Simulation of the aloha protocol

Abstract-The present essay is a tutorial on the OMNeT++ simulation environment, through the analysis of the known ALOHA protocol. The model implements the ALOHA random access protocol on the Link layer, and simulates a host to server instant broadcast. ALOHA is rather simple yet convenient to demonstrate the potential of OMNeT++ in simulating wireless protocols. The final part evaluates the pure and the slotted ALOHA variations, in regard to the theoretical models.

Introduction

OMNeT++ is a discrete event simulation (DES) environment, developed by Andras Varga as public source, and is accompanied by Academic Public License, which means that it is free for nonprofit academic use. The intention behind OMNeT++ was the development of an open source generic simulation environment, not exclusively dedicated to network simulations as the more known ns-2, or the commercial Opnet. The environment offers instead, a generic and flexible platform to develop simulation frameworks dedicated to complex IT systems, as wireless and sensor networks, the classic IP and IPv6 stacks, queuing networks, optical networks and various hardware architectures.

Typical example of a framework that provides simulation components for IP, TCP, UDP, Ethernet and MPLS, is the INET Framework and the MiXiM, which is an aggregation of several frameworks for mobile and wireless simulations. The OMNeT++ ver. 4. 0 is built on the known Eclipse CDT ver. 5. 0, and uses most of its resources. It is offered for Windows and Linux operating systems. The core of the models is coded in C++, in Linux uses the gcc compiler and in Windows uses the MinGW port for the gcc suite. There is a commercial version called OMNEST, with no significant accessories than the open version, except the optional use of the native Visual C++ compiler for the Windows platform.

The ALOHA protocol was one of the oldest random access protocols, invented by Norm Abramson in 1969. The first wireless network, implementing packet switching over radio, used the pure ALOHA variation, have initially established in Hawaii. Later Abramson interfaced the ALOHAnet with the ARPAnet, the primitive form of internet. The ALOHA have inspired the creation of CSMA/CD and the birth of Ethernet. Finally, the random access protocol has evolved to contemporary CSMA/CA, the MAC layer of Wi-Fi. The slotted ALOHA and the later pure ALOHA have simple implementations, appropriate for simulation. It uses only the host to server broadcast instant, but is adequate to calculate the maximum channel capacity and demonstrate some other interesting attributes, as well.

OMNeT++ DESCRIPTION

1. The Structure of Models
2. OMNeT++ is based on C++ programming and follows the object-oriented approach with classes and class instances, the so-called objects. The simulation model consists of modules, which communicate by message passing. The core element is the simple module, which is written in C++, and constitutes an instance of a module type from the simulation class library. The next structural element in the hierarchy is the compound model, which is formed from simple modules or other compound models. Every module, simple or compound, has ports to communicate with the external environment, called gates. Gates could

be bidirectional or could be restricted to input or output. Modules are connected through their gates via connections and they communicate by exchanging messages, via these connections.

The block diagram in Fig. 1 depicts the internal module structure according to the declared hierarchy, in OMNeT++. The connections are limited within the module range but the message exchange can be established across hierarchy levels. This is applicable in the case of modeling wireless systems and the simulation of ALOHA stations will make use of it. Otherwise, messages are traveling through the chain of connections. Modules have parameters, which are used to pass initialization data during the initiation of the simulation. The compound models can pass parameters to the contained sub-modules. The final model which contains the aggregate of the modules is called network model, is represented as a class and each simulation run is executed on an instance of this class.

3. The NED language
4. The structure of the simulation in OMNeT++ is based on the network description language (NED). The NED includes declarations for the simple modules and definitions for the compound modules and the network model. The language programming is accomplished by the integrated graphic editor, as shown in Fig. 2 or the text editor, as shown in Fig. 3. Both editors are producing equivalent code, and the programmer can switch from one another without any derogation.
5. The programming model

6. The typical code development includes the following steps: The programmer creates the network model, by creating the appropriate network_name. ned file, using the IDE. The NED file describes the network name and the topology, which are the names of the sub-modules, simples and compounds. Every sub-module should have its own module_name. ned file, which includes the parameter declarations and other meta-data. As mentioned, the behavior of every simple module is expressed in C++, so there should be two specific files, the module_name. cc and the module_name. h, for every simple module. These files are compiled during simulation execution by the supporting C++ compiler, and linked with the simulation kernel and libraries.

   The programmer usually tests the behavior of the simulation model according to different inputs. These could be entered manually by the user, during simulation execution, or could be included in a configuration file. Generally, there is a special type of file the omnetpp. ini that contains these parameters and the rest of the building blocks, to support user interaction. The IDE includes an editor for the initialization files, which can switch between form view, as shown in Fig. 4, and source view, as shown in Fig. 5. The two fields are equivalent.

   There are two fundamental methods to develop C++ code for programming the simple module behavior: The co-routine based and the event processing function. In the first approach, every module executes its own threat of code, which is activated every time it receives a message from the simulation kernel. In the last approach, the simulation kernel calls the module function, having the specific message as argument. Prior to main() function execution, an initialization function declares variables and objects and before program termination, a finalization function saves the data logged during simulation, and produces histograms.

7. OMNeT++ Architecture and Potentiality
8. The following Fig. 6 presents the internal logic structure of OMNeT++. The first block is the model component library, which the programmer develops in C++, and contains the compiled code of simple and compound modules. The simulation kernel and the class library (SIM) instantiates the modules and build the concrete simulation model. The user interface libraries (Envir and Cmdenv or Tkenv) provide the simulation environment, which defines the source of input data, the sink of simulation results and the debugging information. It controls the simulation execution, visualization and animation.

   Cmdenv provides only command line and text mode input-output, and it is more appropriate for batch simulations. Tkenv is the graphical user interface (GUI) of OMNeT++. It provides automatic animation, module output windows and object inspectors. The following Fig. 7 depicts an active simulation output through OMNeT++/Tkenv.

   OMNeT++ includes very powerful tools to visualize the interaction among modules. A sequence chart diagram provides a way to inspect the timing of the events during simulation by extracting data from an event log file. During the finalization routine, the logged data are saved to specific result files, the vectors in network_name. vec and the scalars in network_name. sca files, respectively. For the result analysis, OMNeT++ produces the analysis file network_name. anf, which contains aggregated data in vectors and scalars plus any histograms, created during the final stage. All the types of data can be further processed by using pattern rules, in “ datasets and charts” section, to produced advanced charts and graphs. In the ALOHA simulation most of the available choices are used for demonstration.

Aloha Simulation

1. Background Theory
2. The slotted ALOHA is the most simple random access protocol. The transmitting station always broadcasts at the full rate R of the channel. The transmission initiates at the beginning of the slot, which is common for the aggregate of the stations. If two or more stations transmit simultaneously, then the condition is called collision and all the stations involved, after a random time different for each, retransmit the frame until successful delivery. The procedure is presented at the following figure:

   The slotted ALOHA protocol allows each station to transmitat at the channel’s full speed R, but requires slots to be synchronized in all the stations, something not nessesary for the unslotted or pure ALOHA. The following assumtions are made to simplify simulation:

   - The source generates single frames of length L bits. The inter-arrival times between frames follow exponential distribution.

- If R bps is the capacity of the wireless link then the slot time is set equal to the transmission time of each frame, which is: tframe= LR sec.
- All nodes are synchronized and transmit frames only at the beginning of a slot.
- If a node has a new frame to send, it waits until the beginning of the next slot.
- If two or more frames collide, then their hosts retransmit after random time, following exponential distribution.
- If there is no collision, then the node transmits its next frame following exponential distribution.

I define N the number of stations operating the slotted ALOHA protocol and p the probability of each station to transmit in the next slot. The probability for the same station to do not transmit in the next slot is then 1-p, and for the rest of the stations is 1-pN-1. Therefore, the probability for a station to have a successful transmission during the next slot is to transmit and the rest of the stations to do not transmit, so it is p1-pN-1, and because there are N stations, the probability that an arbitrary node has a successful transmission is Np1-pN-1.

A slot where a single station transmits is called a successful slot. The efficiency of slotted ALOHA is defined as the long run fraction of successful slots, which is:

Ep= Np1-pN-1 (1)

To find the maximum efficiency, we seek p* that maximizes (1). Then:

E’p= N1-pN-1-NpN-11-pN-2= N1-pN-21-p-pN-1

If E’p= 0 then p*= 1N . Using this value, the maximum efficiency is:

Ep*= N1N1-1NN-1= 1-1NN-1= 1-1NN1-1N (2)

For a large number of active stations, the maximum efficiency accrues from (2) as N approaches infinity:

limN?? Ep*= limN?? 1-1NNlimN?? 1-1N= 1e1= 1e= 0. 368 (3)

From (3), the maximum efficiency of slotted ALOHA is 0. 368 or 36. 8%

The unslotted version or pure ALOHA protocol does not have the restriction of slot synchronizing, and the station is able to broadcast when a new frame is available. So pure ALOHA is a full-decentralized random access protocol. When a transmitting station detects a collision, after completing the transmission, it retransmits the frame with probability p. If it chooses to postpone the transmission for a single frame transmission period tframe= LR sec, then the probability is (1-p). The figure below depicts transmissions and collisions in the unslotted channel.

The maximum efficiency of pure ALOHA protocol is calculated similarly as the slotted ALOHA. The only difference here is that the rest of the stations should have not begun transmitting before and should not begin during the broadcast of the given station. The probability that the rest of the stations remain idle is 1-pN-1 and the probability that they remain idle is 1-pN-1 again. Therefore, the probability that the given station will have a successful transmission is p1-p2N-1.

Again, we seek the value of p* that maximizes (4), which is the probability of successful transmission for the sum of the N stations.

Ep= Np1-p2N-1 (4)

E’p= N1-p2N-2-Np2N-11-p2N-3= N1-p2N-31-p-p2N-1

If E’p= 0 then p*= 12N-1 . Using this value the maximum efficiency is:

Ep*= N2N-11-12N-12N-1 (5)

From (5), the maximum efficiency accrues as N approaches infinity, which is:

limN?? Ep*= 121e= 12e (6)

From (6) I assume that the maximum efficiency, for the pure ALOHA protocol, is 0. 184 or 18. 39%, the half of slotted ALOHA.

Another useful diagram is in Fig. 10. It depicts the apparent superiority of slotted ALOHA over the pure ALOHA protocol, despite the limitations that turn it to non-functional. The normalized total traffic is the aggregate traffic, which generated by the source of the station, divided by the channel capacity R and the normalized throughput ? is the average successful traffic (non-collided) divided by R. The slotted ALOHA achieves double throughput than the pure ALOHA and achieves its maximum efficiency when the generated traffic rate equals the channel’s capacity R. The pure ALOHA although, achieves its maximum efficiency when the generated traffic equals to R/2.

3. Model Development
    - NED language

- The following paragraphs describe the process of creating a functional model for the simulation of ALOHA protocol in OMNeT++. The object of simulation is to study the behavior of the ALOHA model and to confirm the theoretical values of maximum efficiency for pure and slotted ALOHA. The ALOHA random access protocol is peer based and does not use a server-client architecture. It is convenient to study the effect of collisions and random retransmissions only in the case when one host is receiving (becomes server) and the rest of the hosts are transmitting.

  The first step is to develop the NED code that describes the network Aloha. The following Aloha. ned file creates the Aloha network, which consist of simple modules, one called server and a number of hosts, equal to numHosts parameter. The txRate defines the transmission rate R, of the wireless channel, and slotTime defines the type of protocol. Zero means pure ALOHA and 100ms defines the slot time length. The parameter @display selects a background image, taken from the library.

```
network Aloha
{
parameters:
int numHosts; // number of hosts
double txRate @unit(bps); // transmission rate
double slotTime @unit(ms);// zero means no slots (pure Aloha)
@display(“ bgi= background/terrain”);

submodules:

server: Server;

host[numHosts]: Host {

txRate = txRate;

slotTime = slotTime;

}

}
```

The following Server. ned file describes the server’s simple module. It loads an image for the server icon and defines a gate of input type (in), with which it is not necessary to establish a connection. It can receive a message directly from a host via @directIn, something that is usual to wireless simulations.

```
simple Server

{

parameters:

@display(“ i= device/antennatower_l”);

gates:

input in @directIn;

}
```

The following Host. ned describes the host’s simple module. It loads a set of parameters from the omnetpp. ini file, the radioDelay, which is the propagation delay over the radio link, pkLenBits, which is the length of the frame, and iaTime, which is the random inter-arrival time, following exponential distribution. The rest of the parameters, txRate and slotTime, are loaded in Aloha. ned, during sub-module instantiation.

```
simple Host

{

parameters:

double txRate @unit(bps); // transmission rate

double radioDelay @unit(s);// propagation delay of radio link

volatile int pkLenBits @unit(b); // packet length in bits

volatile double iaTime @unit(s); // packet interarrival time

double slotTime @unit(s); // zero means no slots (pure Aloha)

@display(“ i= device/pc_s”);

}
```

- Configuration
- The most critical file is the configuration file omnetpp. ini. It stores the values of the parameters that are loaded in the NED parameter fields. When declaring on the [General] field that Aloha. slotTime= 0, is presets globally the pure ALOHA protocol. Similarly, the Aloha. numHosts= 20 defines the number of hosts

to be 20, the Aloha. txRate= 9. 6kbps defines the R to be 9600bps. The last definitions load the parameters of Aloha model and consequently the parameters of the simple modules that Aloha model controls, which are the server and the host modules. The definitions Aloha. host[*]. pkLenBits= 952b and Aloha. host[*]. radioDelay= 10ms load directly the parameters pk. LenBits and radioDelay on every host submodule, respectively.

```
[General]

network = Aloha

#debug-on-errors = true

#record-eventlog = true

Aloha. numHosts = 20

Aloha. slotTime = 0 # no slots

Aloha. txRate = 9. 6Kbps

Aloha. host[*]. pkLenBits = 952b #= 119 bytes, so that (with +1 byte guard) slotTime is a nice round number

Aloha. host[*]. radioDelay = 10ms

[Config PureAloha1]

description = “ pure Aloha, overloaded”

# too frequent transmissions result in high collision rate and low channel utilization

Aloha. host[*]. iaTime = exponential(2s)
```

```
[Config PureAloha2]

description = “ pure Aloha, optimal load”

# near optimal load, channel utilization is near theoretical maximum 1/2e

Aloha. host[*]. iaTime = exponential(6s)

[Config PureAloha3]

description = “ pure Aloha, low traffic”

# very low traffic results in channel being idle most of the time

Aloha. host[*]. iaTime = exponential(30s)

[Config PureAlohaExperiment]

description = “ Experimental mutliparameter demostration”

repeat = 2

sim-time-limit = 90min

**. vector-recording = false

Aloha. numHosts = ${numHosts= 10, 15, 20}

Aloha. host[*]. iaTime = exponential(${mean= 1, 2, 3, 4, 5.. 9 step 2}s)

[Config SlottedAloha1]

description = “ slotted Aloha, overloaded”

# slotTime = pkLen/txRate = 960/9600 = 0. 1s

Aloha. slotTime = 100ms
```

```
# too frequent transmissions result in high collision rate and low channel utilization
Aloha. host[*]. iaTime = exponential(0. 5s)
[Config SlottedAloha2]
description = “ slotted Aloha, optimal load”
# slotTime = pkLen/txRate = 960/9600 = 0. 1s
Aloha. slotTime = 100ms
# near optimal load, channel utilization is near theoretical maximum 1/e
Aloha. host[*]. iaTime = exponential(2s)
[Config SlottedAloha3]
description = “ slotted Aloha, low traffic”
# slotTime = pkLen/txRate = 960/9600 = 0. 1s
Aloha. slotTime = 100ms
# very low traffic results in channel being idle most of the time
Aloha. host[*]. iaTime = exponential(20s)
```

A selection of the SlottedAloha2 configuration overrides the value of slotTime with Aloha. host[*]. slotTime= 100ms, which fixes the slotted ALOHA protocol with slot time to 100ms. The Aloha. host[*]. iaTime= exponential(2s) sets the frame inter-arrival time on every host to follow exponential distribution, with mean time equals to 2 seconds.

The Config option PureAlohaExperiment exploits the OMNeT’s capabilities of organizing different experiments with simple repetition declarations. The statement Aloha. numHosts=${numHosts= 10, 15, 20} declares three (3) repetitions having 10, 1 and 20 hosts respectively.

The statement Aloha. host[*]. iaTime= exponential(${mean= 1, 2, 3, 4, 5.. 9 step 2}s) declares seven (7) repetitions, with interarrival times equal to exponential distribution and means, 1, 2 , 3, 4, 5, 7 and 9, respectively. The repeat= 2 statement doubles the number of runs, so finally the available choices will be 2x3x7= 42 from 0 to 41 optional runs. The statement sim-time-limit= 90min constrains the simulation time to 90 minutes.

- C++ model coding
- The simple modules Host and Server are based on C++ programming. The relevant host. cc, host. h, server. cc and server. h, which are included entirely in the appendix section, implement the model behavior during simulation by exchanging messages directly one-another or with the simulation kernel. The following Fig. 11 is a design- level class diagram, describing the basic relationships among network module Aloha and simple modules, Host and Server. The two last, inherit from cSimpleModule simulation class library, and redefine the basic methods initialize(), handleMessage(), activity() and finish(), according to the desired function.

The Aloha network model comprises of several Host objects and one Server, so it keeps an aggregation association with Host and Server classes. It passes also to them some parameter values, some declared in the omnetpp. ini file and some taken from user dialog form. The Host module keeps an one-way association with Server because every Host declares a Server object in the attribute field, in order to send a direct message (pk) later, by calling the sendDirect() function. The scheduleAt() function programs the kernel to send the Host an “ endTxEvent” message when the transmission ends. This is represented by the self-association. Similarly, the Server module programs the kernel to send the Server an “ endRxEvent”, when the reception of the message sent from Host finishes, and is represented as the self-association.

The module code is cited commented in the appendix. Here, I will explain the finish() function of the server module, because it creates the result reports, necessary for the exploitation of the simulation.

```
void Server:: finish()

{

EV << " duration: " << simTime() << endl;

EV << " total frames: " << totalFrames << endl;

EV << " collided frames: " << collidedFrames << endl;

EV << " total receive time: " << totalReceiveTime << endl;
EV << " total collision time: " << totalCollisionTime << endl;

EV << " channel utilization: " << currentChannelUtilization << endl;

recordScalar(“ duration”, simTime());

recordScalar(“ total frames”, totalFrames);

recordScalar(“ collided frames”, collidedFrames);

recordScalar(“ total receive time”, totalReceiveTime);

recordScalar(“ total collision time”, totalCollisionTime);

recordScalar(“ channel utilization”, currentChannelUtilization);

recordStatistic(&collisionMultiplicityHistogram, “ packets”);

recordStatistic(&collisionLengthHistogram, “ s”);

}
```

The finish() function is called before the termination of the simulation. The first six commands enable the printout of the class variables on the Tkenv window. It prints the final simulation time, the total transmitted frames, the collided frames on the server, the total reception time of un-collided frames, the total time spent on collisions, and the last value of channel utilization, which comes from the following formula:

Final Channel Utilization= Final Total Receive TimeFinal Simulation Time

It is expected that the Final Channel Utilization, after an adequate simulation time, that will reach the maximum theoretical values if the incoming traffic is adjusted at an optimal value. The recordScalar() function records these values to the relevant scalar file config_name. sca. Another note is that during initialize() function there is code to create vector logging, which is the following functions:

- collisionMultiplicityVector. setName(“ collision multiplicity”);
- collisionMultiplicityVector. setType(cOutVector:: TYPE_INT);
- collisionMultiplicityVector. setInterpolationMode(cOutVector:: NONE);
- collisionLengthVector. setName(“ collision length”);
- collisionLengthVector. setUnit(“ s”);
- collisionLengthVector. setInterpolationMode(cOutVector:: NONE);
- channelUtilizationVector. setName(“ channel utilization”);
- channelUtilizationVector. setType(cOutVector:: TYPE_DOUBLE);
- channelUtilizationVector. setInterpolationMode(cOutVector:: LINEAR);

It creates three vectors, the current number of collisions, the current time wasted by the collision and the current channel utilization, with the simulation timestamp. The following function creates two histogram functions, the number of collisions and the time wasted by collisions. The histogram is auto-ranging, having declared only the lower limit to be 0. 0 by the setRangeAutoUpper() function:

- collisionMultiplicityHistogram. setName(“ collision multiplicity”);
- collisionMultiplicityHistogram. setRangeAutoUpper(0. 0);
- collisionLengthHistogram. setName(“ collision length”);
- collisionLengthHistogram. setRangeAutoUpper(0. 0);
- The data-logging code lies in the handleMessage() function, where:
- collisionMultiplicityVector. record(currentCollisionNumFrames);
- collisionMultiplicityHistogram. collect(currentCollisionNumFrames);
- collisionLengthVector. record(dt);
- collisionLengthHistogram. collect(dt);
- channelUtilizationVector. record(currentChannelUtilization);

The record() function appends a time-stamped value on the relevant vector and the collect() function adds another value on the histogram graph. The recordStatistic() function finalizes the two histogram graphs before termination.

recordStatistic(&collisionMultiplicityHistogram, “ packets”);

recordStatistic(&collisionLengthHistogram, “ s”);

4. Simulation Results

- Sequence Charts
- To enable the event logging feature, the command record-eventlog= true should be placed on the omnetpp. ini file. Then, on the end of the simulation, a config_name. elog file is created. To have a clear demonstration of the ALOHA protocol, I choose two options: A slotted ALOHA simulation, with exponential interarrival times having 2 sec mean and pure ALOHA simulation, with exponential interarrival times having 6 sec mean. A typical slotted ALOHA sequence is the following:

  On the #2096 event, the host[4] receives a self scheduled message " endTxEvent" from the kernel, on the beginning of a new slot.

  It generates the packet " pk-7-#26", begin to transmit it to the server with the sendDirect() command and set the host's state= TRANSMIT.

  It schedules the end of transmission message " endTxEvent" with a scheduleAt() command, after time:

  ttrans= LR= 100 msec, where L is the frame length (960b) and R is the channel capacity (9600bps).

  On the #2097 event, after the simulated propagation time tprop= 10 msec (on the radio channel), the server receives the packet " pk-7-#26" from host[4], set the channelBusy= true and schedules the end of reception with a scheduleAt() command, after 100 simulated msec.

On the #2098 event, the host[4] receives the self scheduled " endTxEvent" message from the kernel, set the host's state= IDLE and schedule the next frame transmission after random time.

On the #2101 event, the server receives the self-scheduled message " endRxEvent", set the channelBusy= false and calculate the rest of the class variables.

The events #2100 and #2099 happen almost concurrently, therefore they collide on the server, starting with event #2102, then #2103 and ending the collision with event #2106. The packets transmitted are supposed to be discarded by the server. The overlapping blue parallelograms indicate the collision zone. The Fig. 12 depicts the above sequence:

The pure ALOHA protocol is free of slot restrictions and the host is able to transmit whenever a packet is available. Similarly as the previous procedure on the event #52, host[0] begins transmitting and on the event #53 the server starts receiving. However, on event #54 the host[6] starts transmitting and the server detects a collision on the event #55. The collision ends on the event #58 and both packets are discarded. It is obvious the server waste longer time interval on the former collision, due to the non-synchronized condition of the hosts. Pure ALOHA looks less efficient than slotted ALOHA. Afterwards, host[6] retransmits on event #59 and the server responds on event #60. The broadcast remain un-collided and ends normally on event #62.

- The pure ALOHA protocol

- This section is dealing with the evaluation of the simulation model, its behavior comparing the theoretical predictions and some conclusions that results from the extracted graphs. The most crucial parameter to ALOHA simulation is the traffic seed. The incoming traffic is considered a Poisson process. The specifications of a Poisson process declare Poisson distribution for the number of incoming packets and exponential distribution for the times between the packets (interarrival times). It is assumed that the station’s source generates one packet (frame) per message of constant length (960b). So the model needs to simulate only the interarrival times, using the function exponential(mean). The mean is a double precision number representing the mean of the distribution, in seconds. The function returns a double precision float, representing the random time, in seconds.

  For the pure ALOHA model: The user selects initially, via a dialog field, a low traffic profile (mean= 30 sec), then a heavy traffic profile (mean= 2 sec) and finally the optimal traffic profile (mean= 6 sec). It is expected that the optimal traffic profile induces a channel utilization ratio near the theoretical maximum of 0. 184. After the execution of the pure ALOHA simulation, using the tree traffic profiles, the relevant vector and scalar files (PureAloha*-0. vec and PureAloha*-0. sca) appear in the aloha/results directory. The post processing of these files produces the PureAloha. anf analysis file, which is included. The simulation time is 90 min for all models and the number of host is 20.

The first group of histograms in Fig. 13 depicts the PDF of the collision multiplicity, for the three traffic distributions. It is obvious that the heavy traffic profile induces much more collisions and therefore the probabilities to have more than two or three simultaneous collisions are higher. The number of multiplicity extends up to 20 comparing to 9 in optimal traffic and 3 in low traffic. The scalar files give some accessional data on every distribution, like minimum, maximum and mean values, standard deviation, etc.

The second group of histograms in Fig. 13, d