ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecture with Computer Exercises:
Modelling and Simulating Social Systems with MATLAB

Project Report

Analysis of Packet Transportation Systems
With a Focus on Buffer Systems and Blockades

Lukas Cavigelli & Pascal Hager & Christian Schürch

Zurich
May 2011

Agreement for free-download

We hereby agree to make our source code for this project freely available for download from the web pages of the SOMS chair. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

Lukas Cavigelli Pascal Hager Christian Schürch

Please wait...

If this message is not eventually replaced by the proper contents of the document, your PDF viewer may not be able to display this type of document.

You can upgrade to the latest version of Adobe Reader for Windows®, Mac, or Linux® by visiting http://www.adobe.com/products/acrobat/readstep2.html.

For more assistance with Adobe Reader visit http://www.adobe.com/support/products/acrreader.html.

Contents

1 Individual contributions

As a team of three we had to split some of the work. All together we worked out the research questions and organized the work-share.

Pascal Hager and Lukas Cavigelli were concerned about the programming in Matlab. Together they implemented the simulation. Pascal then worked on the linear transportation system and the buffers. Lukas implemented the linear distribution system. Christian Schürch took care of the written part and the typesetting in LaTeX.

2 Introduction & Research Questions

In today's society complex delivery systems are omnipresent. They are used by the postal services, airports or the military to name a few. Also in the industry logistics is of great importance. For example the automotive industry is very much depending on a good working logistic system [3].

A big subtopic of logistics is buffering. To be able to store units for a certain amount of time is always an issue when it comes to continuous processes. For example in an industrial production facility it is expensive to not operate production machines at full capacity. On the other hand the raw material might not come continuously and this is exactly when a buffer comes to play. Another buffer is necessary to compensate for the differences between production capacity and customer demand for the products [5].

Also in information technology buffers are a big issue: Computation speed is increasing exponentially (Moores Law [1]) but memory read and write access times however are improving much slower. This enormous speed difference requires extremely sophisticated buffering and caching technologies. Furthermore in communication electronics buffering is omnipresent: incoming data of the communication channels have to be buffered until the device is ready to process the data [4].

The above stated facts motivate for the need of buffers using physical and economical reasons. The question of the size of the buffers is a totally different one but is still of great importance. Taking security considerations into account one can say that the bigger a buffer is the more secure a logistic system runs [5].

Another point of view is the flexibility of a transportation system. Wisely used buffers can make a system more flexible, what enables stable performance under changing conditions [2].

Before we can analyze buffer systems we have to understand how simple transportation systems work. We have to find the operating conditions where a buffer makes sense at all. Therefore we worked out the following four research questions:

1. Which input-to-output ratio is the limit to have a valid[1] transportation system?
2. What is the maximum possible throughput of a valid system?
3. At which path occupation of a linear transportation system do we have maximum throughput?
4. Is it possible to optimize a given geometry for both throughput and travel time?

For buffers we mainly want to find out how it has to be dimensioned (see question 6). But first we had to find meaningful boundary conditions (question 5). For buffers we worked out the following four sub questions:

5. How should a buffer be stressed such that it is meaningful, but not overly stressed?
6. What minimum capacity must a buffer have for a given source rate signal?
7. Is it possible to make a conclusion about packet drops by looking at the buffer's occupation?
8. Is it possible to improve the buffer properties (smaller capacity or mean travel time) by a more complex buffer design than the linear design?

3 Description of the Model & Implementation

For this work we decided to analyze a packet transportation system. A real world example might be found in an airport where luggage has to be transported from one point to the different gates depending on the destination of the owner of the piece of luggage.

This section explains the simplifications made for our model and how we define a transportation system as valid. Further we explain how we implemented our model and which elements exist on a map. At last we name the different geometries we used. A short overview of the statistic parameters used to quantify our results is given in the last subsection.

3.1 Simplifications of the Model

We strongly simplified the problem for our model. First we discretized in space and time. A real world luggage transportation system consists of conveyor belts that move continuously. We on the other hand split the transportation ways into discrete transportation elements that are able to move one packet per time step.

[1] See section 3.2 for the definition of a valid transportation system.

Secondly we assumed that all packets are of the same physical dimensions. With this assumption we can define that one packet fits one discrete transportation element. Like this we don't have to take complex collision problems into account.

3.2 The Definition of a Valid Transportation System

Our model is based on a map of a characteristic transportation system. It contains sources, transportation elements, junctions, sinks and of course packets which need to be delivered.

In a well designed transportation system the sources must be able to deal with a defined income of packets. The sinks define the system outputs. The following definition helps to categorize the results of section 4:

Definition 1 *A transportation system is **valid** as long as all packets that - by the definition of the source have to enter the system - can be processed by the system. Otherwise the transportation system is termed **invalid**.*

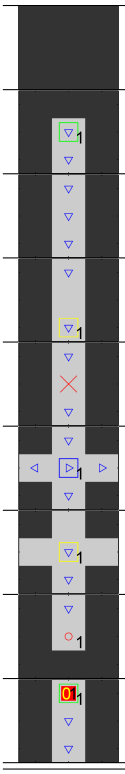
In the further text we often use the expression 'packet drop' if a packet can't enter the system because it is blocked right at the source. This of course is a reason for a system to be invalid.

If a system is valid it can be linked in a serial way with other valid systems. With that concept, we can create arbitrary complex transportation systems.

3.3 Elements of a Map

The following section describes all elements of our maps. In table 1 all elements are listed with their graphical appearance and their name.

Tab. 1: All elements of the maps used in this work.

	void field
	source field
	directed transportation element (DTE)
	waiting field
	error field
	diverging junctions (JND)
	converging junctions (JNC)
	sink field
	packet

First of all there are so called *void fields* in dark grey. No packet may ever reach this region.

The *source fields* are marked with a green frame. They generate packets after certain rules and send them to the neighboring transportation element. These transportation elements are light grey and show a blue arrow that defines the direction of the element.

All packets are moving on *directed transportation elements* (short DTE) shown in light grey. In every time step the packet is shifted in the direction marked by the blue arrows. If the neighbor field is occupied the packet waits on its own field until the neighbor field is free.

A packet reaching a *waiting field* stays on this field until it is further processed by the following converging junction.

An *error field* indicates an failure in the transportation path. If a packet reaches such a field it stays there until the error is cleared.

There are two different types of junctions: *Diverging junctions* (short JND) are marked with a blue square, have one input direction and can have up to three output directions. A JND can steer the packet flow after an implemented rule by altering its directed transportation path (the blue arrow within the blue square). *Converging junctions* (short JNC) are marked with a yellow square. They can have up to three input directions

and one output direction. All input fields must be waiting fields. Also JNC's have rules implemented which decide wether to process a packet and which packet to choose.

Sink fields are marked with a red circle and simply remove and register all packets that are delivered to them according to a given rule. The number shown in black indicates the address of the specific sink.

Packets themselves are shown completely in red and feature an ID (black) and a destination address (yellow) telling which sink it should be delivered to.

3.4 Geometry

In this work we take a closer look at three different characteristic transportation systems or elements respectively:

The linear transportation system (=LTS) is the simplest system geometry that we use to find out the main characteristics of the transportation problem and can be seen in fig. 3. It contains a source, a sink and a few DTE's in between.

A buffer is a set of DTE's arranged in such a way that it is capable of storing a certain amount of packets (see fig. 15) and giving them free when ever asked to. A buffer connects a feeder line to a capacity limited consumer line. The aim of a buffer is to store packets as long as the consumer line is blocked. In an ideal case a buffer is sized such that it never fills up completely.

Linear distribution systems (=LDS) have only one connecting path between the sources and the sinks. Figure 19 shows this geometry. Every packet has only one chance to reach its target sink.

3.5 Implementation

3.5.1 General Structure

The implementation of our model is based on a cellular automaton with a 1^{st} order Von Neumann neighborhood. This means each cell only interacts with its direct neighbors (see fig. 1). The cellular automata is modified in the following way: The sources and sinks only account for themselves. The DTPs only account for themselves and the field in front of them. The junctions on the other hand account in general for the whole 1^{st} order Von Neumann neighborhood whereby some fields depending on the rules can be ignored.

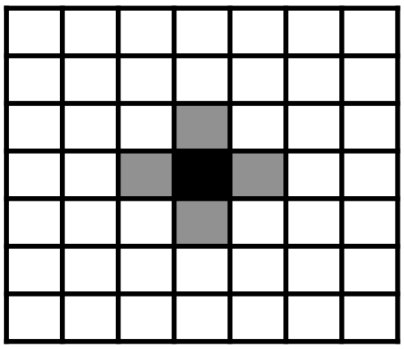


Fig. 1: Visualization of a 1st order Von Neumann neighborhood.

How the map is updated is shown in fig. 2. In every time step, the following iteration routine is performed: First all sinks containing packets are checked if the packet should be removed. Then the outflow directions of the JND's are updated. Now all packet are moved within the DTP. After that the JNC's take in the packets located on a waiting field if they have permission. In the end the source fields generate new packets.

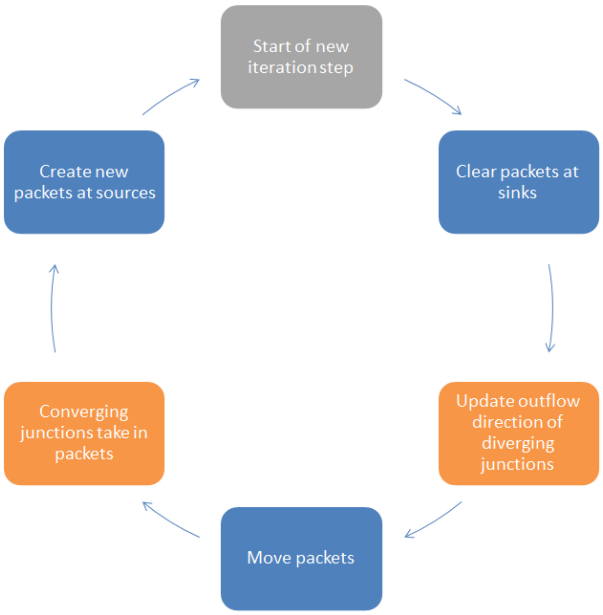


Fig. 2: Visulazion of the main iteration loop as it is implemented.

3.5.2 Description of the Most Important Files

All the MATLAB scripts are stored in two folders. '/bin' contains all scripts related to the simulation of the system and '/maps' contains all scripts related to the generation of all maps used. This section shortly

describes the most important files. First we take a look at the '/bin' folder:

initialize_simulation.m sets all constants and superior variables.

run_simulation.m runs the simulation.

XXXRULE.m contains all rules for the junctions (JND and JNC), sources and sinks. Sources need two rules, one for the generation of packets and one for the distribution of the target numbers.

cur_fig_style_and_export.m is a function to automatize the labeling and exporting of the plots.

In the '/maps' folder we find the following scripts:

MAP_Buffer.m creates a fully configurable buffer map.

MAP_LDS.m creates a fully configurable linear distribution system.

MAP_CDS.m creates a simple circular distribution system.

All the 'Test_...' scripts run the analysis:

Test_LTS.m analyzes the linear transportation system.

Test_Buffer_Linear.m analyzes the linear buffers.

Test_Buffer_Double.m analyzes the double buffers.

Test_LDS_simplest.m analyzes the evolution of the mean realtive path occupation over time.

Test_LDS_Test1.m analyzes the different failure rates in dependence of the buffer and batch size.

Test_LDS_Test2.m analyzes the different failure rates in dependence of the buffer size and sink rate.

Test_LDS_Simplest_Graph1.m analyzes failure rates and packet deliveries in dependence of the sink and source rate.

Test_LDS_Simplest_Graph2.m visualizes package drop in dependence of the sink rate and buffer length.

Test_LDS_Simplest_Graph3.m shows the number of dropped packets in dependence of the sink rate, buffer length and batch size.

/Test_XXX are output folders that contain all results of a specific analysis.

Graphics_for_Doku.m crates certain graphics for the report.

3.6 Measurement Quantities

Packet Drop Occurrence This indicator tells if in a specific model run a packet had to be dropped or not. In other words this measurement quantity is set to one when a packet gets dropped and the specific constellation of parameters therefore describes an invalid system.

Relative Throughput T_r This quantity is defined as the total number of packets N_p delivered to the sinks divided by the simulation time t_S. It provides information on the overall capacity of the observed transportation system.

$$T_r = \frac{N_p}{t_S} \tag{1}$$

Relative Path Occupation O_p Here we have a value that is calculated by taking the sum of all packets $N_{p,i}$ in a defined path section at a specific time step and then dividing this value by the length s of the path section observed. The mean relative path occupation is the mean value of the relative path occupation over time.

$$O_p = \frac{mean(\sum_{i=1}^{n} N_{p,i})}{s} \tag{2}$$

Mean Travel Time $t_{t,m}$ To quantify the travel time of a packet we simply take the mean value of the number of time steps N_{ts} used for a packet to get from source to sink over all packets delivered.

$$t_{t,m} = mean(N_{ts,i}) \tag{3}$$

Normalized Mean Travel Time $t_{t,m}^n$ We normalize the mean travel time with the path length s from sink to source. This quantity is used for better visualization of the results of the analysis of linear transportation systems.

$$t_{t,m}^n = \frac{t_{t,m}}{s} \tag{4}$$

Failure Rate r_f For the linear distribution system we need to quantify how reliable the system delivers it's packets. Therefore we define the failure rate r_f as the number of packets dropped $N_{p,d}$ divided by the total number of packets sent N_p plus the number of packets dropped.

$$r_f = \frac{N_{p,d}}{N_p + N_{p,d}} \tag{5}$$

4 Simulation Results & Discussion

This section shows the results of the analysis of the three geometries introduced in section 3.4 in a logical order.

4.1 Linear Transportation Systems

To begin we take a look at the linear transportation system shown in fig. 3. This allows us to find out general information on the different parameters we use to build up an experiment. With this information we get an overview of meaningful intervals of our parameters for further use in more complex transportation systems.

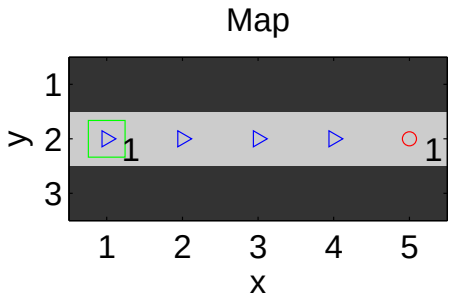


Fig. 3: The empty map of the linear transportation system.

4.1.1 Packet Drop Occurrence

As a first step we want to find out which source rate to sink rate defines a valid transportation system regarding to question 1 of section 2.

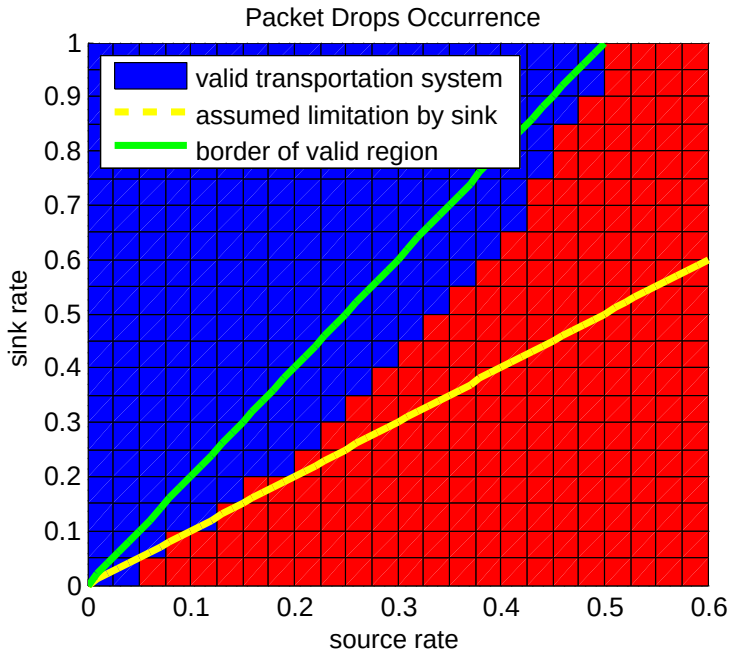


Fig. 4: Packet drop occurrence in dependence of constant sink and source rates. All blue marked fields show configurations of valid transportation Systems. Red means this sink / source rate combination leads to an invalid transportation system.

In fig. 4 we see the result of an experiment where we simulated for 200 time steps. Intuitively one would say that a transportation system is always valid as long as the sink rate is bigger than the source rate. This intuitive border is shown in yellow. In fact we see that this is not the actual border. Obviously there are other limiting factors.

Furthermore we conclude that the maximum source rate is 0.5. Higher source rates always lead to packet drops. Therefore the maximum throughput seems to be limiting the maximum source rate and might be around 0.5 what will be analyzed later in this work.

4.1.2 Relative Throughput

To answer question 2 of section 2 we take a look at the relative throughput (fig. 5). We see that the maximum is 0.485 at a sink rate of 1 and a source rate of 0.5. The theoretical maximum of the throughput which is 0.5 is not exactly reached. This might be explained by the fact, that there is a finite number of simulation steps.

As another fact we conclude that the crucial parameter for the throughput is the source rate. The sink rate has only a minor influence on the throughput. It is more of a required criteria for the packets to be removed fast enough to avoid packet drops.

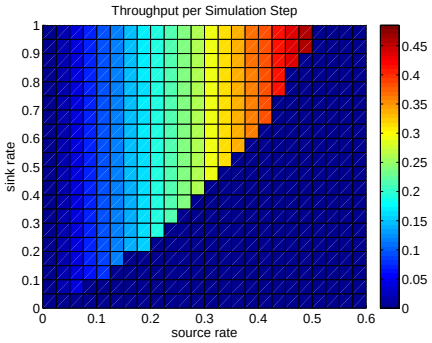


Fig. 5: Relative packet throughput in dependence of constant sink and source rates. In this plot the invalid configurations are set to zero.

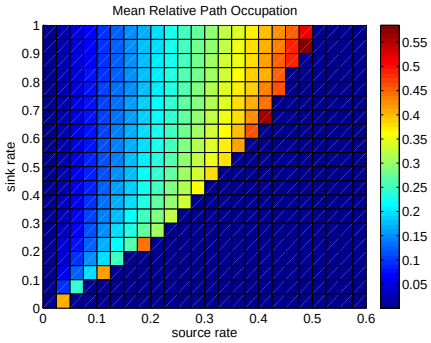


Fig. 6: Mean relative Path occupation in dependence of constant sink and source rates. In this plot the invalid configurations are set to zero.

4.1.3 Relative Path Occupation

Lets now focus on the relative path occupation. When we first look at the case of maximum throughput (upper right corner in fig. 6) we conclude that the mean path occupation is 0.492. This is slightly bellow 0.5 which is the value for maximum theoretical mobility because then only every second field is occupied and all packets can always move freely. This also answers question 3 of section 2.

For the above declared case of maximum throughput we now take a look at the path occupation over time (fig. 7). There we see that after a short transient behavior in the beginning, the occupation oscillates around 0.5, the value of maximum mobility. If we neglected the transient behavior in the beginning, the mean path occupation would get very close to that value.

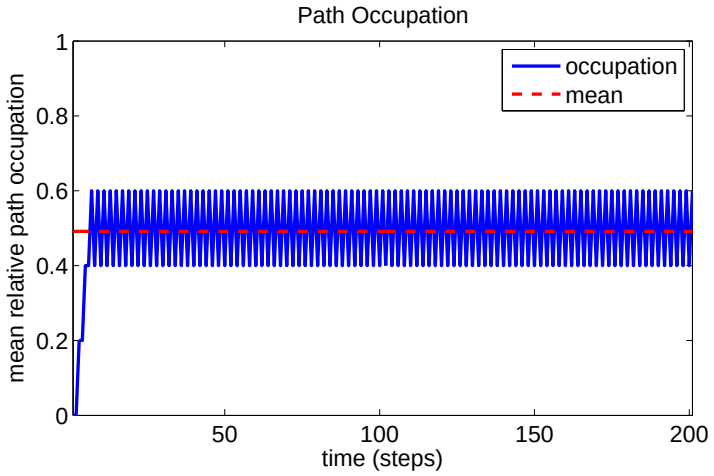


Fig. 7: Relative path occupation in blue and it's mean value shown in red over all time steps. This shows the scenario of maximum throughput where the source rate is 0.5 and the sink rate is 1.

If we go back to fig. 6 we recognize some constellations (source rate of 0.475 and sink rate of 0.9) where the mean relative path occupation is higher than 0.5 even though it is a valid transportation system. This might seam confusing in the first place. But when we take a look at fig. 8 we see that the mean occupation rises with time. This on the other hand shows that for longer simulation time, this constellation will lead to jamming, which in other words means that a packet drop will occur.

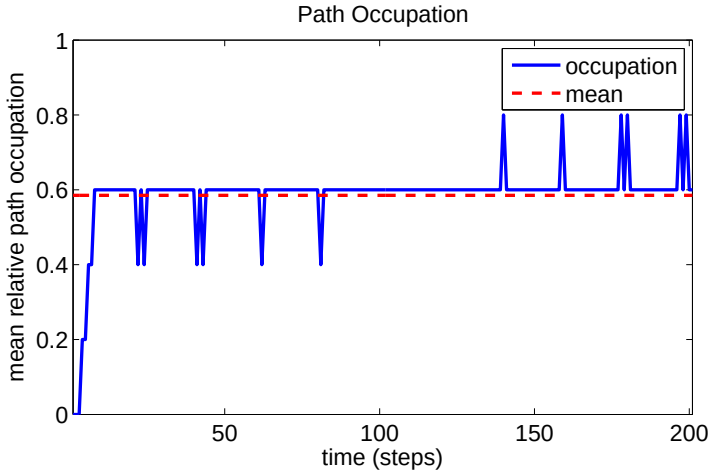


Fig. 8: The trend shows a slowly filling path. A source rate of 0.475 and a sink rate of 0.9 (dark red field in the upper right corner of fig. 6) lead to this behavior.

Out of the above stated results we presume that a mean path occupation of over 0.5 will always lead to a packet drop for a long enough simulation time.

4.1.4 Normalized Mean Travel Time

When a packet can move freely it reaches by definition a normalized mean travel time of 1. In fig. 9 it is obviously cognizable that 1 is the minimum travel time achieved. It's possible to reach minimum travel time while having maximum throughput at the same time (see fig. 5 at a source rate of 0.5 and a sink rate of 1). In other words and also to refer to question 4 of section 2 we can say that the linear transportation system can be optimized in both, throughput and travel time.

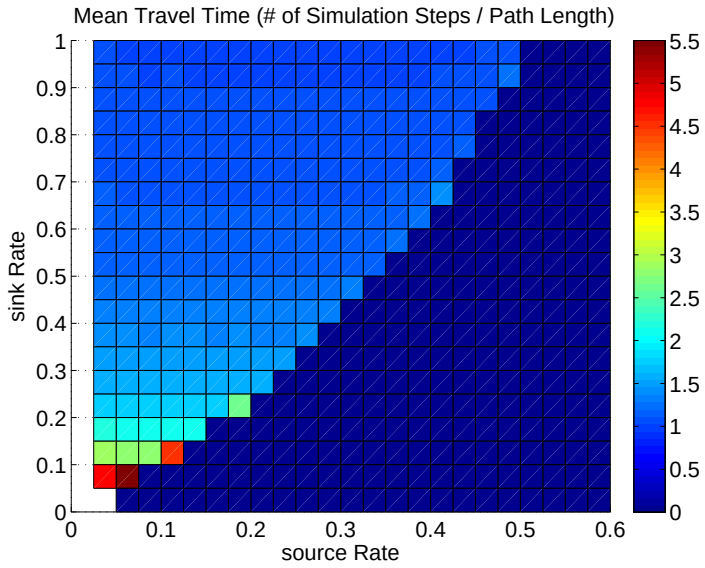


Fig. 9: Mean travel time of the packets in dependence of source and sink rate.

The mean travel time seems to only depend on the sink rate in the following manner: The smaller the sink rate, the bigger becomes the travel time. This can be explained by the unsynchronicity of the arrival at the sink and the removal of the packets. The smaller the sink rate the bigger is of course the unsynchronicity.

4.2 Buffer Systems

We found out that a linear transportation system can only be operated at certain sink and source rates. But what if the external circumstances require the system to deal with higher source rates for short times? For example a post packet center where the packets are not arriving continuously but rather in batches of a truck load.

This is exactly where the idea of a buffer comes to play. It should be able to store the incoming packets until the sink is again able to process packets. We define the *buffer size* as the number of fields (including sources, waiting fields, junctions and sinks) that are able to store packets

in a buffer.

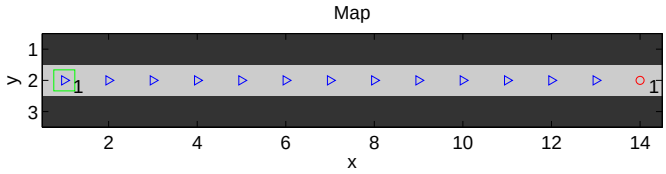


Fig. 10: Linear buffer with a size of 14.

To test buffers we must clarify for what sink and source rates it makes sense at all to use a buffer (see question 5 of section 2). For this reason let's take a look at fig. 4. For a constant sink rate we consider source rates as well chosen as long as the mean value lies in the blue region. A mean value lying in the red region doesn't make sense. In this case the buffer would get filled up in a finite amount of time. This would lead to a packet drop and consequently to an invalid buffer. Peak source rates on the other hand have to lie in the red region. Otherwise a temporal fill up would never happen and the buffer would be senseless. We now have no longer constant source rates but rather *source rate signals*.

The upper limit of a source rate is 0.5 which was shown above. Also for our consideration of buffer systems we take this as the global maximum of our source rate signals.

For our analysis we choose a constant sink rate of 0.5. For the mean value of our source rate signal we have to choose a value smaller than 0.25. With temporarily source rates of over 0.35 we sure are in the red region of fig. 4. With those considerations for our source rate signal we will have enough dynamic in the system.

To generate the desired source rate signal we used the concept of modified pulse length as it can be seen in fig. 11.

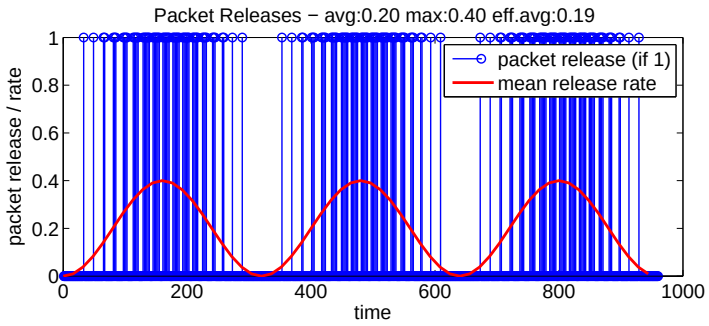


Fig. 11: Time depending source rate signal as it is used in the following simulations.

Looking at the blue lines a one means a packet enters the system at the source at this specific time step. The red line shows approximatively the

current mean source rate. Due to the discrete pulses the effective mean source rate is 0.1938 (calculated) and the maximum is 0.5. In general for discrete operations it is difficult to define a rate because the derivative doesn't exist.

In the following sections we discuss two types of buffers: the linear buffer and the double buffer.

4.2.1 Linear Buffers

The simplest buffer is the linear buffer as it is shown in fig. 10. We want to find out what buffer size our above stated conditions (constant sink rate and the defined source rate signal from above) require.

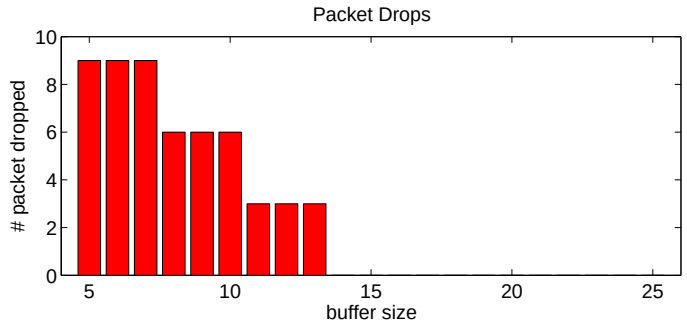


Fig. 12: Counts of packet drops over a simulation time of 960 time steps. 14 and more as a buffer size leads to a valid transportation system with the given source rate signal.

In our case we need a buffer with a size of 14 (see fig. 12). To answer question 6 of section 2 we tried to find a formula to calculate this minimum buffer size but this seems to be very difficult. The minimum buffer size is strongly dependent on the characteristics of the source rate signal. On the other hand it is easy to find the minimum buffer size by simulating.

Buffer Occupation Let's take a look at the time depending buffer occupation to find out, if it lets us make any conclusions on packet drops.

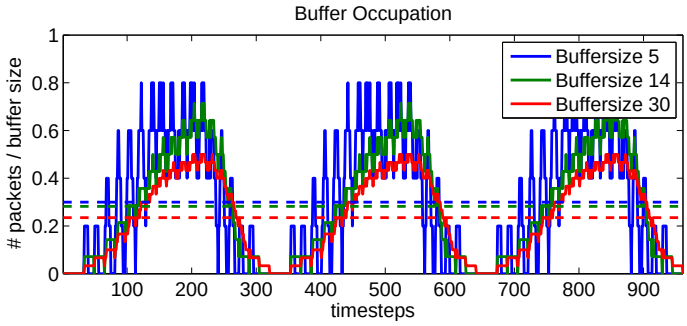


Fig. 13: Buffer occupation over time. Solid lines show the occupation over time, dashed lines the mean value. Blue shows a buffer that causes drops, green is the optimum buffer and red is an over sized buffer both causing no packet drops.

We conclude that the smaller the buffer, the bigger are the peak occupations of it. All buffers can get totally emptied at some points. Referring to question 7 of section 2 a direct conclusion on packet drops can't be made: It seems that short time occupations of over 0.5 are possible without causing packet drops (see the green buffer in fig. 13). Still the blue case in fig. 13 shows that for longer occupations of over 0.5 packets will get dropped.

Mean Travel Time The travel time of course increases with the size of a linear buffer. Its minimum is defined directly by the size of the buffer.

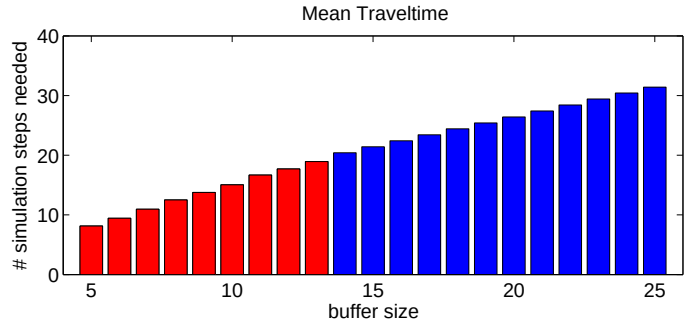


Fig. 14: Valid buffer systems are shown in blue. Red means a packet had to be dropped which makes the system invalid.

By looking at fig. 14 it gets obvious that if a buffer is low occupied the mean travel time becomes unnecessary big. In the following more advanced buffer design we try to improve this.

4.2.2 Double Buffers

In order to shorten the mean travel time we have to shorten the shortest possible path from source to sink. At the same time the buffer performance

must stay the same. The solution to this problem can be seen in fig. 15.

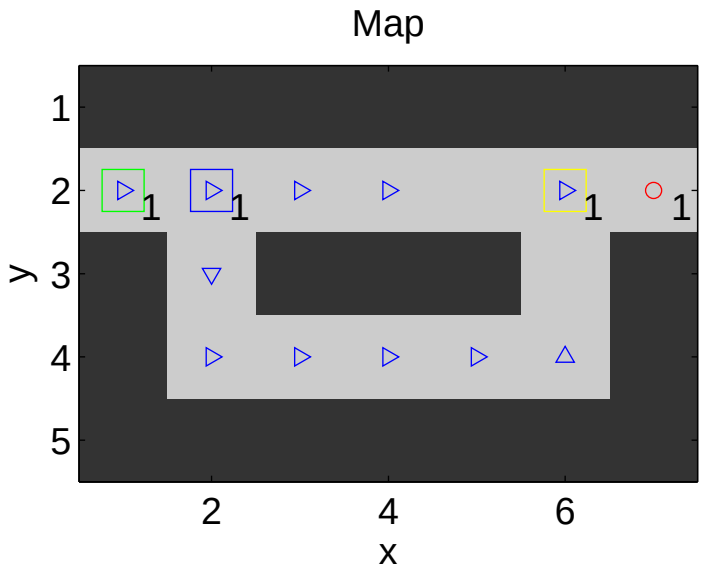


Fig. 15: Double buffer with a size of 14. The direct path is referred to as 'path one' and the U shaped path as 'path two'.

A second parallel path has to be built. The JND is controlled by a rule that tries to move the packets directly to the right in the first place. Only if path one (see caption of fig. 15) is blocked it sends its oncoming packets downwards. The JNC on the other hand prioritizes path 2 such that this path gets emptied first. These rules allow a complete discharging of the buffer.

As seen in fig. 16 no drops occur for the same source rate signal as in section 4.2.1.

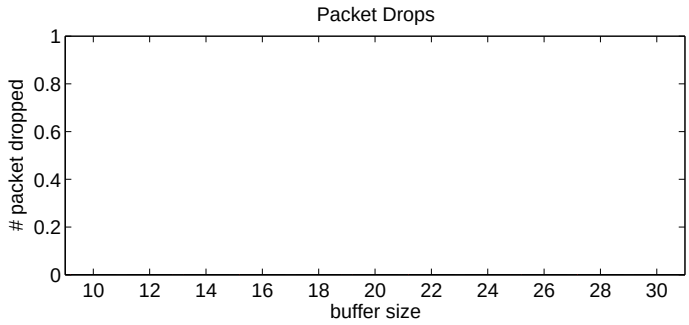


Fig. 16: For the same conditions as in fig. 12 no drops occur for a double buffer.

Figure 17 shows that with the double buffer design already a size of 10 satisfies the requirements of a source rate signal as it is shown in fig. 11. The reason is that in this design the buffer size can be capitalized better

due to the intelligent junctions.

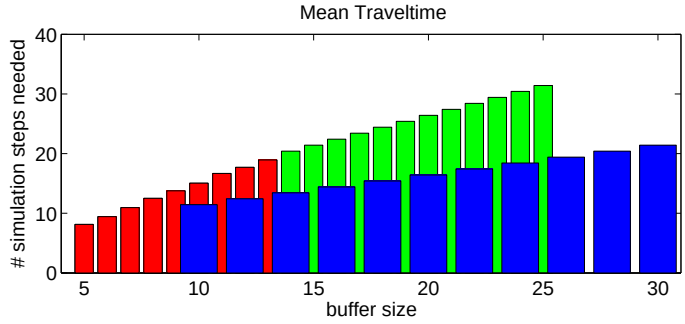


Fig. 17: Blue bars show double buffers, green show linear buffers and red bars show invalid buffers. Double buffers can be much smaller in size for the same source rate signal. It can also be seen clearly that the mean travel time is always smaller for double buffers.

Also the mean travel time is always shorter for a double buffer as it can be seen in fig. 17. And if we take a look at the occupation (fig. 18) we see that the double buffer is always less occupied than the linear buffer. This can be explained by the fact that the double buffer has a much smaller mean travel time. In other words the packets stay less long in the buffer and therefore the buffer's occupation decreases.

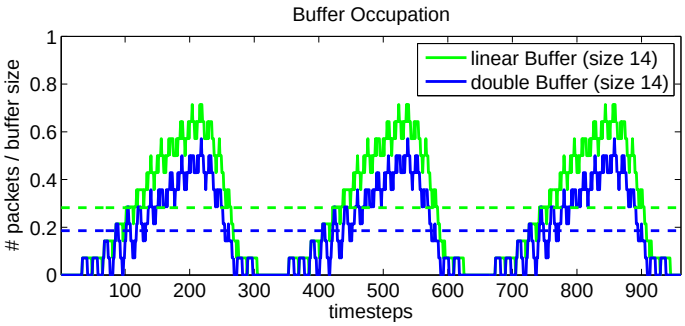


Fig. 18: Occupation in dependence of the time for both the linear and the double buffer system. Dashed lines show the mean value of the occupation.

As a final conclusion in terms of question 8 of section 2 it can be said that with a simple redesign of the linear buffer all properties can be improved.

4.3 Linear Distribution Systems

A typical map of a linear distribution system can be seen in fig. 19. Every sink has its own buffer to compensate for fluctuations in the incoming packet rates. Packets now have addresses and have to be delivered to their assigned target sink.

This is a new challenge for the implementation of the source. Now there has to be a rule for the distribution of the different target numbers to the incoming packets. We solved this problem by creating so called *batches* with a number of packets to which we refer to as the *batch size*. The source sends out one batch after another. All packets of one batch have the same target number. Two batches following each other never have the same target number.

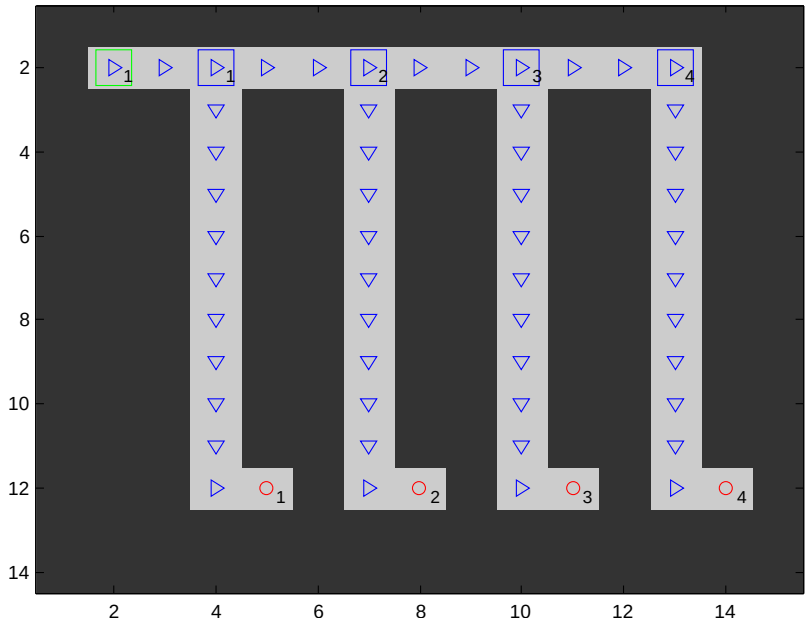


Fig. 19: Linear distribution system with 4 targets. The buffer for each sink has a size of 10.

The batch size is a parameter that can be adjusted as well as the buffer size of the sinks. Also the source and sink rates are still variable parameters.

Lets now take a look at a simple case where the sink and source rates are held constant at 0.3 and 0.15 respectively. This also helps to understand the use of batches. For now every packet gets a new target number (batch size = 1) and the buffer size is two.

Figure 20 shows the mean relative path occupation as it is defined in equation (2). As well as for the linear transportation system the occupation is fluctuating but here we have a bigger amplitude (compare to fig. 7, 8).

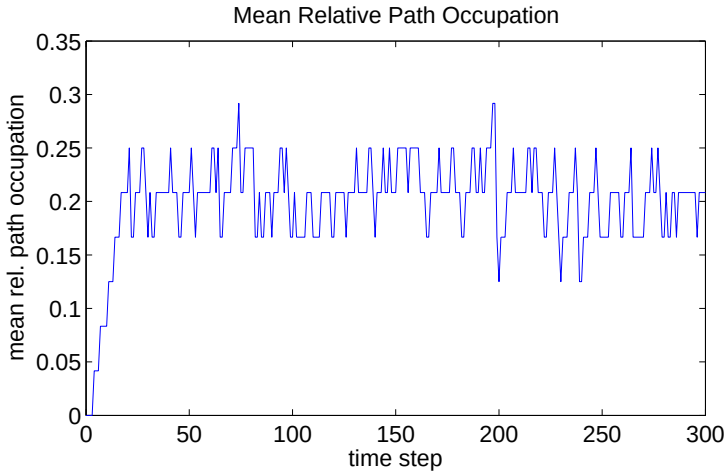


Fig. 20: The mean relative path occupation of a system as it is shown in fig. 19 plotted over time. The source rate here was 0.3 and the sink rate 0.15.

If we only look at fig. 20 we might have the impression that everything is working fine and no packets have to get dropped. But what if for example ten packets with the same address enter the system right after each other. We can not predict the behavior of the validity for scenarios like this. Therefore we have to introduce the batch which simulates a series of packets with the same address.

4.3.1 Influence of the Batch Size

For the following experiment we wanted to test the linear distribution system at full load. This is the reason why the source rate was chosen to be 0.5 (see section 4.1.1). Figure 21 shows the failure rate as it is defined in equation (5) in dependence of the batch and buffer size. Only if the failure rate is $r_f = 0$ the system is valid after definition 1.

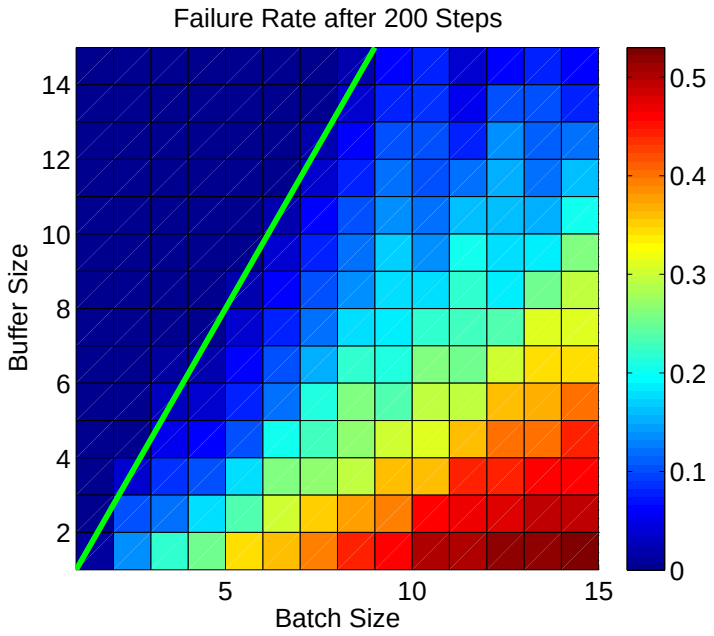


Fig. 21: Failure rate in dependence of buffer and batch size. The green line is the limit above which 95% of the packets reach their target. For this experiment the sink rate was constant 0.2 and the source rate 0.5. The system used is shown in fig. 19.

The green line in fig. 21 clearly shows that the bigger the batch size is the bigger has the buffer size to be chosen so that the system is still valid.

Another interesting fact is that if the batch size is one (every packet entering the system gets another target number) the buffer size can easily be chosen as one as well. But if we increase the batch size the buffer size has to increase faster for the system to stay valid.

4.3.2 Source and Sink Rate Influences

Because the source rate can be split into four different sinks the sink rate can be chosen smaller than the source rate. To find out how much smaller the sink rate can be chosen we analyze our above stated simple system regarding the sink and source rates. The criterion we use is the validity after definition 1. In fig. 22 the failure rate f_r is shown in dependence of sink and source rate. For a system to be valid f_r must be zero. If we take a look at the green line in fig. 22 which is about the limit above which 95% of the packets reach their target, we see that it has a slope of a bit more than 0.25. In other words - as one would expect - for four sinks the sink rate can be chosen as about a fourth of the source rate.

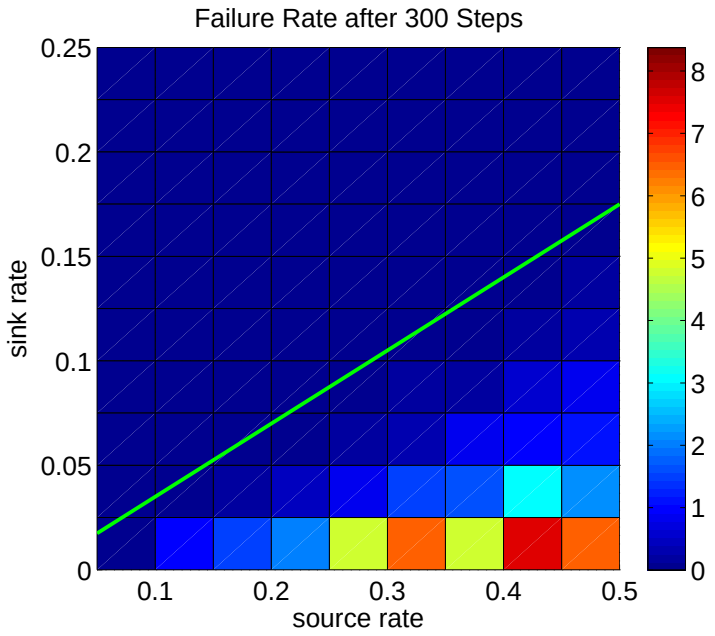


Fig. 22: Source and sink rate combinations to find out about the validity of a four sink linear distribution system. The system is valid as long as the failure rate is zero. The green line is the limit above which 95% of the packets reach their target.

Now we do not only want to have valid systems but we also want to deliver as many packets as possible. Therefore we take a look at the number of packets delivered. Figure 23 shows how many packets reached their target after 300 time steps of simulation. The green line is the same as in fig. 22 and tells us that above this line we have a valid system.

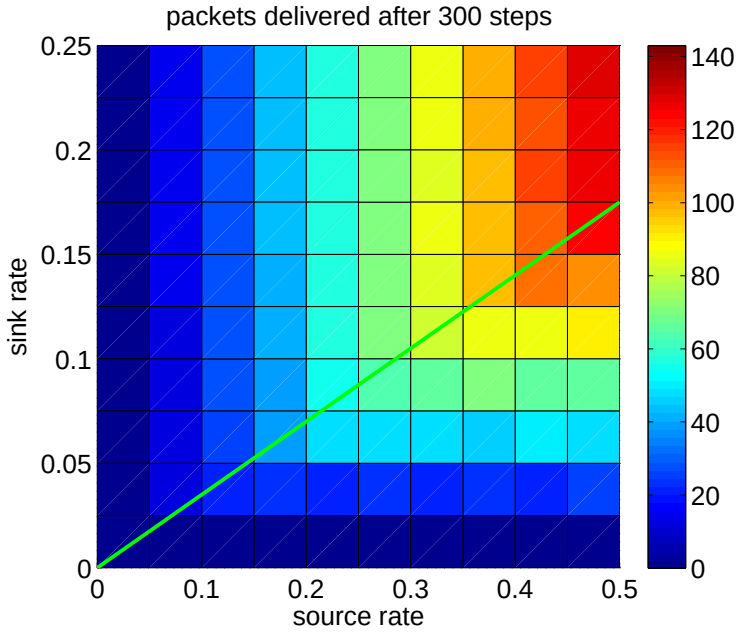


Fig. 23: Source and sink rate combinations to find out about the transport capacity for a four sink linear distribution system. The green line is the limit above which 95% of the packets reach their target.

The number of delivered packets is of course proportional to the source rate as long as there are no packet drops (underneath the green line).

5 Summary & Outlook

5.1 Summary

We found out that it is very important to have a good definition of a valid transportation system (see definition 1). Otherwise it is impossible to characterize the found results.

The validity of a system is mainly depending on the sink rate to source rate ratio. Generally we can state the following three conclusions regarding to this question:

- For low source and sink rates the sink rate is the limiting factor. This leads to the yellow line in fig. 4.
- For higher source and sink rates the DTP is limiting. The maximum source rate seams to be around 0.5.
- If the parameters are chosen in a way that the green line in fig. 4 is taken as a limit, one should roughly always have a valid transportation system. In mathematical terms this means:

$$source\ rate < 2 \cdot sink\ rate \tag{6}$$

Regarding the questions about buffers we found out that it is essential to firstly define a system state where a buffer makes sense at all. This state must contain a transient behavior of the source rate signal. The mean value of this signal must lie in the blue region of fig. 4 and peak values have to lie in the red region.

When the source rate is fluctuating as seen in fig. 11 also the buffer capacity and geometry have a big influence on the validity of the system. Double buffers are in general more effective than linear buffers.

As a final conclusion we can transform the concept of validity into the question of design and state that the design of any transportation system is massively depending on the boundary conditions. This means that inflow and outflow conditions mainly define the system design.

5.2 Outlook

5.2.1 Sinks and Sources

For this work we analyzed the impact of a varying source rate signal (see fig. 11). Another approach could be to do the same with the sink rate. It's possible that a whole new dynamic could appear that we didn't see yet. It might also be interesting to find out if an unstable oscillation of the path occupation would occur for certain source / sink rate signal frequencies.

Obviously a double buffer can get stressed more than we thought when we designed the source rate signal. Therefore one could think of a more aggressive source rate signal or a smaller sink rate. With this new constellations one could analyze other buffer designs and improve the buffer characteristics even more.

5.2.2 Buffer Layout

To further improve the buffer behavior we thought about much more buffer geometries than discussed in 4.2. Because the time was short we couldn't analyze those buffers in this work.

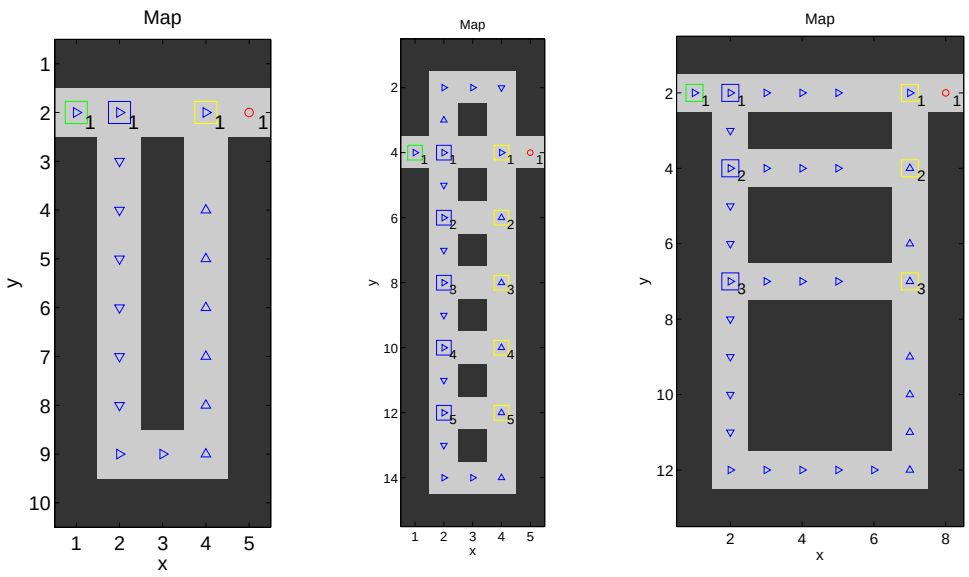


Fig. 24: Three different possible buffer systems. On the left side is a so called *short response buffer*. In the middle is the *massive paralization buffer* and on the right side the 2^n *multi depth buffer*.

With a *short response buffer* (shown on the left side of fig. 24) the mean travel time of a source rate signal with a small peak density could be improved though there is a certain buffer capacity to dampen peaks. For a signal with a lot of peaks this buffer might be a bad choice because a lot of packets would have to take the much longer way through the buffer.

The *massive paralization buffer* (shown in the middle of fig. 24) could shorten the long loop ways of the above described buffer because a lot of short buffer ways are possible. This layout will utilize its capacity very well due to the numerous intelligent junctions. A good control of these nodes will not be easy to realize. In the end this means that the calculation costs for this buffer will be large.

For a strongly varying source rate signal we thought that a 2^n *multi depth buffer* as it is shown on the right side of fig. 24 might be useful. The size of loops in this buffer grow exponentially. For a low frequent source rate signal this buffer features a short shortest path and for high amplitude source rate signals it has some long loops for buffering. Due to the exponential growth of the loops it contains much less junctions than the massive paralization buffer for the same buffer capacity.

5.2.3 Distribution Systems

A big problem of the linear distribution system is that it can block quickly if one sink is jammed. A good solution to this problem is the so called *circular distribution system* as it is shown in fig. 25.

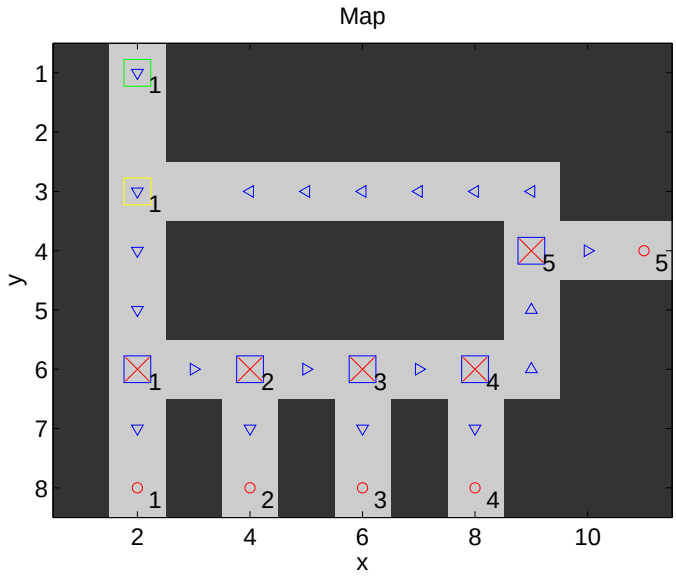


Fig. 25: Circular distribution system.

If one sink is blocked the packets can just pass by and start a full circle journey to try to enter the target sink another time. Like this the mentioned packet wouldn't block the whole system and the packets behind it can still reach their target.

In all discussed distribution systems the sink path could get equipped with buffer systems to improve their flexibility. Circular systems could additionally have a short response buffer within the distributing ring to expand the capacity without increasing the mean travel time around the ring massively.

6 Matlab Code

6.1 bin

code/initialize_simulation.m

```
%% Initialize Simulation
% this scripts sets all constants

% check for reconfiguration
if exist('system_initialized','var')
    disp('reinitialization avoided');
    break;
end
system_initialized = true;
disp('initialize Simulation');

```

```
%%  - map constants
% do not change direction/empty constant.
c_em = 0; % Empty field
% Directions (up, down, left, right)
c_up = 1; c_dn = 2; c_lt = 3; c_rt = 4;
c_sn = 91; % Sinks
c_wt = 92; % Waiting field
c_er = 93; % Error field

%%  - packet list constants
c_pck_id = 1;
c_pck_dest_id = 2;
c_pck_started_at_timestep = 3;
c_pck_started_at_sourceid = 4;
c_pck_delivered_at_sinkid = 5;
c_pck_delivered_at_timestep = 6;
c_pck_numcols = 6;

%%  - source constants
c_src_id = 1;
c_src_coordx = 2;
c_src_coordy = 3;
c_src_adressrule = 4;
c_src_adressrule_r1 = 5;
c_src_adressrule_r2 = 6;
c_src_adressrule_r3 = 7;
c_src_adressrule_r4 = 8;
c_src_rule = 9;
c_src_rule_rate_rate = 10;
c_src_rule_rate_counter = 11;
c_src_rule_controlled_pckcounter = 12;
c_src_packetssent = 13;
c_src_packetsdropped = 14;
c_src_numcols = 14;

% source rules
c_src_rule_rate = 0;
c_src_rule_controlled = 1;

% source addressrules
c_src_adressrule_random = 0;
c_src_adressrule_random_batch = 1;
c_src_adressrule_random_batch_countermax =
   c_src_adressrule_r1;
```

```
55 c_src_adressrule_random_batch_counter =
      c_src_adressrule_r2;
56 c_src_adressrule_random_batch_currentaddr =
      c_src_adressrule_r3;
57
58 %%  - sink constants
59 c_snk_id = 1;
60 c_snk_coordx = 2;
61 c_snk_coordy = 3;
62 c_snk_rule = 4;
63 c_snk_rule_constant_rate = 5;
64 c_snk_rule_constant_counter = 6;
65 c_snk_numcols = 6;
66
67 c_snk_rule_immediately = 2;
68 c_snk_rule_constant = 0;
69 c_snk_rule_batch = 1;
70
71 %%  - converging junction constants
72 c_jnc_id = 1;
73 c_jnc_coordx = 2;
74 c_jnc_coordy = 3;
75 c_jnc_rule = 4;
76 c_jnc_p1 = 5;
77 c_jnc_p2 = 6;
78 c_jnc_p3 = 7;
79 c_jnc_p4 = 8;
80 c_jnc_numcols = 8;
81
82 c_jnc_rule_Random = 1;
83 c_jnc_rule_TakeIfAnyRandom = 2;
84 c_jnc_rule_TakeIfAnyPriorized = 3;
85
86 %%  - diverging junction constants
87 c_jnd_id = 1;
88 c_jnd_coordx = 2;
89 c_jnd_coordy = 3;
90 c_jnd_indir = 4;
91 c_jnd_rule = 5;
92 c_jnd_p1 = 6;
93 c_jnd_p2 = 7;
94 c_jnd_p3 = 8;
95 c_jnd_p4 = 9;
96 c_jnd_numcols = 9;
97
```

```
c_jnd_rule_Random = 1;
c_jnd_rule_SearchFreeWay = 2;
c_jnd_rule_SearchFreeWayPriorized = 3;
c_jnd_rule_TakewayAddrMap = 4;

%%  - initialize lists
jnd_list = [];
jnc_list = [];
jnd_addr_list = [];
```

code/run_simulation.m

```
% Simulation Script
% This script performs the main simulation
%  It can either be manually called with an individual configuration
%   (see "manual configuration" below)
%  Or, as it is mostly done, called by external test scripts. In this case
%   a map must be created and all the configuration parameter must be set
%   before calling this script


%% Manual Configuration
if ~exist('testing_in_progress','var')
    % only allow manual configuration if no superior script is controlling
    clear; clc;
    initialize_simulation

    % Set simulation parameters
    slm_enable_plot_sim = 1;
    slm_num_steps = 500;
    slm_plot_waiting_time = 0.05;

    % Create Map
    run ../maps/MAP_CDS
    %run ../maps/MAP_Buffer
end

% Snap shot of map requested? - does not work in manual mode
if ~exist('slm_make_crowded_map_snapshot','var')
    slm_make_crowded_map_snapshot = 0;
    slm_make_crowded_map_snapshot_rel_occupation =
```

```
        0.3;
30      slm_make_crowded_map_snapshot_filename = 'map'
        ;
31  end
32
33  %% Initialization
34
35  % Clear Counters
36  src_list(:,c_src_adressrule_random_batch_counter)
      = 0;
37  src_list(:,
      c_src_adressrule_random_batch_currentaddr) = 0;
38  src_list(:,c_src_packetsdropped) = 0;
39  src_list(:,c_src_packetssent) = 0;
40  src_list(:,c_src_rule_controlled_pckcounter) = 0;
41  src_list(:,c_src_rule_rate_counter) = 0;
42
43  snk_list(:,c_snk_rule_constant_counter) = 0;
44
45  % Define Packet list
46  %  used for packet accounting
47  pck_list = zeros(0,c_pck_numcols);
48  pck_list_history = zeros(0,c_pck_numcols,0);
49  pck_id_last = 0;
50
51  % Define Packet map
52  %  has the packet's id in the according field, 0
      if empty
53  pck_map = zeros(size(map));
54  pck_map_history = [];
55  pck_map_history(:,:,1) = pck_map;
56
57
58  % Predefine helping matrices, for simplier packet
      move execution
59  map_width = size(map,2);
60  map_height = size(map,1);
61  map_zeros = zeros(size(map));
62  map_zero_row = zeros(1,map_width);
63  map_zero_column = zeros(map_height, 1);
64
65  % Check special source rule predefinition
66  if exist('src_rule_controlled_controllist','var')
67      if slm_num_steps > size(
          src_rule_controlled_controllist,2)
```

```
68          disp('SRCRULE: more simulation steps than
               specified for the controlled packet
               generation');
69      end
70  end
71
72  %% Simulation
73  for i = 1:slm_num_steps
74      %%  - generate frequently needed variables
75      pck_exists_map = pck_map ~= map_zeros;
76
77      %%  - clear old packets at sinks
78      % Checks all sinks occupied by packets,
           whether they should be cleared
79      [row,col,v] = find(pck_map(map == c_sn)); %v
           contains the IDs of the packets
80      for k=1:length(v)
81          pck_cur = pck_list(v(k),:);
82          [pck_cur_coordy, pck_cur_coordx, tmp] =
               find(pck_map == pck_cur(c_pck_id));
83
84          sink_found = 0; % control variable for for
               -loop
85          for l = 1:size(snk_list,1)
86              if snk_list(l,c_snk_coordx) ==
                   pck_cur_coordx && snk_list(l,
                   c_snk_coordy) == pck_cur_coordy
87                  snk_cur = snk_list(l,:);
88
89                  % Takes packet out of network due
                       to the rule of the sink
90                  SNKRULE;
91                  % uses & updates    : snk_cur,
                       pck_list, pck_map, etc.
92
93                  sink_found = 1;
94                  snk_list(l,:) = snk_cur; %#ok<
                       SAGROW>
95              end
96          end
97          % Checks correct declaration of sinks in
               map
98          if (sink_found == 0)
99              disp('error: sink not found');
100         end
```

```
101
102         pck_list(v(k),:) = pck_cur;
103     end
104
105     %%  - switch diverging junctions
106     % Sets the junction direction (blue arrow) of
          all diverging junctions
107     for k = 1:size(jnd_list,1)
108         jnd_cur = jnd_list(k,:);
109         x = jnd_cur(c_jnd_coordx);
110         y = jnd_cur(c_jnd_coordy);
111         out_dir = c_er; % Initial direction: '
              error'
112
113         % In which direction should packet be
              pushed?
114         JNDRULE; % Determinate 'out_dir' due Rule
              of Junction
115         % uses    : jnd_cur, x, y, etc.
116         % returns : out_dir
117
118         % Set junction direction due junction
              decision
119         %  (the actual move is performed in update
               position)
120         map(y,x) = out_dir;  %#ok<SAGROW>
121     end
122
123     %%  - update position
124     % Moves all packet according to the path
          directions (blue arrows) if
125     %  path is not blocked by another packet.
126
127     % The new packet situation is temporaly stored
           in pck_map_n to perform
128     %  consitent update regardless of the order
          the updates are done.
129     pck_map_n = pck_map;
130
131     % down
132     m_map = (map == c_dn); % 1 where we need to
          move down, 0 else
133     m_map = m_map & ~(m_map & [pck_exists_map(2:
          map_height,:) ; map_zero_row]); % check
          ahead for collisions
```

```
134     m_map = m_map & ([map(2:map_height,:) ;
            map_zero_row] ~= c_em); % Check ahead for
            empty places
135     m_map = m_map .* pck_map;
136     p_map = [map_zero_row ; m_map(1:(map_height-1)
            ,:)];
137     pck_map_n = pck_map_n - m_map + p_map;
138     % up
139     m_map = (map == c_up); % 1 where we need to
            move down, 0 else
140     m_map = m_map & ~(m_map & [map_zero_row ;
            pck_exists_map(1:(map_height-1),:)]); %
            check ahead for collisions
141     m_map = m_map & ([map_zero_row ; map(1:(
            map_height-1),:)] ~= c_em);
142     m_map = m_map .* pck_map;
143     p_map = [m_map(2:map_height,:) ; map_zero_row
            ];
144     pck_map_n = pck_map_n - m_map + p_map;
145     % left
146     m_map = (map == c_lt);
147     m_map = m_map & ~(m_map & [map_zero_column ,
            pck_exists_map(:,1:(map_width-1))]);%[
            pck_exists_map(:,2:map_width) ,
            map_zero_column]);
148     m_map = m_map & ([map_zero_column , map(:,1:(
            map_width-1))] ~= c_em);
149     m_map = m_map .* pck_map;
150     p_map = [m_map(:,2:map_width) ,
            map_zero_column];
151     pck_map_n = pck_map_n - m_map + p_map;
152     % right
153     m_map = (map == c_rt);
154     m_map = m_map & ~(m_map & [pck_exists_map(:,2:
            map_width) , map_zero_column]);%[
            map_zero_column , pck_exists_map(:,1:(
            map_width-1))]);
155     m_map = m_map & ([map(:,2:map_width) ,
            map_zero_column] ~= c_em);
156     m_map = m_map .* pck_map;
157     p_map = [map_zero_column , m_map(:,1:(
            map_width-1))];
158     pck_map_n = pck_map_n - m_map + p_map;
159
160     %%  - process converging junctions
```

```
161     % After the packets are moved, all converging junctions check the
162     %  neighbouring waiting field for packets and decide according to their
163     %  rule if and which packet should be taken in.
164     for k = 1:size(jnc_list,1)
165         jnc_cur = jnc_list(k,:);
166         x = jnc_cur(c_jnc_coordx);
167         y = jnc_cur(c_jnc_coordy);
168         if(pck_exists_map(y,x) == 0) % is junction empty?
169             in_dir = c_er; % Inital direction: 'error'
170 
171             % Packet from which Direction should be taken in?
172             %  (all incoming lines must be waiting positions)
173             JNCRULE % Determinate 'in_dir' due Rule of Junction
174             % uses    : jnc_cur, x, y, etc.
175             % returns : in_dir
176 
177             % Takes packet in
178             switch in_dir
179                 case c_up
180                     if(pck_map(y-1,x) ~= 0)
181                         pck_map_n(y,x) = pck_map(y-1,x);
182                         pck_map_n(y-1,x) = 0;
183                     end
184                 case c_dn
185                     if(pck_map(y+1,x) ~= 0)
186                         pck_map_n(y,x) = pck_map(y+1,x);
187                         pck_map_n(y+1,x) = 0;
188                     end
189                 case c_rt
190                     if(pck_map(y,x+1) ~= 0)
191                         pck_map_n(y,x) = pck_map(y,x+1);
192                         pck_map_n(y,x+1) = 0;
193                     end
194                 case c_lt
```

```
195                     if(pck_map(y,x-1) ~= 0)
196                         pck_map_n(y,x) = pck_map(y
                               ,x-1);
197                         pck_map_n(y,x-1) = 0;
198                     end
199                 otherwise
200                     %No packet is taken in
201             end
202
203         end
204     end
205
206     %%  - perform iteration step:
207     % Now that all packet-moving updates are done,
            the temporally packet
208     %  situation is released
209     pck_map = pck_map_n;
210
211     %%  - create new packets at source points
212     % All packet source are checked, whether they
           have to release a new packet
213     for k = 1:size(src_list,1)
214         src_cur = src_list(k,:);
215         x = src_cur(c_src_coordx);
216         y = src_cur(c_src_coordy);
217
218         % Should a packet be created?
219         SRCRULE; % Determinates 'createPacket' due
               Rule of Source
220         % uses    : src_cur, x, y, etc.
221         % returns : createPacket
222
223         % if a packet should be created
224         if createPacket == 1
225             % generate id
226             newid = pck_id_last + 1;
227             pck_id_last = newid;
228
229             % create packet list entry
230             newpacket = zeros(1,c_pck_numcols);
231             newpacket(c_pck_id) = newid;
232             newpacket(c_pck_started_at_timestep) =
                    i;
233             newpacket(c_pck_started_at_sourceid) =
                    src_cur(c_src_id);
```

```
234
235             % Where should this packet be sent?
236             SRCADRRULE;
237             % Sets 'newpacket(c_pck_dest_id)' due
                   to Rule of Source
238
239             pck_list = [pck_list; newpacket]; %#ok
                   <AGROW>
240
241             % create packet map entry
242             pck_map(y,x) = newid;
243         end
244
245         src_list(k,:) = src_cur;  %#ok<SAGROW>
246     end
247
248     %%  - update history
249     % all simulation steps are stored for post-
          evaluation:
250     %  pck_map and pck_list
251     pck_map_history(:,:,i+1) = pck_map;
252     %adapt pck_list_history size first
253     for k=size(pck_list_history,1)+1:size(pck_list
          ,1)
254         pck_list_history(k,:,:) = zeros(size(
              pck_list_history,2),size(
              pck_list_history,3));
255     end
256     pck_list_history(:,:,i) = pck_list;
257
258     %%  - Snapshot
259     if slm_make_crowded_map_snapshot == 1
260         rel_occup = sum(sum(pck_map ~= 0)) / sum(
              sum(map ~= 0));
261         if (rel_occup >
              slm_make_crowded_map_snapshot_rel_occupation
              ) || (
              slm_make_crowded_map_snapshot_rel_occupation
               == -1)
262             slm_enable_plot_sim = 1;
263         end
264     end
265
266     %%  - Graphical Output
267     % Plot the entire network with all packets in
```

```
    it
268 if slm_enable_plot_sim == 1
269
270     % Constants for plotting
271     P_ArrowSize = 6;
272     P_MarkingBoxSize = 20;
273     P_PacketSize = 15;
274 P_DestTestOffset = -0.2;
275     P_mapFeaturesId_FontSize = 8;
276     P_mapFeaturesId_offset = 0.2;
277
278     % Initialization
279     f=figure(1);
280     clf
281
282     % Plot background
283     imagesc(map~=0);
284     colormap([[0.2 0.2 0.2];[0.8 0.8 0.8]]);
285     hold on
286
287     % Plot arrows and other signs of the 'map'
288     [r,c]=find(map == c_up);
289     plot(c,r,'^','MarkerSize',P_ArrowSize);
290     [r,c]=find(map == c_dn);
291     plot(c,r,'v','MarkerSize',P_ArrowSize);
292     [r,c]=find(map == c_lt);
293     plot(c,r,'<','MarkerSize',P_ArrowSize);
294     [r,c]=find(map == c_rt);
295     plot(c,r,'>','MarkerSize',P_ArrowSize);
296     [r,c]=find(map == c_sn);
297     plot(c,r,'or','MarkerSize',P_ArrowSize);
298     [r,c]=find(map == c_er);
299     plot(c,r,'xr','MarkerSize',
          P_MarkingBoxSize);
300
301     % Plot sources
302     for k = 1:size(src_list,1)
303         src_cur = src_list(k,:);
304         x = src_cur(c_src_coordx);
305         y = src_cur(c_src_coordy);
306         plot(x,y,'sg','MarkerSize',
              P_MarkingBoxSize);
307         text(x+P_mapFeaturesId_offset,y+
              P_mapFeaturesId_offset,num2str(
              src_cur(c_src_id)),'FontSize',
```

```
                P_mapFeaturesId_FontSize);
308         end
309         % Plot converging junctions
310         for k = 1:size(jnc_list,1)
311             jnc_cur = jnc_list(k,:);
312             x = jnc_cur(c_jnc_coordx);
313             y = jnc_cur(c_jnc_coordy);
314             plot(x,y,'sy','MarkerSize',
                    P_MarkingBoxSize);
315             text(x+P_mapFeaturesId_offset,y+
                    P_mapFeaturesId_offset,num2str(
                    jnc_cur(c_jnc_id)),'FontSize',
                    P_mapFeaturesId_FontSize);
316         end
317         % Plot diverging junctions
318         for k = 1:size(jnd_list,1)
319             jnd_cur = jnd_list(k,:);
320             x = jnd_cur(c_jnd_coordx);
321             y = jnd_cur(c_jnd_coordy);
322             plot(x,y,'sb','MarkerSize',
                    P_MarkingBoxSize);
323             text(x+P_mapFeaturesId_offset,y+
                    P_mapFeaturesId_offset,num2str(
                    jnd_cur(c_jnd_id)),'FontSize',
                    P_mapFeaturesId_FontSize);
324         end
325         % Plot sinks
326         for k = 1:size(snk_list,1)
327             snk_cur = snk_list(k,:);
328             x = snk_cur(c_snk_coordx);
329             y = snk_cur(c_snk_coordy);
330             text(x+P_mapFeaturesId_offset,y+
                    P_mapFeaturesId_offset,num2str(
                    snk_cur(c_snk_id)),'FontSize',
                    P_mapFeaturesId_FontSize);
331         end
332
333         % Plot packages
334         [r,c]=find(pck_map);
335         plot(c,r,'sr','MarkerFaceColor','r','
                MarkerSize',P_PacketSize);
336         for ip=1:length(c)
337       pck_id = pck_map(r(ip),c(ip));
338             text(c(ip),r(ip),num2str(pck_id));
339             text(c(ip)+P_DestTestOffset,r(ip),
```

```
                num2str(pck_list(pck_id,
                c_pck_dest_id)),'color','y','
                Fontsize',8);
        end

        hold off
        title('Map');
        xlabel('x');
        ylabel('y');

        % make crowded_snapshot?
        if slm_make_crowded_map_snapshot == 1
            fprintf('make snapshot\n');
            fig_map_scl = 1.5;
            cur_fig_style_and_export(size(map,2)*
               fig_map_scl,size(map,1)*fig_map_scl,
               slm_make_crowded_map_snapshot_filename
               ,'')
            slm_enable_plot_sim = 0;
            slm_make_crowded_map_snapshot = 0;
        end

        % Pause execution to get smooth animation
        pause(slm_plot_waiting_time);
    end
end

%% Validation
% check if any packets have disappeared
packets_disappeared_count = size(pck_list,1) - (
   sum(sum(pck_map ~= 0)) + sum(pck_list(:,
   c_pck_delivered_at_timestep) ~= 0) );
if packets_disappeared_count ~= 0; disp('error:
   packets disappeared'); end
```

code/SRCRULE.m

```
% Rules for Sources (Creation)
%  - use src_cur, x, y (all other variables
   accessable as well)
%  - return createPacket: if a packet should be
   created
%  - registers dropped packets in src_cur

createPacket = 0;
dropPacket = 0;
```

```
8
9  switch(src_cur(c_src_rule))
10
11   % RULE 1: create packets at a constant rate
12     case c_src_rule_rate
13     % increment counter by the specified rate
14         src_cur(c_src_rule_rate_counter) = src_cur
               (c_src_rule_rate_counter) + src_cur(
               c_src_rule_rate_rate);
15         if src_cur(c_src_rule_rate_counter) >= 1
16     % ctry to create a packet if the counter is
          high enough
17             src_cur(c_src_rule_rate_counter) =
                  src_cur(c_src_rule_rate_counter) -
                  1; % reset rate counter
18             if pck_map(y,x) == 0
19                 createPacket = 1; % really create
                      the packet (the field is empty)
20             else
21                 dropPacket = 1; % discard packet,
                      because there is no space to
                      create it
22             end
23         end
24
25   % RULE 2: create packets according to specified
       pattern
26   % the pattern is stored in
       src_rule_controlled_controllist(source_id,
       timestep)
27     case c_src_rule_controlled
28         if i <= size(
               src_rule_controlled_controllist,2)
29             src_cur(
                  c_src_rule_controlled_pckcounter) =
                  src_cur(
                  c_src_rule_controlled_pckcounter) +
                  src_rule_controlled_controllist(k,i)
                  ;
30             % check if we have to create a packet
31             if src_cur(
                  c_src_rule_controlled_pckcounter) >=
                   1
32                 src_cur(
                      c_src_rule_controlled_pckcounter
```

```
                    ) = src_cur(
                    c_src_rule_controlled_pckcounter
                    ) - 1;
33
34                  if pck_map(y,x) == 0 % create the
                    packet if there is space
                    otherwise discard it
35                      createPacket = 1;
36                  else
37                      dropPacket = 1;
38                  end
39              end
40
41          end
42  end
43
44  if createPacket == 1
45      src_cur(c_src_packetssent) = src_cur(
          c_src_packetssent) + 1;
46  elseif dropPacket == 1
47      src_cur(c_src_packetsdropped) = src_cur(
          c_src_packetsdropped) + 1;
48  end
```

code/SRCADRRULE.m

```
1  % Rules for Sources (Address)
2  %  - use src_cur, x, y (all other variables
     accessable as well)
3  %  - sets newpacket(c_pck_dest_id) to the
     destination address
4
5  switch src_cur(c_src_adressrule)
6    % assign a random address to each packet
7      case c_src_adressrule_random
8      % create random address
9          adress_domain = snk_list(:,c_snk_id);
10         idx = ceil(rand(1,1)*length(adress_domain)
             );
11     % assign it
12         newpacket(c_pck_dest_id) = adress_domain(
             idx);
13
14   % choose a random address (but the current one)
       and assign the same address to the next few
       packets
```

```
15      case c_src_adressrule_random_batch
16          % generate new address, if needed
17          if src_cur(
               c_src_adressrule_random_batch_counter) >
                src_cur(
               c_src_adressrule_random_batch_countermax
               ) || src_cur(
               c_src_adressrule_random_batch_counter)
               == 0
18              %reset counter
19              src_cur(
                   c_src_adressrule_random_batch_counter
                   ) = src_cur(
                   c_src_adressrule_random_batch_counter
                   ) - src_cur(
                   c_src_adressrule_random_batch_countermax
                   );
20              % generate random adress
21              ok = 0;
22              while ok == 0
23                  adress_domain = snk_list(:,
                       c_snk_id);
24                  if ~exist('mp_src_adrrule_stream',
                       'var') % for backward
                       compatiblity
25                      mp_src_adrrule_stream =
                           RandStream('mrg32k3a','Seed'
                           ,48331);
26                  end
27                  idx = ceil(rand(
                       mp_src_adrrule_stream,1,1)*
                       length(adress_domain));
28                  new_addr = adress_domain(idx);
29                  ok = new_addr ~= src_cur(
                       c_src_adressrule_random_batch_currentaddr
                       );
30              end
31              % set adress
32              src_cur(
                   c_src_adressrule_random_batch_currentaddr
                   ) = new_addr;
33          end
34          src_cur(
               c_src_adressrule_random_batch_counter) =
                src_cur(
```

```
            c_src_adressrule_random_batch_counter) +
             1;
        % assign address
    newpacket(c_pck_dest_id) = src_cur(
       c_src_adressrule_random_batch_currentaddr);

    otherwise
        disp('unknown adressing rule!');
end
```

code/JNCRULE.m

```
% Rules for Converging Junction
%  - use jnc_cur, x, y (all other variables
   accessable as well)
%  - return in_dir: Direction from which paket
   should be taken in
%  output direction of junction(map(y,x)) not
   allowed as in_dir
%  every rule itself must assure that this hold

% helping vectors for easier rule programming
dirs = [c_up c_dn c_lt c_rt];
dirvector = [ [0 -1] ; [0 1]; [-1 0]; [1 0]];

switch jnc_cur(c_jnc_rule)
    case c_jnc_rule_Random
        % RULE - Random
    %  Chooses randomy one possible direction
        in_dir = dirs(ceil(rand(1)*4));
        while in_dir == map(y,x)
            in_dir = ceil(rand(1)*4);
        end
    case c_jnc_rule_TakeIfAnyRandom
        % RULE - Take if any, random
    possible_dirs = [];
    % get all positions where packets are
        tmp = pck_map ~= 0 & map ~= c_em;
        for k = 1:4;
            if(dirs(k) ~= map(y,x) && ...
                    tmp(y+dirvector(k,2),x+
                       dirvector(k,1)))
                possible_dirs = [possible_dirs
                   dirs(k)];
            end
        end
```

```
30      % choose one randomy
31          if(~isempty(possible_dirs))
32              in_dir = possible_dirs(ceil(rand(1)*
                    length(possible_dirs)));
33          end
34      case c_jnc_rule_TakeIfAnyPriorized
35          % RULE - Take if any, strongly priorized
                by direction
36      % get throught all possible direction (due
            priorization)
37          tmp = pck_map ~= 0 & map ~= c_em;
38          for k = jnc_cur([c_jnc_p1 c_jnc_p2
                c_jnc_p3 c_jnc_p4]);
39        % if you find a packet, return according
             direction (takes first found)
40              if(dirs(k) ~= map(y,x) && ...
41                      tmp(y+dirvector(k,2),x+
                            dirvector(k,1)))
42                  in_dir = dirs(k);
43                  break;
44              end
45          end
46      otherwise
47          fprintf('Unknown Rule for Converging
                Junction');
48          return;
49  end
```

code/JNDRULE.m

```
1  % Rules for Diverging Junction
2  %  - use jnd_cur, x, y (all other variables
       accessable as well)
3  %  - return out_dir: Direction to which paket
       should be pushed
4  %  input direction of junction(jnd_cur(c_jnd_indir
       )) not allowed as out_dir
5  %  every rule itself must assure that this hold
6
7  % helping vectors for easier rule programming
8  dirs = [c_up c_dn c_lt c_rt];
9  dirvector = [ [0 -1] ; [0 1]; [-1 0]; [1 0]];
10
11 switch jnd_cur(c_jnd_rule)
12      case c_jnd_rule_Random
13          % RULE - Random
```

```
14      % Chooses randomly one direction
15          out_dir = dirs(ceil(rand(1)*4));
16          while out_dir == jnd_cur(c_jnd_indir)
17             out_dir = dirs(ceil(rand(1)*4));
18          end
19      case c_jnd_rule_SearchFreeWay
20          % RULE - Search free way
21      % get through all directions
22          tmp = pck_map == 0 & map ~= c_em;
23          for k = 1:4
24              if(dirs(k) ~= jnd_cur(c_jnd_indir) &&
                   ...
25                        tmp(y+dirvector(k,2),x+
                             dirvector(k,1)))
26          % if a free way is found, take it
27                   out_dir = dirs(k);
28                   break;
29              end
30          end
31      case c_jnd_rule_SearchFreeWayPriorized
32          % RULE - Search free way, priorized by
                direction
33      % get through all directions (due priorization
            )
34          tmp = pck_map == 0 & map ~= c_em;
35          for k = jnd_cur([c_jnd_p1 c_jnd_p2
                c_jnd_p3 c_jnd_p4]);
36              if(dirs(k) ~= jnd_cur(c_jnd_indir) &&
                   ...
37                        tmp(y+dirvector(k,2),x+
                             dirvector(k,1)))
38          % if a free way is found, take it
39                   out_dir = dirs(k);
40                   break;
41              end
42          end
43      case c_jnd_rule_TakewayAddrMap
44          % RULE - Take way, due address map
45          pck_id = pck_map(y,x);
46          if(pck_id ~= 0)
47        %if a packet is in junction
48        %lookup its direction according its
              destination
49              out_dir = jnd_addr_list(jnd_cur(
                    c_jnd_id),pck_list(pck_id,
```

```
                c_pck_dest_id));
        end
    otherwise
        fprintf('Unknown Rule for Diverging
           Junction');
        return;
end
```

code/SNKRULE.m

```
% Rules for Sinks
%  - use snk_cur, pck_cur, x, y (all other
   variables accessable as well)
%  - clears packet in pck_map and updates
   accounting in pck_cur



%c_snk_rule_constant, c_snk_rule_batch


switch(snk_cur(c_snk_rule))

  % RULE 1: constant sink rate
    case c_snk_rule_constant
    % increment counter by the rate value
        snk_cur(c_snk_rule_constant_counter) =
           snk_cur(c_snk_rule_constant_counter) +
           snk_cur(c_snk_rule_constant_rate);
        if snk_cur(c_snk_rule_constant_counter) >=
            1
      % if the counter is high enough, a packet is
          absorbed
            pck_cur(c_pck_delivered_at_sinkid) =
               snk_cur(c_snk_id);
            pck_cur(c_pck_delivered_at_timestep) =
                i;
            pck_map(pck_cur_coordy,pck_cur_coordx)
                = 0;
            snk_cur(c_snk_rule_constant_counter) =
                snk_cur(c_snk_rule_constant_counter
               ) - 1;
        end

  % RULE 2: all packets received immediately
    case c_snk_rule_immediately
```

```
26      % absorb any packet immediatly
27          pck_cur(c_pck_delivered_at_sinkid) =
               snk_cur(c_snk_id);
28          pck_cur(c_pck_delivered_at_timestep) = i;
29          pck_map(pck_cur_coordy,pck_cur_coordx) =
               0;
30      otherwise
31      %c_snk_rule_batch
32          disp('error: unknown sink rule');
33  end
```

code/cur_fig_style_and_export.m

```
1  function cur_fig_style_and_export(width,height,
       name,root)
2  % Formats and saves plots automatically
3
4    set(gcf,'units','centimeters',...
5          'PaperPosition',[0 0 width height],...
6          'PaperSize', [width height]);
7
8    fontsize = 14;
9    set(findall(gcf,'type','text'),'fontname','Arial
       ');
10   set(findall(gcf,'type','axes'),'fontname','Arial
       ');
11   set(findall(gcf,'type','text'),'fontsize',
       fontsize);
12   set(findall(gcf,'type','axes'),'fontsize',
       fontsize);
13
14     print(gcf,'-dpdf',strcat(root,name,'.pdf'))
15     saveas(gcf,strcat(root,name,'.fig'),'fig')
16 end
```

6.2 maps

code/MAP_Buffer.m

```
1  % Buffer Map
2  % This script generates a Buffer
3  % If it is called in testing environment, all
       buffer parameters must be
4  %  set externally, otherwise the buffer can be
       manually configurated
5
```

```
6  fprintf('Generate Buffer\n');
7
8  %% Manual Configuration
9  if ~exist('testing_in_progress','var')
10     % Buffer geometry
11     mp_bufferlen = 10;     % Main buffer length  0-
          n
12     mp_inbufferlen = 4;    % Input buffer length 0-
          n
13     mp_outbufferlen = 6;   % Output buffer length
          0-n
14     mp_bufferdepths = [1 2 4 8]; % buffer depths
            may be empty
15     mp_topbuffer = false;
16
17     % Diverging Junction rules
18     mp_jnd_std_rule =
          c_jnd_rule_SearchFreeWayPriorized;
19     mp_jnd_std_rule_priorization = [c_rt c_dn c_dn
           c_dn];
20     mp_jnc_std_rule =
          c_jnc_rule_TakeIfAnyPriorized;
21     mp_juc_std_rule_priorization = [c_dn c_lt c_lt
           c_lt];
22
23     % Converging Junction rules
24     mp_jnd_topbuffer_rule =
          c_jnd_rule_SearchFreeWayPriorized;
25     mp_jnd_topbuffer_rule_priorization = [c_rt
          c_up c_dn c_dn];
26     mp_jnc_topbuffer_rule =
          c_jnc_rule_TakeIfAnyPriorized;
27     mp_juc_topbuffer_rule_priorization =  [c_up
          c_dn c_lt c_lt];
28 end
29
30 %% Generate Map
31 % Top line
32 map = zeros(2, 5 + mp_bufferlen + mp_inbufferlen +
       mp_outbufferlen);
33
34 % Depthcounter (for orientation)
35 mp_depthcount = 2;
36
37 % has buffer a top buffer? if yes, append it
```

```
38 if mp_topbuffer
39     map = [ map(1,:); ...
40             [0 zeros(1,mp_inbufferlen) c_rt ones
                  (1,mp_bufferlen)*c_rt c_rt c_dn
                  zeros(1,mp_outbufferlen) 0]; ...
41             [0 zeros(1,mp_inbufferlen) c_up zeros
                  (1,mp_bufferlen)        0 c_dn zeros
                  (1,mp_outbufferlen) 0]; ...
42             map(2,:)];
43     mp_depthcount = mp_depthcount + 2;
44 end
45
46 % buffer main pass line
47 map(mp_depthcount,:) = c_rt;
48 map(mp_depthcount,end) = c_sn;
49
50 % define sinks
51 snk_list(1,:) = zeros(1,c_snk_numcols);
52 snk_list(1,c_snk_id) = 1;
53 snk_list(1,c_src_coordx) = size(map,2);
54 snk_list(1,c_src_coordy) = mp_depthcount;
55 snk_list(1,c_snk_rule) = c_snk_rule_constant;
56 snk_list(1,c_snk_rule_constant_rate) = 0.2;
57
58 % define sources
59 src_list(1,:) = zeros(1,c_src_numcols);
60 src_list(1,c_src_id) = 1;
61 src_list(1,c_src_coordx) = 1;
62 src_list(1,c_src_coordy) = mp_depthcount;
63 src_list(1,c_src_rule) = c_src_rule_rate;
64 src_list(1,c_src_rule_rate_rate) = 0.3;
65
66 % buffer lines, predefinition
67 mp_row = [0 zeros(1,mp_inbufferlen) c_dn zeros(1,
      mp_bufferlen)        0 c_up zeros(1,
      mp_outbufferlen) 0];
68 mp_con = [0 zeros(1,mp_inbufferlen) c_rt ones(1,
      mp_bufferlen)*c_rt c_rt c_up zeros(1,
      mp_outbufferlen) 0];
69
70 mp_jnd_cnt = 1;
71 mp_jnc_cnt = 1;
72 for d = mp_bufferdepths
73     % merge bufferlines
74     map = [map; repmat(mp_row,d,1)]; %#ok<*AGROW>
```

```
75    map = [map; mp_con];
76
77    % configurate diverging junctions
78    jnd_list(mp_jnd_cnt,:) = zeros(1,c_jnd_numcols)
         ; %#ok<*SAGROW>
79    jnd_list(mp_jnd_cnt,c_jnd_id) = mp_jnd_cnt;
80    jnd_list(mp_jnd_cnt,c_jnd_coordx) = 2 +
         mp_inbufferlen;
81    jnd_list(mp_jnd_cnt,c_jnd_coordy) =
         mp_depthcount;
82    jnd_list(mp_jnd_cnt,c_jnd_indir) = c_up; %
         Direction where packetes are not allowed to
         go
83    jnd_list(mp_jnd_cnt,c_jnd_rule) =
         mp_jnd_std_rule;
84    jnd_list(mp_jnd_cnt,[c_jnd_p1 c_jnd_p2 c_jnd_p3
          c_jnd_p4]) = mp_jnd_std_rule_priorization;
85    mp_jnd_cnt = mp_jnd_cnt + 1;
86
87    % configurate converging junctions
88    jnc_list(mp_jnc_cnt,:) = zeros(1,c_jnc_numcols)
         ;
89    jnc_list(mp_jnc_cnt,c_jnc_id) = mp_jnc_cnt;
90    jnc_list(mp_jnc_cnt,c_jnc_coordx) = 4 +
         mp_bufferlen + mp_inbufferlen;
91    jnc_list(mp_jnc_cnt,c_jnc_coordy) =
         mp_depthcount;
92    jnc_list(mp_jnc_cnt,c_jnc_rule) =
         mp_jnc_std_rule;
93    jnc_list(mp_jnc_cnt,[c_jnc_p1 c_jnc_p2 c_jnc_p3
          c_jnc_p4]) = mp_juc_std_rule_priorization;
94    % set wait fields for junctions
95    map(jnc_list(mp_jnc_cnt,c_jnc_coordy), jnc_list
         (mp_jnc_cnt,c_jnc_coordx)-1) = c_wt;
96    map(jnc_list(mp_jnc_cnt,c_jnc_coordy)+1,
         jnc_list(mp_jnc_cnt,c_jnc_coordx)) = c_wt;
97    mp_jnc_cnt = mp_jnc_cnt + 1;
98
99    mp_depthcount = mp_depthcount + d + 1;
100 end
101 % Correct first JNC
102 if(~isempty(mp_bufferdepths))
103     jnd_list(1,c_jnd_indir) = c_lt;
104 end
105 % Reconfigurate first JNC&JND if topbuffer enabled
```

```
106 if mp_topbuffer
107     jnd_list(1,:) = zeros(1,c_jnd_numcols); %#ok<*
        SAGROW>
108     jnd_list(1,c_jnd_id) = 1;
109     jnd_list(1,c_jnd_coordx) = 2 + mp_inbufferlen;
110     jnd_list(1,c_jnd_coordy) = 4;
111     jnd_list(1,c_jnd_indir) = c_lt; % Direction
        where packetes are not allowed to go
112     jnd_list(1,c_jnd_rule) = mp_jnd_topbuffer_rule;
113     jnd_list(1,[c_jnd_p1 c_jnd_p2 c_jnd_p3 c_jnd_p4
        ]) = mp_jnd_topbuffer_rule_priorization;
114
115     jnc_list(1,:) = zeros(1,c_jnc_numcols);
116     jnc_list(1,c_jnc_id) = 1;
117     jnc_list(1,c_jnc_coordx) = 4 + mp_bufferlen +
        mp_inbufferlen;
118     jnc_list(1,c_jnc_coordy) = 4;
119     jnc_list(1,c_jnc_rule) = mp_jnc_topbuffer_rule;
120     jnc_list(1,[c_jnc_p1 c_jnc_p2 c_jnc_p3 c_jnc_p4
        ]) = mp_juc_topbuffer_rule_priorization;
121     map(jnc_list(1,c_jnc_coordy), jnc_list(1,
        c_jnc_coordx)-1) = c_wt;
122     map(jnc_list(1,c_jnc_coordy)-1, jnc_list(1,
        c_jnc_coordx)) = c_wt;
123 end
124
125 % add last line
126 map = [map; zeros(1, 5 + mp_bufferlen +
    mp_inbufferlen + mp_outbufferlen)];
```

code/MAP_LDS.m

```
1  % Linear Distribution System (LDS)
2  % This script generates a LDS
3  % If it is called in testing environment, all LDS
     parameters must be
4  %  set externally, otherwise the LDS can be
     manually configurated
5
6  fprintf('Generate LTS map\n');
7
8  %% Manual Configuration
9  if ~exist('testing_in_progress','var')
10     % geometry
11     mp_src_adrrule_stream = RandStream('mrg32k3a',
          'Seed',48331);
```

```
12      mp_buffersize = 10;
13      mp_batchsize = 10;
14      mp_sinkrate = 0.3;
15      mp_sourcerate = 0.3;
16  end
17
18  if ~exist('coords_manually','var') ||
        coords_manually==0
19      mp_horizontal_coords = [2 5 8 11];
20  end
21
22  %% Generate Map
23
24  % Auswertungsideen:
25  % - dimensionierung buffer-länge, so dass 95%
        ankommen
26  %   assumptions: sink rate infinite, never two of
        the same address batch,
27  %   fixed sink rate
28  %   variation: buffer-länge v. batch size, s.t.
        failure < 95%
29
30  mp_vertical_length = [1 1 1 1]*mp_buffersize;
31
32  mp_map_width = max(mp_horizontal_coords) + 4;
33  mp_map_height = 2 + 1 + 1 + max(mp_vertical_length
        );
34
35  % init
36  map = zeros(mp_map_height,mp_map_width);
37  snk_list = [];
38  jnd_addr_list = [];
39
40  % create source
41  src_list(1,:) = zeros(1,c_src_numcols);
42  src_list(1,c_src_id) = 1;
43  src_list(1,c_src_coordx) = 2;
44  src_list(1,c_src_coordy) = 2;
45  src_list(1,c_src_adressrule) =
        c_src_adressrule_random_batch;
46  src_list(1,
        c_src_adressrule_random_batch_countermax) =
        mp_batchsize;
47  src_list(1,c_src_rule) = c_src_rule_rate;
48  src_list(1,c_src_rule_rate_rate) = mp_sourcerate;
```

```
49
50 % draw horizontal transport line
51 map(2,:) = [0 ones(1,mp_map_width - 3)*c_rt 0 0];
52
53 % draw vertical transport lines
54 for k = 1:length(mp_vertical_length)
55     x = 1+1+mp_horizontal_coords(k);
56     y_offset = 2;
57
58     % create junction
59     jnd_list(k,:) = zeros(1,c_jnd_numcols); %#ok<*
          SAGROW>
60     jnd_list(k,c_jnd_id) = k;
61     jnd_list(k,c_jnd_coordx) = x;
62     jnd_list(k,c_jnd_coordy) = 2;
63     jnd_list(k,c_jnd_indir) = c_lt;
64     jnd_list(k,c_jnd_rule) =
          c_jnd_rule_TakewayAddrMap;
65     jnd_addr_list(k,:) = ones(1,length(
          mp_vertical_length))*c_rt;
66     jnd_addr_list(k,k) = c_dn;
67
68     % create vertical path
69     ys = y_offset + (1:(mp_vertical_length(k)-1));
70     map(ys,x) = c_dn;
71
72     % create special map points
73     y = mp_vertical_length(k) + y_offset;
74     map(y,x) = c_rt;
75     map(y,x+1) = c_sn;
76
77     % create sink
78     snk_list_item = zeros(1,c_snk_numcols);
79     snk_list_item(c_snk_id) = k;
80     snk_list_item(c_snk_coordx) = x+1;
81     snk_list_item(c_snk_coordy) = y;
82     snk_list_item(c_snk_rule) =
          c_snk_rule_constant;
83     snk_list_item(c_snk_rule_constant_rate) =
          mp_sinkrate;
84     snk_list = [snk_list ; snk_list_item]; %#ok<
          AGROW>
85 end
```

code/MAP_CDS.m

```
%% Cirular Distribution System
% Map only to illustrate outlook

map = [...
 c_em c_dn c_em c_em c_em c_em c_em c_em c_em c_em
     c_em; ...
 c_em c_wt c_em c_em c_em c_em c_em c_em c_em c_em
     c_em; ...
 c_em c_dn c_wt c_lt c_lt c_lt c_lt c_lt c_lt c_em
     c_em; ...
 c_em c_dn c_em c_em c_em c_em c_em c_em c_up c_rt
     c_sn; ...
 c_em c_dn c_em c_em c_em c_em c_em c_em c_up c_em
     c_em; ...
 c_em c_rt c_rt c_rt c_rt c_rt c_rt c_rt c_up c_em
     c_em; ...
 c_em c_dn c_em c_dn c_em c_dn c_em c_dn c_em c_em
     c_em; ...
 c_em c_sn c_em c_sn c_em c_sn c_em c_sn c_em c_em
     c_em; ...
];


% define sinks
snk_list(1,:) = zeros(1,c_snk_numcols);
snk_list(1,c_snk_id) = 1;
snk_list(1,c_src_coordx) = 2;
snk_list(1,c_src_coordy) = 8;
snk_list(1,c_snk_rule) = c_snk_rule_constant;
snk_list(1,c_snk_rule_constant_rate) = 0.2;
snk_list(2,:) = zeros(1,c_snk_numcols);
snk_list(2,c_snk_id) = 2;
snk_list(2,c_src_coordx) = 4;
snk_list(2,c_src_coordy) = 8;
snk_list(2,c_snk_rule) = c_snk_rule_constant;
snk_list(2,c_snk_rule_constant_rate) = 0.2;
snk_list(3,:) = zeros(1,c_snk_numcols);
snk_list(3,c_snk_id) = 3;
snk_list(3,c_src_coordx) = 6;
snk_list(3,c_src_coordy) = 8;
snk_list(3,c_snk_rule) = c_snk_rule_constant;
snk_list(3,c_snk_rule_constant_rate) = 0.2;
snk_list(4,:) = zeros(1,c_snk_numcols);
snk_list(4,c_snk_id) = 4;
snk_list(4,c_src_coordx) = 8;
```

```
snk_list(4,c_src_coordy) = 8;
snk_list(4,c_snk_rule) = c_snk_rule_constant;
snk_list(4,c_snk_rule_constant_rate) = 0.2;
snk_list(5,:) = zeros(1,c_snk_numcols);
snk_list(5,c_snk_id) = 5;
snk_list(5,c_src_coordx) = 11;
snk_list(5,c_src_coordy) = 4;
snk_list(5,c_snk_rule) = c_snk_rule_constant;
snk_list(5,c_snk_rule_constant_rate) = 0.2;

% define sources
src_list(1,:) = zeros(1,c_src_numcols);
src_list(1,c_src_id) = 1;
src_list(1,c_src_coordx) = 2;
src_list(1,c_src_coordy) = 1;
src_list(1,c_src_adressrule) =
   c_src_adressrule_random;
src_list(1,c_src_rule) = c_src_rule_rate;
src_list(1,c_src_rule_rate_rate) = 0.3;

% define JNC
jnc_list(1,:) = zeros(1,c_jnc_numcols);
jnc_list(1,c_jnc_id) = 1;
jnc_list(1,c_jnc_coordx) = 2;
jnc_list(1,c_jnc_coordy) = 3;
jnc_list(1,c_jnc_rule) =
   c_jnc_rule_TakeIfAnyPriorized;
jnc_list(1,[c_jnc_p1 c_jnc_p2 c_jnc_p3 c_jnc_p4])
   = [c_up c_rt c_rt c_rt];

% define JND
jnd_list(1,:) = zeros(1,c_jnd_numcols); %#ok<*
   SAGROW>
jnd_list(1,c_jnd_id) = 1;
jnd_list(1,c_jnd_coordx) = 2;
jnd_list(1,c_jnd_coordy) = 6;
jnd_list(1,c_jnd_indir) = c_up; % Direction where
   packetes are not allowed to go
jnd_list(1,c_jnd_rule) = c_jnd_rule_TakewayAddrMap
   ;
jnd_addr_list(1,[1 2 3 4 5]) = [c_dn c_rt c_rt
   c_rt c_rt];

jnd_list(2,:) = zeros(1,c_jnd_numcols); %#ok<*
   SAGROW>
```

```
75 jnd_list(2,c_jnd_id) = 2;
76 jnd_list(2,c_jnd_coordx) = 4;
77 jnd_list(2,c_jnd_coordy) = 6;
78 jnd_list(2,c_jnd_indir) = c_lt; % Direction where
      packetes are not allowed to go
79 jnd_list(2,c_jnd_rule) = c_jnd_rule_TakewayAddrMap
      ;
80 jnd_addr_list(2,[1 2 3 4 5]) = [c_rt c_dn c_rt
      c_rt c_rt];
81
82 jnd_list(3,:) = zeros(1,c_jnd_numcols); %#ok<*
      SAGROW>
83 jnd_list(3,c_jnd_id) = 3;
84 jnd_list(3,c_jnd_coordx) = 6;
85 jnd_list(3,c_jnd_coordy) = 6;
86 jnd_list(3,c_jnd_indir) = c_lt; % Direction where
      packetes are not allowed to go
87 jnd_list(3,c_jnd_rule) = c_jnd_rule_TakewayAddrMap
      ;
88 jnd_addr_list(3,[1 2 3 4 5]) = [c_rt c_rt c_dn
      c_rt c_rt];
89
90 jnd_list(4,:) = zeros(1,c_jnd_numcols); %#ok<*
      SAGROW>
91 jnd_list(4,c_jnd_id) = 4;
92 jnd_list(4,c_jnd_coordx) = 8;
93 jnd_list(4,c_jnd_coordy) = 6;
94 jnd_list(4,c_jnd_indir) = c_lt; % Direction where
      packetes are not allowed to go
95 jnd_list(4,c_jnd_rule) = c_jnd_rule_TakewayAddrMap
      ;
96 jnd_addr_list(4,[1 2 3 4 5]) = [c_rt c_rt c_rt
      c_dn c_rt];
97
98 jnd_list(5,:) = zeros(1,c_jnd_numcols); %#ok<*
      SAGROW>
99 jnd_list(5,c_jnd_id) = 5;
100 jnd_list(5,c_jnd_coordx) = 9;
101 jnd_list(5,c_jnd_coordy) = 4;
102 jnd_list(5,c_jnd_indir) = c_lt; % Direction where
      packetes are not allowed to go
103 jnd_list(5,c_jnd_rule) = c_jnd_rule_TakewayAddrMap
      ;
104 jnd_addr_list(5,[1 2 3 4 5]) = [c_rt c_rt c_rt
      c_rt c_up];
```

6.3 tests

code/Test_LTS.m

```
% Test: Linear Transportation System

clc;
clear
close all;

addpath(strcat(pwd,'\bin'))

run bin/initialize_simulation

testing_in_progress = true;

% Figure export directory
fig_root = 'Test_LTS/';

% Set Simulation Parameters
slm_enable_plot_sim = 0;
slm_num_steps = 200;
slm_plot_waiting_time = 0.05;

% Make map snapshot
slm_make_crowded_map_snapshot = 1;
slm_make_crowded_map_snapshot_rel_occupation = -1; % empty
slm_make_crowded_map_snapshot_filename = strcat('../',fig_root,'LTS_map');

%% Create simple linear buffer as Transportation System

mp_inbufferlen = 0;
mp_outbufferlen = 0;
mp_bufferlen = 0; % variate
mp_bufferdepths = [];
mp_topbuffer = false;

% dont have to set rules

run maps/MAP_Buffer;

%% Perfom Test
% Test different sink and source rates
```

```
40
41 slm_sc = 0; % Simulation Counter
42 test_results = [];
43
44 sink_rates = 0:0.2/4:1;
45 src_rates = 0:0.1/4:0.6;
46
47 slm_sc_tot = length(sink_rates)*length(src_rates);
48 for sink_rate = sink_rates
49     for src_rate = src_rates
50
51         slm_sc = slm_sc + 1;
52         fprintf('Run simulation %i/%i with
              sink_rate:%3.2f and src_rate:%3.2f\n',
              slm_sc,slm_sc_tot,sink_rate,src_rate);
53
54         % Set sink/source rates
55         snk_list(1,c_snk_rule_constant_rate) =
              sink_rate;
56         src_list(1,c_src_rule_rate_rate) =
              src_rate;
57
58         % Simulate
59         run bin/run_simulation
60
61         % get test results
62         packets_arrived = pck_list(pck_list(:,
              c_pck_delivered_at_timestep)~=0);
63         packet_traveltimes = pck_list(
              packets_arrived,
              c_pck_delivered_at_timestep)-pck_list(
              packets_arrived,
              c_pck_started_at_timestep);
64         packet_traveltime_mean = mean(
              packet_traveltimes);
65         buffer_occupation_mean = mean(sum(sum(
              pck_map_history ~= 0,1),2));
66         packet_drop = src_list(1,
              c_src_packetsdropped);
67
68         test_results = [test_results;...
69             length(packets_arrived)
                 packet_traveltime_mean
                 buffer_occupation_mean packet_drop];
70     end
```

```
end
fprintf('all %i simulations completed\n',slm_sc);
save(strcat(fig_root,'test_results.mat'),'
   test_results');

%% Visualize Results
[X,Y] = meshgrid(src_rates,sink_rates);
xmin = min(min(X));
xmax = max(max(X));
ymin = min(min(Y));
ymax = max(max(Y));

% Export Settings
fig_width = 18;
fig_height = 13;

invalid_networks = test_results(:,4) ~= 0;
path_size = sum(sum(map ~= 0,1),2);

% Packet Drops
figure(1)
surf(X,Y,double(vec2mat(invalid_networks,length(
   src_rates))));
hold on
plot3(xmin:0.01:xmax,(xmin:0.01:xmax),ones(1,
   length(xmin:0.01:xmax))*10,'y--','LineWidth',3)
plot3(xmin:0.01:xmax,(xmin:0.01:xmax).*2,ones(1,
   length(xmin:0.01:xmax))*10,'g-','LineWidth',3)
legend('save networks','assumed limitation by sink
   ', 'border of valid region',2);
hold off
colormap([0  0  1; 1  0  0])
view(0,90);
xlim([xmin xmax]);
ylim([ymin ymax]);
xlabel('source rate')
ylabel('sink rate')
title('Packet Drops Occurrence')
cur_fig_style_and_export(18,15,'LTS_drops',
   fig_root);

% Clear all invalid networks
test_results_clear = test_results;
test_results_clear(invalid_networks,:) = 0;

```

```
110 % Troughput
111 figure(2)
112 surf(X,Y,vec2mat(test_results_clear(:,1),length(
      src_rates)) / slm_num_steps);
113 view(0,90);
114 colorbar;
115 xlim([xmin xmax]);
116 ylim([ymin ymax]);
117 xlabel('source rate')
118 ylabel('sink rate')
119 title('Throughput per Simulation Step')
120 fprintf('Maximal Troughput: %3.3f\n',max(
      test_results_clear(:,1))/ slm_num_steps)
121 cur_fig_style_and_export(fig_width,fig_height,'
      LTS_troughput',fig_root);
122
123 % Path Occupation
124 figure(3)
125 surf(X,Y,vec2mat(test_results_clear(:,3),length(
      src_rates)) / path_size);
126 view(0,90);
127 colorbar;
128 xlim([xmin xmax]);
129 ylim([ymin ymax]);
130 xlabel('source rate')
131 ylabel('sink rate')
132 title('Mean Relative Path Occupation')
133 cur_fig_style_and_export(fig_width,fig_height,'
      LTS_mean_path_occupation',fig_root);
134
135 % Travel Time
136 figure(4)
137 surf(X,Y,vec2mat(test_results_clear(:,2),length(
      src_rates))/ path_size);
138 view(0,90);
139 colorbar;
140 xlim([xmin xmax]);
141 ylim([ymin ymax]);
142 xlabel('source Rate')
143 ylabel('sink Rate')
144 title('Mean Travel Time (# of Simulation Steps /
      Path Length)')
145 cur_fig_style_and_export(fig_width,fig_height,'
      LTS_mean_travel_time',fig_root);
146
```

```
147 %% Test - Path Occupation over time - Max
    troughput
148 % In maximal Troughput case:
149 sink_rate = 1;
150 src_rate = 0.5;
151
152 fprintf('Run simulation with sink_rate:%3.2f and
    src_rate:%3.2f\n',sink_rate,src_rate);
153
154 % Set sink/source rates
155 snk_list(1,c_snk_rule_constant_rate) = sink_rate;
156 src_list(1,c_src_rule_rate_rate) = src_rate;
157
158 % Simulate
159 run bin/run_simulation
160
161 fprintf('Simulation Completed\n');
162
163 % Plot occupation versus time
164 figure(5)
165 path_occupation = squeeze(sum(sum(pck_map_history
    ~= 0,1),2)) ./ path_size;
166 plot(1:slm_num_steps+1,path_occupation,'b','
    LineWidth',2);
167 hold on
168 plot(1:slm_num_steps+1,ones(1,slm_num_steps+1)*
    mean(path_occupation),'r--','LineWidth',2);
169 hold off
170 legend('occupation','mean')
171 xlim([1 slm_num_steps+1])
172 ylim([0 1]);
173 xlabel('time (steps)')
174 ylabel('mean relative path occupation');
175 title('Path Occupation');
176 fprintf('Mean Path Occupation: %3.3f\n',mean(
    path_occupation))
177 fprintf('# Packet dropped: %i\n',src_list(1,
    c_src_packetsdropped))
178 cur_fig_style_and_export(17,10,'
    LTS_occupation_over_time_max_troughput',fig_root
    );
179
180 %% Test - Path Occupation over time - Slow filling
     occupation
181 % Slow filling occupation:
```

```
182 sink_rate = 0.9;
183 src_rate = 0.475;
184
185 fprintf('Run simulation with sink_rate:%3.2f and
    src_rate:%3.2f\n',sink_rate,src_rate);
186
187 % Set sink/source rates
188 snk_list(1,c_snk_rule_constant_rate) = sink_rate;
189 src_list(1,c_src_rule_rate_rate) = src_rate;
190
191 % Simulate
192 run bin/run_simulation
193
194 fprintf('Simulation Completed\n');
195
196 % Plot occupation versus time
197 figure(6)
198 path_occupation = squeeze(sum(sum(pck_map_history
    ~= 0,1),2)) ./ path_size;
199 plot(1:slm_num_steps+1,path_occupation,'b','
    LineWidth',2);
200 hold on
201 plot(1:slm_num_steps+1,ones(1,slm_num_steps+1)*
    mean(path_occupation),'r--','LineWidth',2);
202 hold off
203 legend('occupation','mean')
204 xlim([1 slm_num_steps+1])
205 ylim([0 1]);
206 xlabel('time (steps)')
207 ylabel('mean relative path occupation');
208 title('Path Occupation');
209 fprintf('Mean Path Occupation: %3.3f\n',mean(
    path_occupation))
210 fprintf('# Packet dropped: %i\n',src_list(1,
    c_src_packetsdropped))
211 cur_fig_style_and_export(17,10,'
    LTS_occupation_over_slow_filling',fig_root);
```

code/Test_Buffer_Linear.m

```
1 % Test: Linear Buffer
2
3 clc;
4 clear
5 close all;
6
```

```
addpath(strcat(pwd,'\bin'))

run bin/initialize_simulation

testing_in_progress = true;

% Figure export directory
fig_root = 'Test_Buffer/';
fig_width = 20;
fig_height = 8;

% Set Simulation Parameters
slm_enable_plot_sim = 0;
slm_num_steps = 960;
slm_plot_waiting_time = 0.05;

%% Create simple linear buffer

mp_inbufferlen = 0;
mp_outbufferlen = 0;
mp_bufferlen = 0; % variate
mp_bufferdepths = [];
mp_topbuffer = false;

% dont have to set rules

run maps/MAP_Buffer;

%% Perfom Test
% Test different sink and source rates
slm_sc = 0; % Simulation Counter
test_results_linear = [];

mp_bufferlens = 0:20;

src_max = 0.4;
src_mean = 0.20;
src_frequency = 0.05; %0.1
snk_rate = 0.5;

%% Generate Packet Releases List
PWM_len = 8;

% Create Input Signal
input_len = slm_num_steps/(2*PWM_len)-1;
```

```
52 input = ((-1*cos((0:input_len)*src_frequency*(2*pi
   )))*(src_max-src_mean)+src_mean);
53
54 % Make PWM
55 PWM = [];
56 for i = input
57     cnt = round(PWM_len*i*2);
58     PWM = [PWM ones(1,cnt) zeros(1,PWM_len-cnt)];
59 end
60
61 % Limit PWM to 0.5
62 PWM = [PWM' zeros(length(PWM),1)]';
63 src_releases_list = PWM(:);
64
65 % Plot
66 stem(src_releases_list,'LineWidth',1)
67 hold on
68 plot(0:2*PWM_len:input_len*2*PWM_len,input,'r','
   LineWidth',2)
69 hold off
70 str = sprintf('Packet Releases - avg:%3.2f max
   :%3.2f eff.avg:%3.2f',mean(input),max(input),sum
   (src_releases_list)./slm_num_steps);
71 title(str)
72 xlabel('time');
73 ylabel('packet release / rate');
74 legend('packet release (if 1)','mean release rate'
   );
75 cur_fig_style_and_export(fig_width,fig_height,'
   Release_Packets',fig_root);
76
77 %% Simulate
78 slm_sc_tot = length(mp_bufferlens);
79 for mp_bufferlen = mp_bufferlens
80         if(mp_bufferlen == 9)
81             % Make map snapshot when bufferlen ==
                9
82             slm_make_crowded_map_snapshot = 1;
83             slm_make_crowded_map_snapshot_rel_occupation
                   = -1; % empty
84             slm_make_crowded_map_snapshot_filename
                   = strcat('../',fig_root,'
                  Linear_Buffer_map');
85         end
86
```

```
87          slm_sc = slm_sc + 1;
88          fprintf('Run simulation %i/%i with
               mp_bufferlen:%3.2f\n',slm_sc,slm_sc_tot,
               mp_bufferlen);
89
90          % Create Map
91          run maps/MAP_Buffer;
92
93          % Set Source/Sink Parameters
94          src_list(1,c_src_rule) =
               c_src_rule_controlled;
95          src_rule_controlled_controllist(1,:) =
               src_releases_list;
96
97          snk_list(1,c_snk_rule_constant_rate) =
               snk_rate;
98
99          % Simulate
100         run bin/run_simulation
101
102         % get test results
103         packets_arrived = pck_list(pck_list(:,
               c_pck_delivered_at_timestep)~=0);
104         packet_traveltimes = pck_list(
               packets_arrived,
               c_pck_delivered_at_timestep)-pck_list(
               packets_arrived,
               c_pck_started_at_timestep);
105         packet_traveltime_mean = mean(
               packet_traveltimes);
106         buffer_occupation_mean = mean(sum(sum(
               pck_map_history ~= 0,1),2))/(
               mp_bufferlen+5);
107         packet_drop = src_list(1,
               c_src_packetsdropped);
108
109         test_results_linear = [test_results_linear
               ;...
110           mp_bufferlen+5 packet_drop
                 packet_traveltime_mean
                 buffer_occupation_mean];
111 end
112 fprintf('all %i simulations completed\n',slm_sc);
113 save(strcat(fig_root,'test_results_linear.mat'),'
    test_results_linear');
```

```
114
115 %% Visualize Results
116 xmin = min(min(test_results_linear(:,1)));
117 xmax = max(max(test_results_linear(:,1)));
118
119 invalid_networks = test_results_linear(:,2) ~= 0;
120
121 % Drops
122 figure(10);
123 bar(test_results_linear(:,1),test_results_linear
      (:,2),'r')
124 xlim([xmin-1 xmax+1]);
125 xlabel('buffer size')
126 ylabel('# packet dropped')
127 title('Packet Drops')
128 cur_fig_style_and_export(fig_width,fig_height,'
      Linear_Buffer_drops',fig_root);
129
130 % Mean Traveltime
131 figure(11);
132 bar(test_results_linear(:,1),test_results_linear
      (:,3),'b')
133 hold on
134 bar(test_results_linear(invalid_networks,1),
      test_results_linear(invalid_networks,3),'r')
135 hold off
136 xlim([xmin-1 xmax+1]);
137 xlabel('buffer size')
138 ylabel('# simulation steps needed')
139 title('Mean Traveltime')
140 cur_fig_style_and_export(fig_width,fig_height,'
      Linear_Buffer_mean_travel_time',fig_root);
141
142 %% Test - Buffer Occupation
143 mp_bufferlens = [5 14 30]-5;%6;14;24
144 snk_rate = 0.5;
145
146 slm_sc = 0;
147 test_results_linear_occup = [];
148
149 slm_sc_tot = length(mp_bufferlens);
150 for mp_bufferlen = mp_bufferlens
151     slm_sc = slm_sc + 1;
152     fprintf('Run simulation %i/%i with
          mp_bufferlen:%3.2f\n',slm_sc,slm_sc_tot,
```

```
        mp_bufferlen);
153
154     % Create Map
155     run maps/MAP_Buffer;
156
157     % Set Source/Sink Parameters
158     src_list(1,c_src_rule) = c_src_rule_controlled
        ;
159     src_rule_controlled_controllist(1,:) =
        src_releases_list;
160
161     snk_list(1,c_snk_rule_constant_rate) =
        snk_rate;
162
163     % Simulate
164     run bin/run_simulation
165
166     path_size = mp_bufferlen+5;
167     path_occupation = squeeze(sum(sum(
        pck_map_history ~= 0,1),2)) ./ path_size;
168
169     test_results_linear_occup = [
        test_results_linear_occup; path_occupation
        '];
170 end
171 fprintf('Simulation Completed\n');
172 save(strcat(fig_root,'test_results_linear_occup.
    mat'),'test_results_linear_occup');
173
174 %% Plot occupation versus time
175 figure(5)
176 plot(1:slm_num_steps+1,test_results_linear_occup,'
    LineWidth',2);
177 hold on
178 plot(1:slm_num_steps+1,repmat(mean(
    test_results_linear_occup')',1,slm_num_steps+1),
    '--','LineWidth',2);
179 hold off
180 legend('Buffersize 5','Buffersize 14','Buffersize
    30')
181 xlim([1 slm_num_steps+1])
182 ylim([0 1]);
183 xlabel('timesteps')
184 ylabel('# packets / buffer size');
185 title('Buffer Occupation');
```

```
186 cur_fig_style_and_export(fig_width,fig_height,'
      Linear_Buffer_occupation_over_time',fig_root);
187
188 fprintf('Mean Buffer Occupation: \n');
189 mean(test_results_linear_occup')
190 fprintf('# Packet dropped: %i\n',src_list(1,
      c_src_packetsdropped))
```

code/Test_Buffer_Double.m

```
1  % Test: Double Buffer
2
3  clc;
4  clear
5  close all;
6
7  addpath(strcat(pwd,'\bin'))
8
9  run bin/initialize_simulation
10
11 testing_in_progress = true;
12
13 % Figure export directory
14 fig_root = 'Test_Buffer/';
15 fig_width = 20;
16 fig_height = 8;
17
18 % Set Simulation Parameters
19 slm_enable_plot_sim = 0;
20 slm_num_steps = 960;
21 slm_plot_waiting_time = 0.05;
22
23 %% Create simple linear buffer
24
25 mp_inbufferlen = 0;
26 mp_outbufferlen = 0;
27 mp_bufferlen = 0; % variate
28 mp_bufferdepths = [1];
29 mp_topbuffer = false;
30
31 mp_jnd_std_rule =
      c_jnd_rule_SearchFreeWayPriorized;
32 mp_jnd_std_rule_priorization = [c_rt c_dn c_dn
      c_dn];
33 mp_jnc_std_rule = c_jnc_rule_TakeIfAnyPriorized;
34 mp_juc_std_rule_priorization = [c_dn c_lt c_lt
```

```
    c_lt];
35
36 mp_jnd_topbuffer_rule =
    c_jnd_rule_SearchFreeWayPriorized;
37 mp_jnd_topbuffer_rule_priorization = [c_rt c_up
    c_dn c_dn];
38 mp_jnc_topbuffer_rule =
    c_jnc_rule_TakeIfAnyPriorized;
39 mp_juc_topbuffer_rule_priorization =  [c_up c_dn
    c_lt c_lt];
40
41 run maps/MAP_Buffer;
42
43 %% Perfom Test
44 % Test different sink and source rates
45 slm_sc = 0; % Simulation Counter
46 test_results_double = [];
47
48 mp_bufferlens = 0:10;
49
50 src_max = 0.4;
51 src_mean = 0.20;
52 src_frequency = 0.05; %0.1
53 snk_rate = 0.5;
54
55 %% Generate Packet Releases List
56 PWM_len = 8;
57
58 input_len = slm_num_steps/(2*PWM_len)-1;
59 input = ((-1*cos((0:input_len)*src_frequency*(2*pi
    )))*(src_max-src_mean)+src_mean);
60
61 PWM = [];
62 for i = input
63     cnt = round(PWM_len*i*2);
64     PWM = [PWM ones(1,cnt) zeros(1,PWM_len-cnt)];
65 end
66
67 PWM = [PWM' zeros(length(PWM),1)]';
68 src_releases_list = PWM(:);
69
70 stem(src_releases_list,'LineWidth',1)
71 hold on
72 plot(0:2*PWM_len:input_len*2*PWM_len,input,'r','
    LineWidth',2)
```

```
hold off
str = sprintf('Packet Releases - avg:%3.2f max
   :%3.2f eff.avg:%3.2f',mean(input),max(input),sum
   (src_releases_list)./slm_num_steps);
title(str)
xlabel('time');
ylabel('packet release (if 1)');
% Plot not saved again

%% Simulate
slm_sc_tot = length(mp_bufferlens);
for mp_bufferlen = mp_bufferlens
        if(mp_bufferlen == 2)
            % Make map snapshot when bufferlen ==
               10
            slm_make_crowded_map_snapshot = 1;
            slm_make_crowded_map_snapshot_rel_occupation
                = -1; % empty
            slm_make_crowded_map_snapshot_filename
                = strcat('../',fig_root,'
               Double_Buffer_map');
        end

        slm_sc = slm_sc + 1;
        fprintf('Run simulation %i/%i with
           mp_bufferlen:%3.2f\n',slm_sc,slm_sc_tot,
           mp_bufferlen);

        % Create Map
        run maps/MAP_Buffer;

        % Set Source/Sink Parameters
        src_list(1,c_src_rule) =
           c_src_rule_controlled;
        src_rule_controlled_controllist(1,:) =
           src_releases_list;

        snk_list(1,c_snk_rule_constant_rate) =
           snk_rate;

        % Simulate
        run bin/run_simulation

        % get test results
        packets_arrived = pck_list(pck_list(:,
```

```
            c_pck_delivered_at_timestep)~=0);
107         packet_traveltimes = pck_list(
               packets_arrived,
               c_pck_delivered_at_timestep)-pck_list(
               packets_arrived,
               c_pck_started_at_timestep);
108         packet_traveltime_mean = mean(
               packet_traveltimes);
109         buffer_occupation_mean = mean(sum(sum(
               pck_map_history ~= 0,1),2))/(
               mp_bufferlen*2+10);
110         packet_drop = src_list(1,
               c_src_packetsdropped);
111
112         test_results_double = [test_results_double
               ;...
113             mp_bufferlen*2+10 packet_drop
                  packet_traveltime_mean
                  buffer_occupation_mean];
114 end
115 fprintf('all %i simulations completed\n',slm_sc);
116 save(strcat(fig_root,'test_results_double.mat'),'
    test_results_double');
117
118 %% Visualize Results
119 xmin = min(min(test_results_double(:,1)));
120 xmax = max(max(test_results_double(:,1)));
121
122 invalid_networks = test_results_double(:,2) ~= 0;
123
124 % Drops
125 figure(10);
126 bar(test_results_double(:,1),test_results_double
    (:,2),'r')
127 xlim([xmin-1 xmax+1]);
128 ylim([0 1]);
129 xlabel('buffer size')
130 ylabel('# packet dropped')
131 title('Packet Drops')
132 cur_fig_style_and_export(fig_width,fig_height,'
    Double_Buffer_drops',fig_root);
133
134 % Comparison with linear Buffer
135 load 'Test_Buffer\test_results_linear.mat'
136 figure(11);
```

```
137 bar(test_results_linear(:,1),test_results_linear
    (:,3),'g')
138 hold on
139 bar(test_results_linear(test_results_linear(:,2)
    ~= 0,1),test_results_linear(test_results_linear
    (:,2) ~= 0,3),'r')
140 bar(test_results_double(:,1),test_results_double
    (:,3),'b')
141 bar(test_results_double(invalid_networks,1),
    test_results_double(invalid_networks,3),'r')
142 hold off
143 xlim([5-1 xmax+1]);
144 xlabel('buffer size')
145 ylabel('# simulation steps needed')
146 title('Mean Traveltime')
147 cur_fig_style_and_export(fig_width,fig_height,'
    Double_Buffer_mean_travel_time_comparison',
    fig_root);
148
149
150 %% Test - Buffer Occupation
151 mp_bufferlen = 2;
152 snk_rate = 0.5;
153
154 fprintf('Run simulation with mp_bufferlen:%3.2f\n'
    ,mp_bufferlen);
155
156 % Create Map
157 run maps/MAP_Buffer;
158
159 % Set Source/Sink Parameters
160 src_list(1,c_src_rule) = c_src_rule_controlled;
161 src_rule_controlled_controllist(1,:) =
    src_releases_list;
162
163 snk_list(1,c_snk_rule_constant_rate) = snk_rate;
164
165 %slm_enable_plot_sim = 1;
166 % Simulate
167 run bin/run_simulation
168
169 path_size = mp_bufferlen*2+10;
170 test_results_double_occup = (squeeze(sum(sum(
    pck_map_history ~= 0,1),2)) ./ path_size)';
171
```

```
fprintf('Simulation Completed\n');

%% Plot occupation versus time
load 'Test_Buffer\test_results_linear_occup.mat'
figure(20)
plot(1:slm_num_steps+1,test_results_linear_occup
   (2,:),'g','LineWidth',2);
hold on
plot(1:slm_num_steps+1,test_results_double_occup,'
   LineWidth',2);
plot(1:slm_num_steps+1,repmat(mean(
   test_results_linear_occup(2,:)),1,slm_num_steps
   +1),'g--','LineWidth',2);
plot(1:slm_num_steps+1,repmat(mean(
   test_results_double_occup),1,slm_num_steps+1),'
   --','LineWidth',2);
hold off
xlim([1 slm_num_steps+1])
ylim([0 1]);
legend('linear Buffer (size 14)','double Buffer (
   size 14)')
xlabel('timesteps')
ylabel('# packets / buffer size');
title('Buffer Occupation');
cur_fig_style_and_export(fig_width,fig_height,'
   Double_Buffer_occupation_comparison',fig_root);

fprintf('Mean Buffer Occupation: %i\n',mean(
   test_results_double_occup'));
fprintf('# Packet dropped: %i\n',src_list(1,
   c_src_packetsdropped))
```

code/Test_LDS_simplest.m

```
clear; clc;

addpath(strcat(pwd,'\bin'))
run bin/initialize_simulation

testing_in_progress = true;
coords_manually = true;

% Set Simulation Parameters
slm_enable_plot_sim = 0;
slm_num_steps = 300;
slm_plot_waiting_time = 0.05;
```

```

mp_horizontal_coords = [2 5 8 11];

mp_sinkrate = 0.15;
mp_sourcerate = 0.3;

mp_most_simple_map = 1;
mp_buffersize = 2;

mp_batchsize = 1;
mp_src_adrrule_stream = RandStream('mrg32k3a','Seed',48921);
run maps/MAP_LDS;
run bin/run_simulation

packetsinsys = squeeze(sum(sum(pck_map_history ~= 0,1),2));
plot(0:size(packetsinsys)-1,packetsinsys/24) % /24 to get mean rel. path occ.

ylabel('mean rel. path occupation')
xlabel('time step')
title('Mean Relative Path Occupation')

cur_fig_style_and_export(17,10,'LDS_simplest','Test_LDS/');
```

code/Test_LDS_Test1.m

```
% reproduzierbarkeit: bei run_simulation map auf map_lts setzen, deactivate
% grpahical output, slm_numsteps = 100

% slm_buffersize, slm_batchsize
% failure rate, mean travel time
clc;
clear
close all;


addpath(strcat(pwd,'\bin'))
run bin/initialize_simulation

testing_in_progress = true;

```

```
% Set Simulation Parameters
slm_enable_plot_sim = 0;
slm_num_steps = 200;
slm_plot_waiting_time = 0.05;

slm_buffersize_max = 15;
slm_batchsize_max = 15;

failurerate = zeros(slm_buffersize_max,
   slm_batchsize_max);

mp_sinkrate = 0.175;
mp_sourcerate = 0.5;
for buffersize = 1:slm_buffersize_max
    mp_buffersize = buffersize;
    for batchsize = 1:slm_batchsize_max
        mp_batchsize = batchsize;
        mp_src_adrrule_stream = RandStream('
           mrg32k3a','Seed',48921);
        run maps/MAP_LDS;
        run bin/run_simulation
        failurerate(buffersize,batchsize) =
           src_list(1,c_src_packetsdropped)/(
           src_list(1,c_src_packetssent) + src_list
           (1,c_src_packetsdropped));
    end
end

%% plot
figure(3)
surf(1:slm_batchsize_max,1:slm_buffersize_max,
   failurerate);
xlim([1 slm_batchsize_max]); ylim([1
   slm_buffersize_max]);
view(0,90);
line([1 9],[1 15],[1 1],'color','g','linewidth',3)
   ;
colorbar;
ylabel('Buffer Size')
xlabel('Batch Size')
title('Failure Rate with Source Rate 0.5, Sink
   Rate 0.175, 200 Steps')

cur_fig_style_and_export(15,12,'LDS_Test1','
   Test_LDS/');
```

code/Test_LDS_Test2.m

```
% reproduzierbarkeit: bei run_simulation map auf map_lts setzen

% slm_buffersize, slm_batchsize
% failure rate, mean travel time
clc;
clear
close all;

addpath(strcat(pwd,'\bin'))
run bin/initialize_simulation

testing_in_progress = true;

%% set Simulation Parameters
slm_enable_plot_sim = 0;
slm_num_steps = 200;
slm_plot_waiting_time = 0.05;

slm_buffersize_max = 15;

failurerate = zeros(slm_buffersize_max,10);

mp_batchsize = 8;
mp_sourcerate = 0.5;
%% run simulations
for buffersize = 1:slm_buffersize_max
    mp_buffersize = buffersize;
    for mp_sinkrate = 0:0.05:0.5
        mp_src_adrrule_stream = RandStream('mrg32k3a','Seed',48921);
        run maps/MAP_LDS;
        run bin/run_simulation
        failurerate(buffersize,round(mp_sinkrate*20+1)) = src_list(1,c_src_packetsdropped)/(src_list(1,c_src_packetssent) + src_list(1,c_src_packetsdropped));
    end
end

%% plot
figure(3)
surf(0:0.05:0.5,1:slm_buffersize_max,failurerate);
```

```
xlim([0 0.5]); ylim([1 slm_buffersize_max]);
view(0,90);
colorbar;
ylabel('Buffer Size')
xlabel('Sink Rate')
title('Failure Rate after 200 steps')

cur_fig_style_and_export(15,12,'LDS_Test2','
    Test_LDS/');
```

code/Test_LDS_simplest_graph1.m

```
%plot rate depending on sinkrate & source rate,
    killinvalid nets
clear; clc;
addpath(strcat(pwd,'\bin'))
run bin/initialize_simulation

testing_in_progress = true;
coords_manually = true;

% Set Simulation Parameters
slm_enable_plot_sim =0;
slm_num_steps = 300;
slm_plot_waiting_time = 0.05;


mp_horizontal_coords = [2 5 8 11];

mp_sinkrate = 0.2;
mp_sourcerate = 0.0;

mp_most_simple_map = 1;
mp_buffersize = 2;

mp_batchsize = 1;
mp_src_adrrule_stream = RandStream('mrg32k3a','
    Seed',48921);


% sinkrates = 0:0.05:0.5;
% sourcerates = 0:0.05:0.5;

sinkrates = 0:0.025:0.25;
sourcerates = 0:0.05:0.5;
for sinkrate_idx = 1:length(sinkrates)
```

```
33      for sourcerate_idx = 1:length(sourcerates)
34          mp_sourcerate = sourcerates(sourcerate_idx
               );
35          mp_sinkrate = sinkrates(sinkrate_idx);
36          run maps/MAP_LDS;
37          run bin/run_simulation
38          throughput(sinkrate_idx,sourcerate_idx) =
               sum(pck_list(:,
               c_pck_delivered_at_timestep)~=0)
39          packet_drops(sinkrate_idx,sourcerate_idx)
               = sum(src_list(:,c_src_packetsdropped))
40          packet_sent(sinkrate_idx,sourcerate_idx) =
                sum(src_list(:,c_src_packetssent))
41      end
42  end
43
44
45  %% ploting
46  figure(3)
47  surf(sourcerates,sinkrates,throughput);
48  %xlim([0 0.5]); ylim([1 slm_buffersize_max]);
49  view(0,90);
50  colorbar;
51  ylabel('sink rate')
52  xlabel('source rate')
53  title('packets delivered after 300 steps')
54  line([0 0.5],[0 0.175],'color','g','linewidth',2);
55
56  cur_fig_style_and_export(15,12,'
       LDS_Simplest_Graph1_plot1','Test_LDS/');
57
58  figure(4)
59  surf(sourcerates,sinkrates,packet_drops./
       packet_sent);
60  %xlim([0 0.5]); ylim([1 slm_buffersize_max]);
61  view(0,90);
62  colorbar;
63  ylabel('sink rate')
64  xlabel('source rate')
65  title('Failure Rate after 300 Steps')
66  line([0.0 0.5],[0 0.175],'color','g','linewidth'
       ,2);
67  xlim([0.05 0.5])
68  cur_fig_style_and_export(15,12,'
       LDS_Simplest_Graph1_plot2','Test_LDS/');
```

code/Test_LDS_simplest_graph2.m

```
%plot rate depending on sinkrate & source rate, killinvalid nets
clear; clc;
addpath(strcat(pwd,'\bin'))
run bin/initialize_simulation

testing_in_progress = true;
coords_manually = true;

% Set Simulation Parameters
slm_enable_plot_sim =0;
slm_num_steps = 300;
slm_plot_waiting_time = 0.05;


mp_horizontal_coords = [2 5 8 11];

mp_sinkrate = 0.2;
mp_sourcerate = 0.0;

mp_most_simple_map = 1;

mp_batchsize = 1;
mp_src_adrrule_stream = RandStream('mrg32k3a','Seed',48921);

mp_sourcerate = 0.5;

sinkrates = 0:0.05:0.5;
bufferlengths = 2:2:30;
for sinkrate_idx = 1:length(sinkrates)
    for bufferlength_idx = 1:length(bufferlengths)
        mp_sinkrate = sinkrates(sinkrate_idx);
        mp_buffersize = bufferlengths(bufferlength_idx);
        run maps/MAP_LDS;
        run bin/run_simulation;
        %throughput(sinkrate_idx,bufferlength_idx) = sum(pck_list(:,c_pck_delivered_at_timestep)~=0);
        packet_drops(sinkrate_idx,bufferlength_idx) = sum(src_list(:,c_src_packetsdropped));
    end
```

```
end

% figure(3)
% surf(bufferlength_idx,sinkrates,throughput);
% %xlim([0 0.5]); ylim([1 slm_buffersize_max]);
% view(0,90);
% colorbar;
% ylabel('sink rate')
% xlabel('source rate')
% title('throughput after 300 steps')


%% plot 2
figure(4)
surf(bufferlengths,sinkrates,packet_drops);
xlim([round(min(bufferlengths)) round(max(
   bufferlengths))]); %ylim([1 slm_buffersize_max])
   ;
view(0,90);
colorbar;
ylabel('sink rate')
xlabel('buffer length')
title('packets dropped after 300 steps')




cur_fig_style_and_export(15,12,'
   LDS_Simplest_Graph2','Test_LDS/');
```

code/Test_LDS_simplest_graph3.m

```
%plot rate depending on sinkrate & source rate,
   killinvalid nets
clear; clc;
addpath(strcat(pwd,'\bin'))
run bin/initialize_simulation

testing_in_progress = true;
coords_manually = true;

% Set Simulation Parameters
slm_enable_plot_sim =0;
slm_num_steps = 300;
slm_plot_waiting_time = 0.05;

```

```
14
15 mp_horizontal_coords = [2 5 8 11];
16
17 mp_sinkrate = 0.2;
18
19 mp_most_simple_map = 1;
20
21 mp_batchsize = 30;          % !!!!!!!!!SET BATCH
   SIZE HERE!!!!!!!!!
22 mp_src_adrrule_stream = RandStream('mrg32k3a','
   Seed',48921);
23
24 mp_sourcerate = 0.5;
25
26 sinkrates = 0:0.05:0.8;
27 bufferlengths = 2:2:30;
28 for sinkrate_idx = 1:length(sinkrates)
29     for bufferlength_idx = 1:length(bufferlengths)
30         mp_sinkrate = sinkrates(sinkrate_idx);
31         mp_buffersize = bufferlengths(
              bufferlength_idx);
32         run maps/MAP_LDS;
33         run bin/run_simulation;
34         %throughput(sinkrate_idx,bufferlength_idx)
               = sum(pck_list(:,
             c_pck_delivered_at_timestep)~=0);
35         packet_drops(sinkrate_idx,bufferlength_idx
             ) = sum(src_list(:,c_src_packetsdropped)
             );
36     end
37 end
38
39 % figure(3)
40 % surf(bufferlength_idx,sinkrates,throughput);
41 % %xlim([0 0.5]); ylim([1 slm_buffersize_max]);
42 % view(0,90);
43 % colorbar;
44 % ylabel('sink rate')
45 % xlabel('source rate')
46 % title('throughput after 300 steps')
47
48
49 %% plot 2
50 figure(4)
51 surf(bufferlengths,sinkrates,packet_drops);
```

```
xlim([round(min(bufferlengths)) round(max(
    bufferlengths))]); %ylim([1 slm_buffersize_max])
    ;
view(0,90);
colorbar;
ylabel('sink rate')
xlabel('buffer length')
title('packets dropped after 300 steps with batch-
    size 30') % !!!!!!!!!SET BATCH SIZE HERE
    !!!!!!!!!!


cur_fig_style_and_export(15,12,'
    LDS_Simplest_Graph3_bs30','Test_LDS/'); %
    !!!!!!!!!SET BATCH SIZE HERE!!!!!!!!!
```

6.4 auxiliary

code/Graphics_for_Doku.m

```
%% Plot Graphics for Script
% This Scripts only creates additional Grapfics
    for the script
% No Simulations mentioned in the script are
    performed here

clc;
clear
close all;

addpath(strcat(pwd,'\bin'))

run bin/initialize_simulation

testing_in_progress = true;

% Figure export directory
fig_root = 'imag/';

% Set Simulation Parameters
slm_enable_plot_sim = 0;
slm_num_steps = 2;
slm_plot_waiting_time = 0.05;

% Make map snapshot
```

```
slm_make_crowded_map_snapshot = 1;
slm_make_crowded_map_snapshot_rel_occupation = -1;
    % empty

%% Path

map = zeros(3);
map(1:3,2) = c_dn;

slm_make_crowded_map_snapshot = 1;
slm_make_crowded_map_snapshot_filename = strcat('
   ../',fig_root,'path');

run bin/run_simulation

%% Error

map = zeros(3);
map(1:3,2) = c_dn;
map(2,2) = c_er;

slm_make_crowded_map_snapshot = 1;
slm_make_crowded_map_snapshot_filename = strcat('
   ../',fig_root,'error');

run bin/run_simulation

%% Source

map = zeros(3);
map(2,2) = c_dn;
map(3,2) = c_dn;

src_list(1,:) = zeros(1,c_src_numcols);
src_list(1,c_src_id) = 1;
src_list(1,c_src_coordx) = 2;
src_list(1,c_src_coordy) = 2;
src_list(1,c_src_rule) = c_src_rule_rate;
src_list(1,c_src_rule_rate_rate) = 0.3;

slm_make_crowded_map_snapshot = 1;
slm_make_crowded_map_snapshot_filename = strcat('
   ../',fig_root,'source');

run bin/run_simulation
```

```
65
66 src_list = [];
67
68 %% Sink
69
70 map = zeros(3);
71 map(1,2) = c_dn;
72 map(2,2) = c_sn;
73
74 snk_list(1,:) = zeros(1,c_snk_numcols);
75 snk_list(1,c_snk_id) = 1;
76 snk_list(1,c_src_coordx) = 2;
77 snk_list(1,c_src_coordy) = 2;
78 snk_list(1,c_snk_rule) = c_snk_rule_constant;
79 snk_list(1,c_snk_rule_constant_rate) = 0.2;
80
81 slm_make_crowded_map_snapshot = 1;
82 slm_make_crowded_map_snapshot_filename = strcat('
    ../',fig_root,'sink');
83
84 run bin/run_simulation
85
86 snk_list = [];
87
88 %% Waiting field
89
90 map = zeros(3);
91 map(1:3,2) = c_dn;
92 map(2,2) = c_wt;
93
94 jnc_list(1,:) = zeros(1,c_jnc_numcols);
95 jnc_list(1,c_jnc_id) = 1;
96 jnc_list(1,c_jnc_coordx) = 2;
97 jnc_list(1,c_jnc_coordy) = 3;
98 jnc_list(1,c_jnc_rule) = 0;
99 jnc_list(1,[c_jnc_p1 c_jnc_p2 c_jnc_p3 c_jnc_p4])
    = zeros(1,4);
100
101 slm_make_crowded_map_snapshot = 1;
102 slm_make_crowded_map_snapshot_filename = strcat('
    ../',fig_root,'wait');
103
104 run bin/run_simulation
105
106 jnc_list = [];
```

```
107
108 %% Converging junction
109
110 map = zeros(3);
111 map(2:3,2) = c_dn;
112 map(1,2) = c_wt;
113 map(2,1) = c_wt;
114 map(2,3) = c_wt;
115
116 jnc_list(1,:) = zeros(1,c_jnc_numcols);
117 jnc_list(1,c_jnc_id) = 1;
118 jnc_list(1,c_jnc_coordx) = 2;
119 jnc_list(1,c_jnc_coordy) = 2;
120 jnc_list(1,c_jnc_rule) = 0;
121 jnc_list(1,[c_jnc_p1 c_jnc_p2 c_jnc_p3 c_jnc_p4])
      = zeros(1,4);
122
123 slm_make_crowded_map_snapshot = 1;
124 slm_make_crowded_map_snapshot_filename = strcat('
      ../',fig_root,'jnc');
125
126 run bin/run_simulation
127
128 jnc_list = [];
129
130 %% Diverging junction
131
132 map = zeros(3);
133 map(1:3,2) = c_dn;
134 map(2,1) = c_lt;
135 map(2,3) = c_rt;
136
137 jnd_list(1,:) = zeros(1,c_jnd_numcols);
138 jnd_list(1,c_jnd_id) = 1;
139 jnd_list(1,c_jnd_coordx) = 2;
140 jnd_list(1,c_jnd_coordy) = 2;
141 jnd_list(1,c_jnd_indir) = c_up;
142 jnd_list(1,c_jnd_rule) = 1;
143 jnd_list(1,[c_jnd_p1 c_jnd_p2 c_jnd_p3 c_jnd_p4])
      = zeros(1,4);
144
145 slm_make_crowded_map_snapshot = 1;
146 slm_make_crowded_map_snapshot_filename = strcat('
      ../',fig_root,'jnd');
147
```

```
148 run bin/run_simulation
149
150 jnd_list = [];
151
152 %% Packet
153
154 map = zeros(3);
155 map(1:3,2) = c_dn;
156
157 src_list(1,:) = zeros(1,c_src_numcols);
158 src_list(1,c_src_id) = 1;
159 src_list(1,c_src_coordx) = 2;
160 src_list(1,c_src_coordy) = 1;
161 src_list(1,c_src_rule) = c_src_rule_rate;
162 src_list(1,c_src_rule_rate_rate) = 0.3;
163
164 slm_num_steps = 20
165 slm_make_crowded_map_snapshot = 1;
166 slm_make_crowded_map_snapshot_rel_occupation =
      0.1;
167 slm_make_crowded_map_snapshot_filename = strcat('
      ../',fig_root,'packet');
168
169 run bin/run_simulation
170
171 src_list = [];
172
173 %% Buffer layouts
174
175 slm_num_steps = 2
176 slm_make_crowded_map_snapshot = 1;
177 slm_make_crowded_map_snapshot_rel_occupation = -1;
       % empty
178
179 mp_inbufferlen = 0;
180 mp_outbufferlen = 0;
181 mp_jnd_std_rule =
      c_jnd_rule_SearchFreeWayPriorized;
182 mp_jnd_std_rule_priorization = [c_rt c_dn c_dn
      c_dn];
183 mp_jnc_std_rule = c_jnc_rule_TakeIfAnyPriorized;
184 mp_juc_std_rule_priorization = [c_dn c_lt c_lt
      c_lt];
185 mp_jnd_topbuffer_rule =
      c_jnd_rule_SearchFreeWayPriorized;
```

```
186 mp_jnd_topbuffer_rule_priorization = [c_rt c_up
    c_dn c_dn];
187 mp_jnc_topbuffer_rule =
    c_jnc_rule_TakeIfAnyPriorized;
188 mp_juc_topbuffer_rule_priorization =  [c_up c_dn
    c_lt c_lt];
189
190
191 %% - short bypass buffer (for short responses)
192 mp_bufferlen = 0;
193 mp_bufferdepths = [6]; % variate
194 mp_topbuffer = false;
195
196 slm_make_crowded_map_snapshot = 1;
197 slm_make_crowded_map_snapshot_filename = strcat('
    ../',fig_root,'buffer_short_bypass');
198 run maps/MAP_Buffer;
199 run bin/run_simulation
200
201 %% - 2n multidepth buffer
202 mp_bufferlen = 3; % variate
203 mp_bufferdepths = [1 2 4]; % variate
204 mp_topbuffer = false;
205
206 slm_make_crowded_map_snapshot = 1;
207 slm_make_crowded_map_snapshot_filename = strcat('
    ../',fig_root,'buffer_2n_mulitdepth');
208 run maps/MAP_Buffer;
209 run bin/run_simulation
210
211 %% - massive paralization buffer
212 mp_bufferlen = 0;
213 mp_bufferdepths = [1 1 1 1 1]; % variate
214 mp_topbuffer = true;
215
216 slm_make_crowded_map_snapshot = 1;
217 slm_make_crowded_map_snapshot_filename = strcat('
    ../',fig_root,'buffer_massiv_para');
218 run maps/MAP_Buffer;
219 run bin/run_simulation
220
221 %% - ciruclar distribution system
222 clear system_initialized
223 run bin/initialize_simulation
224 slm_make_crowded_map_snapshot = 1;
```

```
225 slm_make_crowded_map_snapshot_filename = strcat('
    ../',fig_root,'CDS_map');
226 run maps/MAP_CDS;
227 run bin/run_simulation
```

References

[1] **Kaeslin, H.:** *Digital Integrated Circuit Design, From VLSI Architectures to CMOS Fabrication*, Cambridge University Press (ISBN-13 978-0-521-88267-5), 2008

[2] **Barad, M., Even Sapir, D.:** *Flexibility in logistic systemsmodeling and performance evaluation* , 2003

[3] **Joachim, I.:** *Logistik im Automobilbau: Logistikkomponenten und Logistiksysteme im Fahrzeugbau*, Hanser Verlag, 2006

[4] **Enachescu, M., Goel, A., McKeown, N., Roughgarden, T.:** *Routers with Very Small Buffers*, 2005

[5] **Pfeiffer, A., Lipovszki, G.:** *Was bedeutet die Simulation in der Logistik?*, Wissenschaftliche Mitteilungen der 14. Frhlingsakademie (ISBN 963 86234 5 4), pp.: 81-86, 2002