=3.841436554689 sim.host[1].nic.phy (PhyLayer, id=29), on `{end SIFS}' (ChannelSenseRequest, id=9)
……
** Event #1847 T=3.841446554689 sim.host[1].nic.phy (PhyLayer, id=29), on selfmsg `{end SIFS}' (ChannelSenseRequest, id=9)
[Host 1] - PhyLayer(Decider): Creating RSSI map.
**** Event #1848 T=3.841446554689 sim.host[1].nic.mac (Mac80211, id=28), on `{end SIFS}' (ChannelSenseRequest, id=9)**
3.841446554689 handleLowerControl end SIFS
Mac80211::timeOut DATA 0.000314000999
sending DATA to 51 with bitrate 6e+06
state WFCTS --> WFACK

Node 1 : sending DATA, waiting for ack

** Event #1849 T=3.841446554689 sim.host[1].nic.mac (Mac80211, id=28), on `{Radio switching over}' (cMessage, id=1077)
3.841446554689 handleLowerControl Radio switching over
** Event #1850 T=3.841446554689 sim.host[1].nic.phy (PhyLayer, id=29), on `BROADCAST_REPLY_MESSAGE' (Mac80211Pkt, id=1078)
AirFrame encapsulated, length: 1520
host[1]::PhyLayer: sendToChannel: sending to gates
……
** Event #1866 T=3.841699888022 sim.host[4].nic.phy (PhyLayer, id=53), on selfmsg `{}' (AirFrame, id=1082)
[Host 4] - PhyLayer(Decider): Processing AirFrame...
[Host 4] - PhyLayer(Decider): Creating RSSI map excluding AirFrame with id 61
snrMin: 4.80805
berHeader: 5.47776e-08 berMPDU: 2.20915e-15
packet was received correctly, it is now handed to upper layer...

Node 4 : receiving DATA

host[4]::PhyLayer: Decapsulating MacPacket from Airframe with ID 61 and sending it up to MAC.
host[4]::PhyLayer: End of Airframe with ID 61.
** Event #1867 T=3.841699888022 sim.host[4].nic.mac (Mac80211, id=52), on `BROADCAST_REPLY_MESSAGE' (Mac80211Pkt, id=1091)
handleLowerMsg frame (Mac80211Pkt)BROADCAST_REPLY_MESSAGE received
updated information for neighbor: 27 snr: 4.80805 bitrate: 1.8e+07
handle msg for me BROADCAST_REPLY_MESSAGE in WFDATA
message decapsulated
** Event #1868 T=3.841699888022 sim.host[4].net (BaseNetwLayer, id=50), on `BROADCAST_REPLY_MESSAGE' (NetwPkt, id=1092)
handling packet from 26
** Event #1869 T=3.841699888022 sim.host[4].nic.phy (PhyLayer, id=53), on `{end SIFS}' (ChannelSenseRequest, id=24)
** Event #1870 T=3.841699888022 sim.host[4].appl (BurstApplLayer, id=48), on `BROADCAST_REPLY_MESSAGE' (ApplPkt, id=1093)
Received reply from host[1]; delete msg

** Event #1871 T=3.841699888022 sim.host[5].nic.phy (PhyLayer, id=61), on selfmsg `{}' (AirFrame, id=1083)
[Host 5] - PhyLayer(Decider): Processing AirFrame...
……
handle msg not for me ERROR
staying in state IDLE
**** Event #1881 T=3.841709888022 sim.host[4].nic.phy (PhyLayer, id=53), on selfmsg `{end SIFS}' (ChannelSenseRequest, id=24)**
[Host 4] - PhyLayer(Decider): Creating RSSI map.
**** Event #1882 T=3.841709888022 sim.host[4].nic.mac (Mac80211, id=52), on `{end SIFS}' (ChannelSenseRequest, id=24)**
3.841709888022 handleLowerControl end SIFS
sent ACK frame!
state WFDATA --> BUSY
**** Event #1883 T=3.841709888022 sim.host[4].nic.mac (Mac80211, id=52), on `{Radio switching over}' (cMessage, id=1098)**
3.841709888022 handleLowerControl Radio switching over
**** Event #1884 T=3.841709888022 sim.host[4].nic.phy (PhyLayer, id=53), on `wlan-ack' (Mac80211Pkt, id=1099)**
AirFrame encapsulated, length: 304
host[4]::PhyLayer: sendToChannel: sending to gates

Node 4 : sending ACK

** Event #1885 T=3.841709888022 sim.host[1].nic.phy (PhyLayer, id=29), on `{}' (AirFrame, id=1101)
host[1]::PhyLayer: Received new AirFrame with ID 62 from channel
PhyLayer(SimplePathlossModel): sqrdistance is: 14900
PhyLayer(SimplePathlossModel): wavelength is: 0.124292
PhyLayer(SimplePathlossModel): distance factor is: 2.85238e-11
[Host 1] - PhyLayer(Decider): Processing AirFrame...
[Host 1] - PhyLayer(Decider): Signal is strong enough (4.85201e-11 > 1.12202e-12) -> Trying to receive AirFrame.
host[1]::PhyLayer: Handed AirFrame with ID 62 to Decider. Next handling in 0.000050666666s.
** Event #1886 T=3.841709888022 sim.host[0].nic.phy (PhyLayer, id=21), on `{}' (AirFrame, id=1102)
……
** Event #1901 T=3.841760554688 sim.host[1].nic.phy (PhyLayer, id=29), on selfmsg `{}' (AirFrame, id=1101)
[Host 1] - PhyLayer(Decider): Processing AirFrame...
[Host 1] - PhyLayer(Decider): Creating RSSI map excluding AirFrame with id 62
snrMin: 4.80805
berHeader: 5.47776e-08 berMPDU: 2.20915e-15
packet was received correctly, it is now handed to upper layer.
host[1]::PhyLayer: Decapsulating MacPacket from Airframe with ID 62 and sending it up to MAC.
host[1]::PhyLayer: End of Airframe with ID 62.
**** Event #1902 T=3.841760554688 sim.host[1].nic.mac (Mac80211, id=28), on `wlan-ack' (Mac80211Pkt, id=1110)**

Node 1 : received ACK, random backoff

handleLowerMsg frame (Mac80211Pkt)wlan-ack received
updated information for neighbor: 51 snr: 4.80805 bitrate: 1.8e+07
handle msg for me wlan-ack in WFACK
3.841760554688 random backoff = 0.000126
[Host 1] - PhyLayer(Decider): Creating RSSI map.
3.841760554688 do contention: medium = [idle with rssi of 1e-11], backoff = 0.000126
state WFACK --> IDLE

5) 802.11g wireless model qualification

a) Raw rate / actual rate comparison

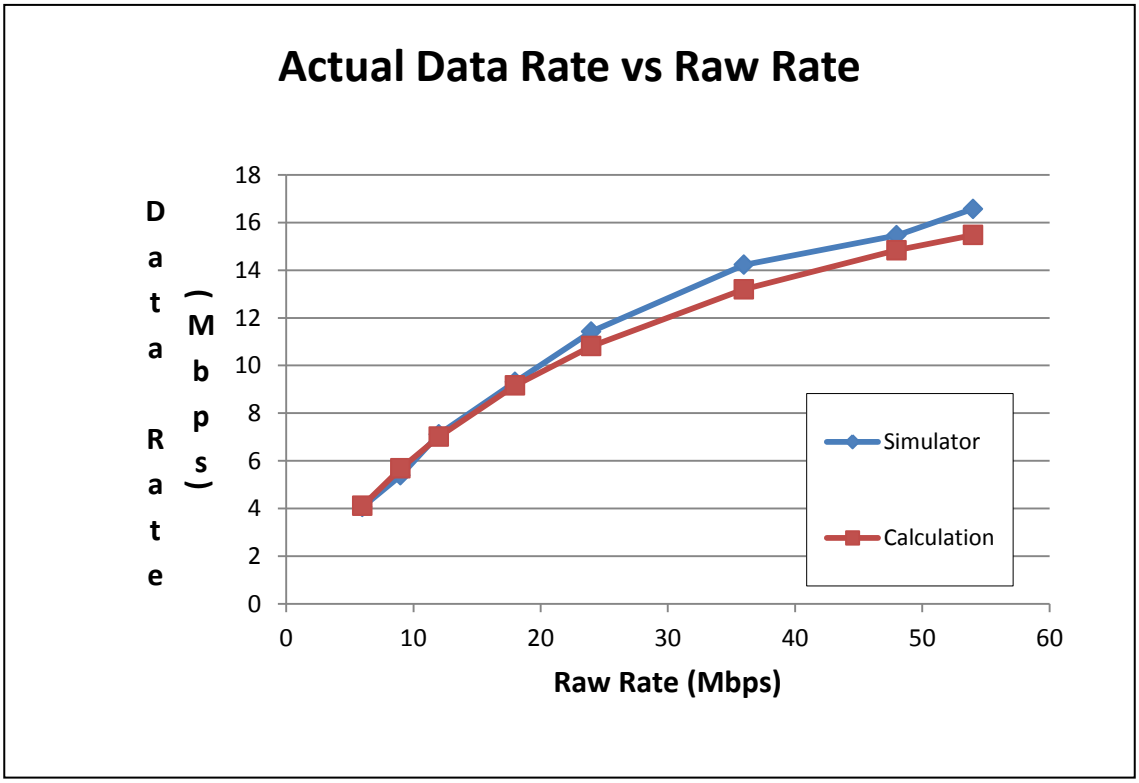


Figure VI-1 – Raw vs. Actual rate comparison

As can be seen there are only minor differences in simulator data rate compared to calculation (As appears in (Zadorojniy, Even, & Moni, Frequency and Time Slot Assignment Algorithm, 2009)) . These differences are results of the simulator not modeling preamble which adjusts the frame size.

b) SNR to PER

The following graph depicts the packet error rate to the signal to noise ratio results of the simulator model in different modulation coding scheme (MCS) as expressed by different transmission rates of the 802.11g protocol rates from 6 to 54 Mbps.

It can be seen from this graph , as expected , that once the coding rate increases it is more vulnerable to noise, for example when transmitting in 54Mbps then at SNR of 19.65dB there is almost no errors however when transmitting when the SNR rate decreases to 17.16dB (in the simulator this is the result of larger distance) the packet error rate increases to almost 1 (meaning almost all packets are lost).

When transmitting in 6Mbps we can see that in SNR rate of 2.39dB there are almost no errors however in SNR of 1.36dB the packet error rate is 1, this graph is known as the "waterfall" of coding since in minor change of SNR we can see large difference of PER between almost zero to almost one.

This variance in PER to SNR between the different coding schemes allows transmitting in varying distances (since larger distance means smaller SNR) but means decreasing transmission rate, the results of the simulator are used by the scheduling algorithm when calculating the data which is being sent in each slot.

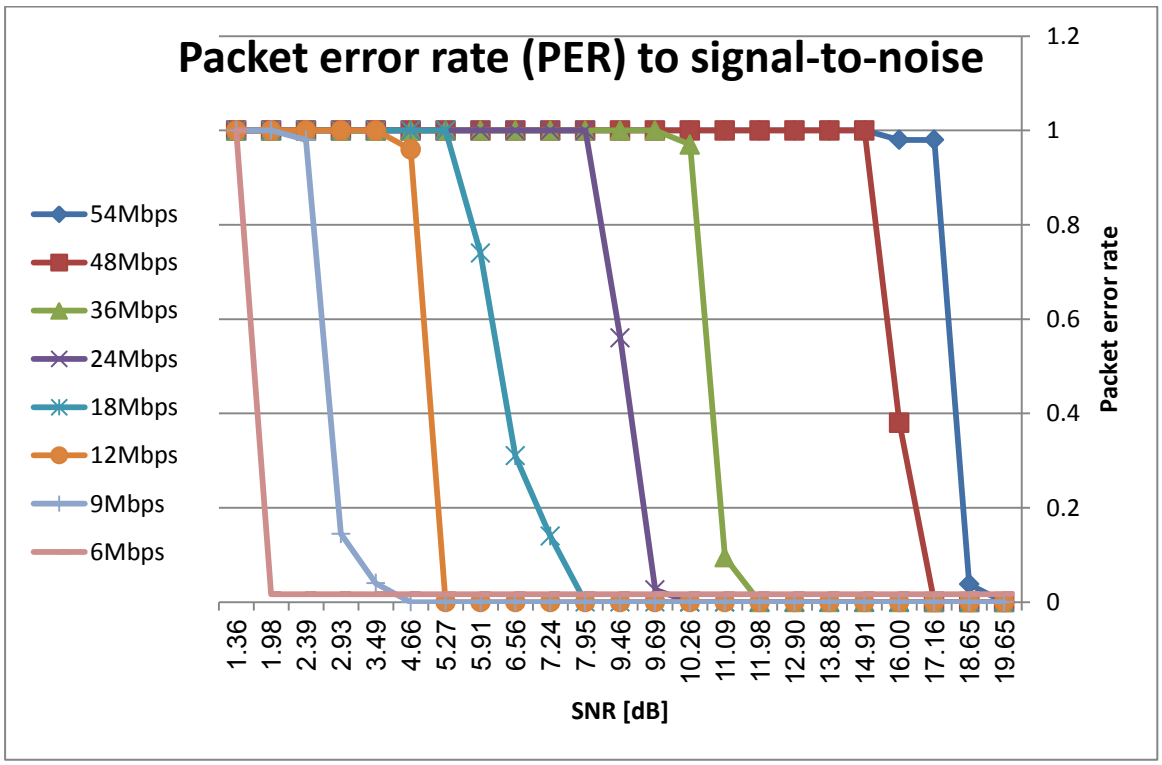


Figure VI-2 PER to SNR rate

6) Simulator output files format

- A messages Log file is used for the statistics analysis, it is a text file with a single header line as comment and lines of text where each line corresponds to a message event where the fields are separated by commas, the fields of such event are :

Message ID , from , to , stream , slot , channel , SNIR , Event ID , Time

The encoding of the event ID is :

0 : packet receive

1 : packet send

2 : packet lost

3 : packet drop

- An RTP Dump file is a text file which is used in order to replay simulation results in a real time video player using the rtpreplay utility (See section IV / 7 / 2).

  This file holds video stream meta-data for a variable rate video stream composed of a single video file encoded into multiple RTP files (See section IV / 71) , the meta-data includes the packet ID and arrival time at destination (respectively playback time) in the following format:

  <Packet ID> @<Time>

  Moreover whenever the source changes a video rate this line would appear:

  #<video-rate-ID>_<Frame-Per-Seconds>

  Any of the above lines can appear multiple times in the file.

The rtpreplay utility would read all the RTP files and transmit the RTP packets from the selected video rate file using the packet ID and arrival time in order to replay the simulation results.

Notice this file is created only when simulator is used in "video" mode.

7) Generic input parameters

```
// output files   - added per configurations in run
script
COMMUNICATION_GRAPH_FILENAME, string, CommGraph.txt
INTERFERENCE_GRAPH_FILENAME, string, InterfGraph.txt
COORDS_FILENAME, string, Coords
REQUESTS_FILENAME, string, Requests.txt
SCHEDULE_FILENAME, string, schedule.txt
LP_REPORT_FILENAME, string, flow_summary.txt
QUEUES_REPORT_FILENAME, string, queues_summary.txt
SNR_TO_PER_FILE_NAME, string, snr2per.txt

// 0 means use no scheduler, any other number uses the
samrt scheduler
//-By run script - USE_SMART_SCHEDULER, int, 0
// 1 - shortest path routing, 2 - max capacity in a
single path routing, 3 - multicommodity flow routing, 4 -
LP with conflicts
//-By run script
ROUTER_TYPE, int, 2


// this mode generates as many slots as needed so the
scheduler generates a
// cyclic schedule whose length may be more than 1 sec
//-By run script-ALLOW_ADDING_SLOTS, int, 0

SCHEDULE_BATCH_SIZE, int, 5
ROUNDING_FACTOR, double, 0.95

N_FREQS_FOR_SCHEDULE, int, 3
N_TIME_SLOTS, int, 200

// output period and resolution
TIME_PERIOD, int, 12
TIME_RESOLUTION, int, 1

// Seed for the random number generator - 0  (or no seed)
means use random seed
SEED_FOR_GENERATOR, int, 7
CLUSTER_RADIUS, double, 0.00

// number of clients
N_CLIENTS, int, 49

// number of requests from each type
N_NEAR_REQUEST, int, 8
N_FAR_REQUEST, int, 0
// definition of when a request is considered far
```

```
FAR_THRESHOLD, double, 0.9

// 1 km square - the program select points in the unit
square,
// and then scales them up
SCALE_IN_METERS, int, 1500


// size in mega bit per second
DEMAND_SIZE, double, 50

// dynamic model (velocities are normalized to unit
square)
// so 0.2 for a square of 30km means 6 km / h
MEAN_VELOCITY, double, 0.0
STD_VELOCITY, double, 0.0
MAX_VELOCITY, double, 0.0

// communication parameters
// in GHZ
FREQUENCY, double, 2.4
// the power is approximately 15 dbm
TRANSMISSION_POWER_IN_MILI_WATTS, double, 100

// Path loss model 1- book, 2-elbit, 3-directly use
alpha,beta
PATHLOSS_MODEL, int, 2
PATHLOSS_ALPHA, double, -1
PATHLOSS_BETA, double, -1
PATHLOSS_ELBIT_C, double, 0
PATHLOSS_ELBIT_H_TR, double, 1090
PATHLOSS_ELBIT_H_RC, double, 1090


// NOISE_IN_DBM, double, -101.7
NOISE_IN_DBM, double, -100
SNR_TO_PER_FILE_NAME, string, snr2per.txt
// In meters (different model before and after)
D_BREAKPOINT, double, 30.
// STD in DB
SF_BEFORE_BREAKPOINT, double, 3.
SF_AFTER_BREAKPOINT, double, 6.
// Arbitrary - to get yes no for and edge instead of
distribution
NUM_STD_FOR_DECAY, double, 0.

// Sensitivity parameters - for communication and
interference (in db)
MIN_SNR_FOR_COMMUNICATION, double, 3.
MIN_SNR_FOR_INTERFERENCE, double, -5.
```

```
// protocol parameters - data size are in kbyte, other
lengths in bits, times are in micro-seconds, rates are in
mbit/sec
HEADER_TIME, double, 4.
PREAMBLE_TIME, double, 16.
MAC_SERVICE_TRAILER_TIME, double, 8
MAC_HEADER_LEN, double, 240.
MAC_FCS_LEN, double, 32.
MAC_RTS_LEN, double, 160.
MAC_CTS_LEN, double, 112.
MAC_ACK_LEN, double, 112.
DIFS_TIME, double, 28.
SIFS_TIME, double, 10.
CW_TIME, double, 135.
// this means 2kbytes = 16 kbits of data before encoding
PAYLOAD, double, 2.
```

8) Tables Format

A communication graph table consists from an edge number, both its nodes *i* and *j*, edge capacity and PER. The graph is undirected and defines which nodes can communicate and at what conditions.

The delimiter is used to separate the fields in all the tables is comma ”,”. In the header (first row) of the table the number of edges and nodes will appear.

Edge number, i ,j , Capacity ,PER, MCS

An interference graph (internal for an algorithm part) table consists from an edge number and both its nodes *i* and *j*. The graph is undirected and defines which nodes interfere to each other. In the header of the table the number of edges will appear.

Edge number , i , j

A coordinates table consists from a node number, its (x,y) and its velocity (speed and angle). In the header of the table the number of nodes will appear.

The coordinate table would appear for each second in simulation if there is an update.

Node number , X , Y , V , A

A location table consists from a node number and its (x,y) location in the plane.

Node number , X , Y

A requests table consists from stream number, source node, required bandwidth, destinations number and list of destinations. Required bandwidth is a requirement for an actual rate in Mbps. In the header of the table the number of requests will appear.

Stream number , Source node , Required rate , Number of destinations , Destination Node

A scheduler table consists from the time slot number, frequency for communication, transmitter node, stream number, receivers number, receivers list, transmission rate and MCS index.

A transmission rate is a flow obtained by the algorithm on the edges between the transmitter and the receivers . We assume that the number of frequencies is 3, but the algorithm is capable to get the number of frequencies as a parameter. In the header of this table will appear the number of time slots in 1 second and a payload size. We assume that the same payload size is chosen for all transmissions in the table the header corresponds.

Slot , Frequency , Transmitter , Stream Number , number of Receivers , receivers list , Transmission rate

9) Additional parameters

The following parameters appear in the AdHocWiFi directory omnetpp.ini file and represent needed parameters for simulation (the format is the OMNET++ configuration file format):

The 3 frequencies used:

```
sim.channel1.carrierFrequency = 2.412e+9Hz  # [Hz]
sim.channel1_node[*].nic.connectionManagerName = "channel1"
sim.channel1_node[*].nic.*.centerFrequency = 2.412e9Hz

sim.channel2.carrierFrequency = 2.417e+9Hz  # [Hz]
sim.channel2_node[*].nic.connectionManagerName = "channel2"
sim.channel2_node[*].nic.*.centerFrequency = 2.417e9Hz

sim.channel3.carrierFrequency = 2.422e+9Hz  # [Hz]
sim.channel3_node[*].nic.connectionManagerName = "channel3"
sim.channel3_node[*].nic.*.centerFrequency = 2.422e9Hz
```

The video used, the video name prefix , the video file name is: <name>_<number>_<frame-per-second>.rtp where <number> varies from 1 to "rtpNumber" parameter and <frame-per-second> is one of 10,15,20,25,30.

For example a file name can be "vids/wildlife_16_30.rtp".

```
sim.channel*node[*].appl.rtpName = "vids/wildlife"
sim.channel*node[*].appl.rtpNumber = 16
```

This parameter controls whether to write statistics to screen about the RTP stream such as packet sizes when running simulation.

```
sim.channel*node[*].appl.rtpStatistics = false
```

This parameter controls whether to run in video or simple (raw data) mode, can be either "video" or "raw"

```
sim.channel*node[*].appl.mode = "video"
```

Controls part or full statistics mode, full mode is slower but provides additional analysis details in the .csv file

```
sim.channel*node[*].appl.statisticsMode = "full"
```

The following parameters control the different configurations used when running simulation for benchmarking

- one_slot means the scheduling table is for a single slot only with conflicts.
- enableArbitration means when having a conflict of multiple tasks the application would send by weighted arbitration (where the weight is relative to the data size) from the current messages in queues
- adjustRoundTime means whether to treat the scheduler maximal time base as a single round since the scheduler would use more slots in the table then allowed

```
[Config Main]
sim.channel*node[*].appl.one_slot = false
sim.channel*node[*].appl.enableArbitration = false
sim.channel*node[*].appl.adjustRoundTime = false

[Config MainNoSched]
sim.channel*node[*].appl.one_slot = true
sim.channel*node[*].appl.enableArbitration = true
sim.channel*node[*].appl.adjustRoundTime = false

[Config ShortestPath]
sim.channel*node[*].appl.one_slot = true
sim.channel*node[*].appl.enableArbitration = true
sim.channel*node[*].appl.adjustRoundTime = false

[Config ShortestPathSched]
sim.channel*node[*].appl.one_slot = false
sim.channel*node[*].appl.enableArbitration = false
sim.channel*node[*].appl.adjustRoundTime = true

[Config MultiCommFlowSched]
sim.channel*node[*].appl.one_slot = false
sim.channel*node[*].appl.enableArbitration = false
sim.channel*node[*].appl.adjustRoundTime = true

[Config MultiCommFlowNoSched]
sim.channel*node[*].appl.one_slot = true
sim.channel*node[*].appl.enableArbitration = true
sim.channel*node[*].appl.adjustRoundTime = false
```