Introduction Tutorial to IKR.QEMU-example models

July 12, 2016

University of Stuttgart - Institute of Communication Networks and Computer Engineering

Associated Libraries:	IKR.QEMU-Example
	IKR.QEMU [6]
	IKR.ProtocolSupport [5]
	IKR.SimLib [4]

1 Introduction

This document is intended for introducing new students, who start a bachelor, research or master thesis project related to protocol layer network simulation, to the concepts and opportunities of the IKR QEMU simulation environment [1]. It shows some example approaches to simulate different problems by providing commented parameter files. These approaches are based on the powerful QEMU-Example models DumbbellModel [2] and ParkingLotModel [3].

Please keep in mind that results are the more valuable the more general they are. To prove this, typically many (randomized) simulation runs and a well-understood and appropriate statistical evaluation are necessary. Simlib and the introduced elements in the protocolsupport library produce some statistics. Use and interpret them with careful consideration!

In order to get a sufficient coverage, i.e. run many simulations, use the tools introduced in this document in combination with SimTree [7]. To use (and understand) this tool, also have a look on the Simlib Reference Guide [8] and a closer look on the documentation how to run a simulation [9].

If using IKR compute nodes (cnode<nn>), always use the global scheduler or talk to your supervisor!

Contents

List of Figures

2 Simulation approaches for example problems

2.1 Problem 1: Behavior of different congestion control algorithms

In our scenario we want to use the congestion control (CC) algorithm implementations of real Linux Kernels to be able to simulate realistic connections between TCP/IP stacks. By using the IKR.QEMU [6] library in combination with the IKR.SimLib [4] and the IKR.Protocolsupport [5] library, we can run these Kernels on virtual QEMU machines and use their TCP/IP stack implementation to start and handle TCP connections.
For the start we want to simulate a connection between a client and a server, connected via a bottleneck link. Therefore the client has an infinite amount of data, which he sends to the server. We want to evaluate the client's congestion window size and the status of the queue at the bottleneck link.
For this problem we can use the DumbbellModel [2], in the IKR.QEMU-example library, in an easy form by simply using the following parameters in the sim.par file.

```
DumbbellModel.NumberOfClients = 1          # one client, i.e. sender
DumbbellModel.NumberOfServers = 1          # one server, i.e. receiver
```

Figure 1 shows the resulting topology.

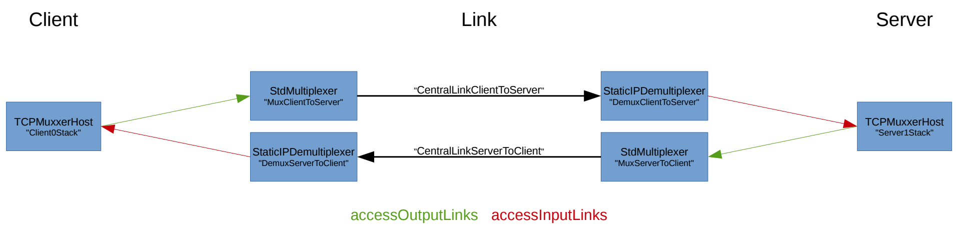

Client
Link
Server
TCPMuxxerHost "Client0Stack"
StdMultiplexer "MuxClientToServer"
"CentralLinkClientToServer"
StaticIPDemultiplexer "DemuxClientToServer"
TCPMuxxerHost "Server1Stack"
StaticIPDemultiplexer "DemuxServerToClient"
"CentralLinkServerToClient"
StdMultiplexer "MuxServerToClient"
accessOutputLinks
accessInputLinks

Figure 1: Topology of the used Dumbbell model (Problem1)

To be able to compare different CC algorithms with each other, we can set the algorithm, the stacks shall use, by the parameter:

```
DumbbellModel.DefaultCongestionControl = reno
```

To generate the log files with the bottleneck queue's status, we have to use a TracingBounded-FIFOQDisc as queuing discipline, instead of the default BoundedFIFOQDisc. We also log the size of the congestion windows by adding the following parameters:

```
DumbbellModel.CentralLinkClientToServer.QueuingEntity.QueuingDiscipline =
ikr.protocolsupport.algorithms.queuingDisciplines.TracingBoundedFIFOQDisc
DumbbellModel.CentralLinkClientToServer.QueuingEntity.QueuingDiscipline
.TraceFile = queue.csv                           # name of the queue log file
DumbbellModel.L4Connection*.LogSocketState = true # log the congestion window
```

Logging socket states is an expensive operation, but may also provide even more detailed data on how the ısender's kernel sees the situtation at a time. While by default only the the sender's congestion window is traced, the following line

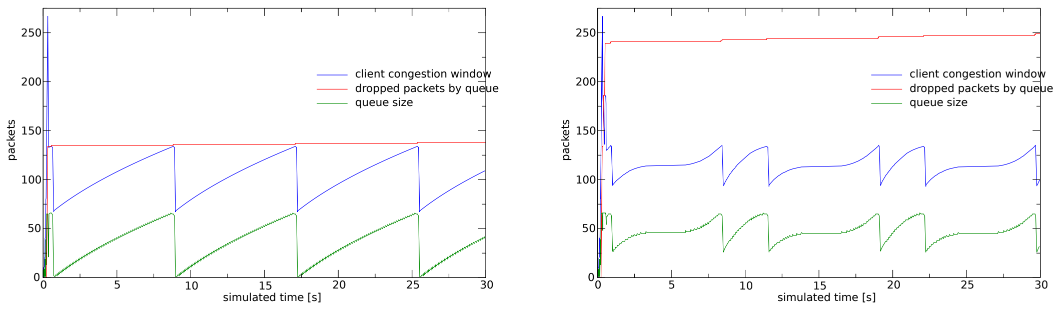
```
DumbbellModel.L4Connection*.LoggedTcpInfoValues = snd_cwnd;snd_ssthresh;rtt;rttvar
```

retrieves

- sender's congestion window (snd_cwnd)
- sender's slow start threshold (snd_ssthresh)
- the current RTT as perceived by the sender (rtt)
- the RTT's variation (rttvar)

You can retrieve any values in the tcp_info struct of the Linux kernel. Remember, that this struct varies (grew) from version to version!

There are several other parameters needed to run the DumbbellModel [2], like the bandwidth and delay of the links. See the attached Problem1.sim (50) file for all used parameters and their purpose. Many of the parameters needed for the DumbbellModel [2] are set by default if they are not set in the .par file. You can see all the parameters used in a simulation run by greping the resulting DumbbellModel.stderr for "query" (command line: "grep query DumbbellModel.stderr"). The results for a run with the attached Problem1.sim (50) file are plotted in Figure 2.



(a) reno congestion control (b) cubic congestion control

Figure 2: Client's Congestion Window and bottleneck queue status for the first 30 seconds

In Figure 2 you can see the different phases of the congestion control algorithms and how they depend on the bottleneck queue's packet drops. Also the differences between the algorithms are visible, e.g. the cubic congestion control in Figure 2b causes more packet drops after slow start than the reno congestion control in Figure 2a.

2.1.1 Testing a new CC algorithm of a modified Kernel

One of the advantages of using a Kernel image in a QEMU is the possibility to quickly change it. If we already compiled a new Kernel with a modified Congestion Control algorithm, we can easily use the scenario mentioned above and simply change the following parameters, to see how our algorithm performs.

```
*.*Stack.QEMU.kernel = "../path/to/yourKernelImage"
DumbbellModel.DefaultCongestionControl = myCC
```

2.2 Problem 2: Fairness of CC algorithms

A major aspect in congestion control designs is fairness. Fairness can be defined in different ways, but mostly means a situation where different TCP flows compete for bandwidth at a bottleneck in a network. So for this problem we want to see how flows with different congestion control algorithms perform at a bottleneck queue. Therefore we can use the DumbbellModel [2] with two clients, each having a greedy connection through a bottleneck link to a server. The resulting topology is shown in Figure 3.

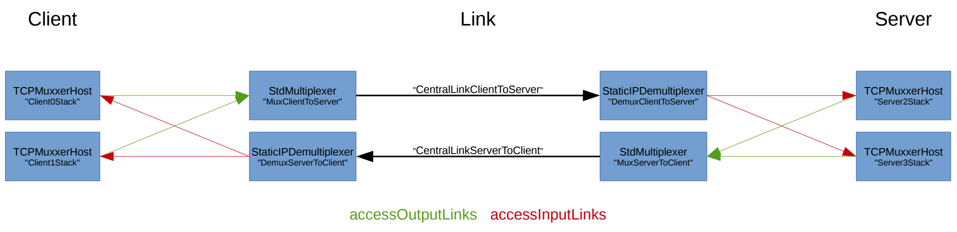
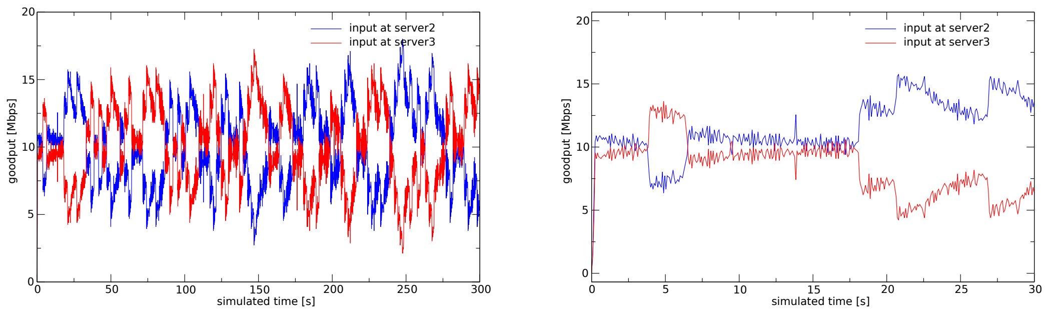

Client
Link
Server
TCPMuxxerHost "Client0Stack"
TCPMuxxerHost "Client1Stack"
StdMultiplexer "MuxClientToServer"
StaticIPDemultiplexer "DemuxServerToClient"
"CentralLinkClientToServer"
"CentralLinkServerToClient"
StaticIPDemultiplexer "DemuxClientToServer"
StdMultiplexer "MuxServerToClient"
TCPMuxxerHost "Server2Stack"
TCPMuxxerHost "Server3Stack"
accessOutputLinks
accessInputLinks

Figure 3: Topology of the used Dumbbell model (Problem2)

To evaluate this problem we are interested in the goodput rate received at the servers. Therefore we need the following settings:

```
DumbbellModel.NumberOfClients = 2          # two clients = senders
DumbbellModel.NumberOfServers = 2          # two server = receivers
DumbbellModel.WriteRateMeter = true        # generates rate logs
```

The interesting log files for us are the Server2Stack.InRates and Server3Stack.InRates, which are plotted in Figure 4.



(a) first 5 minutes (b) first 30 seconds

Figure 4: Goodput rates of two competing reno flows

In Figure 4 we can see how the two flows are competing for bandwidth. In most periods they get an unequal amount of the available bandwidth but in the plot over 5 minutes we can estimate that the distribution of bandwidth is balanced.

By cleverly setting the bandwidth of the accessInputLinks to the bandwidth of the central link, we can use the automatically generated statistics of the accessInputLinks at the server side to measure the average link load. We find these statistics in the generated sim.log file for the CapacityPhase of the nodes AccessInputLinkServer2Stack and AccessInputLinkServer3Stack. The results statistics, of the run plotted in Figure 4, are:

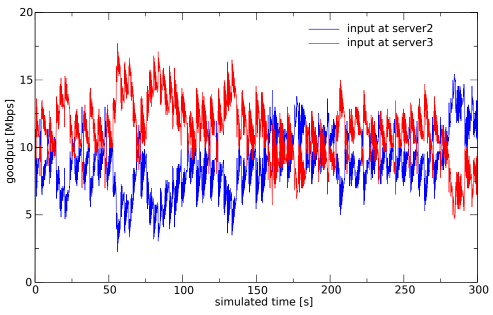
AccessInputLinkServer2Stack.CapacityPhase.meanOccupancy = 0.49821115733333504
AccessInputLinkServer3Stack.CapacityPhase.meanOccupancy = 0.501357906720002

These results show that **in average** the available bandwidth is shared nearly equally between the two competing flows and therefore this situation can be considered to be fair.

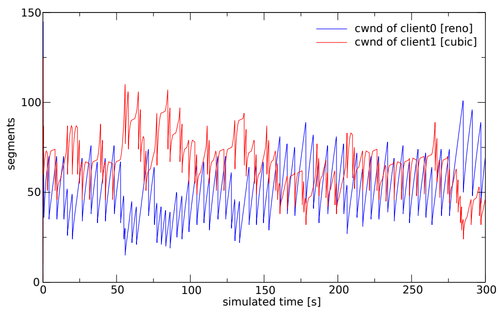
We can also set up a reno flow against a cubic flow, to see how they compete, with the parameter:

```
DumbbellModel.CongestionControl = [reno cubic reno cubic]
```

The results of such a run are plotted in Figure 5:



(a) Goodput rates of reno and cubic flow



(b) Congestion windows of reno and cubic flow

Figure 5: Reno flow vs. cubic flow

Statistics:
AccessInputLinkServer2Stack.CapacityPhase.meanOccupancy = 0.44136550400000213
AccessInputLinkServer3Stack.CapacityPhase.meanOccupancy = 0.5582783046933326

The results shown in Figure 5a and the statistics show, that the bandwidth is also shared nearly equally. The slight plus for the cubic flow may just be a fluctuation or can result from the more aggressive way of the cubic algorithm to request available bandwidth. For statistically reliable results we would need to run this simulation for a much longer time. Figure 5b shows the congestion windows of each flow while they compete. This is only for visualization but not for comparison reasons, because the size of the congestion window strongly depends on the algorithm and the queue behavior and therefore it is not comparable.

2.3 Problem 3: TCP unfairness regarding round trip time

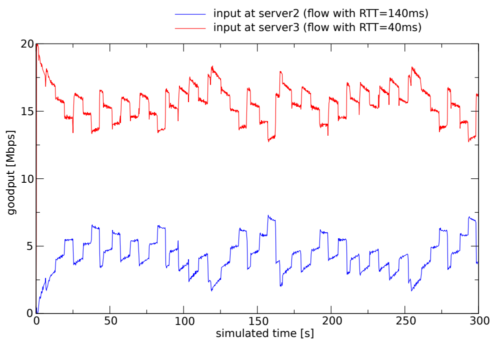
One of the known unfair aspects of todays congestion control algorithms (especially reno) are competition situations, where the competing flows have different round trip times. We can simulate this situation by using the above settings of Problem2.par (44) and simply adjust the delay of the access links. By setting the following parameters, we get different RTTs for the connections of client0 and client1:

```
DumbbellModel.AccessOutputLink*.PropagationDelayInSeconds = 0.00

DumbbellModel.AccessInputLinkClient0Stack.PropagationDelayInSeconds = 0.05
DumbbellModel.AccessInputLinkServer2Stack.PropagationDelayInSeconds = 0.05

DumbbellModel.AccessInputLinkClient1Stack.PropagationDelayInSeconds = 0
DumbbellModel.AccessInputLinkServer3Stack.PropagationDelayInSeconds = 0
```

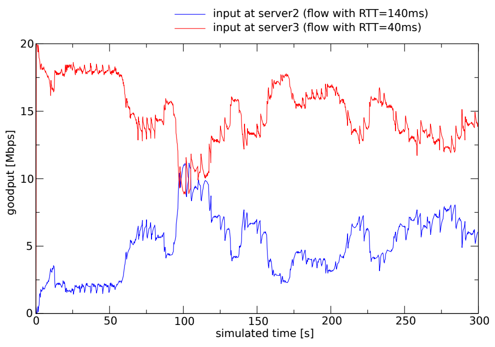
With these settings, we get the following round trip times for the flows:

$$RTT_{\text{client0toServer2}} = 0\,ms + 20\,ms + 50\,ms + 0\,ms + 20\,ms + 50\,ms = 140\,ms$$
$$RTT_{\text{client1toServer3}} = 0\,ms + 20\,ms + 0\,ms + 0\,ms + 20\,ms + 0\,ms = 40\,ms$$

The results for runs with reno and cubic flows are plotted in Figure 6:



(a) reno flows



(b) cubic flows

Figure 6: Goodput rates of two competing flows with different RTTs

Statistics for the reno flows:
AccessInputLinkServer2Stack.CapacityPhase.meanOccupancy = 0.215981370666668843
AccessInputLinkServer3Stack.CapacityPhase.meanOccupancy = 0.781988625013388

Statistics for the cubic flows:
AccessInputLinkServer2Stack.CapacityPhase.meanOccupancy = 0.25293508266669357
AccessInputLinkServer3Stack.CapacityPhase.meanOccupancy = 0.7466114118933846

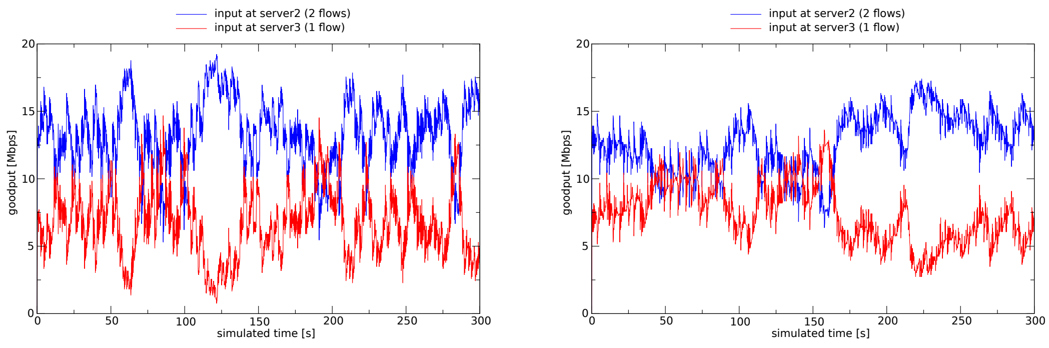
As we can see in Figure 6 and the statistics, the flow with a higher RTT gets less of the available bandwidth. When we compare the results for reno with cubic flows, we can see that there seems to be a slight improvement of the situation if cubic congestion control is used. As mentioned above we would need to simulate for a much longer time to get statistically reliable results, but we can definitely see that this situation is highly unfair. These problems can be solved by intelligent queuing disciplines which e.g. equally distribute packet drops on flows.

2.4 Problem 4: TCP Unfairness regarding number of flows

Another known unfair aspect of todays TCPs are situations, where clients can increase their bandwidth at a bottleneck by simply using more flows. This concept is commonly used by download helper applications, which simply start multiple connections to download fragments of a file and later reassemble it. We can simulate this situation by using the above settings of Problem2.par (44) and simply use two client0ToServer2 connections instead of one. Therefore we need the following parameters:

```
DumbbellModel.NumberOfL4Connections = 3
DumbbellModel.MaxNumberOfFlowsPerStack = 2

DumbbellModel.L4Connection0.Client = 0
DumbbellModel.L4Connection0.Server = 2
DumbbellModel.L4Connection1.Client = 0
DumbbellModel.L4Connection1.Server = 2
DumbbellModel.L4Connection2.Client = 1
DumbbellModel.L4Connection2.Server = 3
```

The results for runs with reno and cubic flows are plotted in Figure 7:



(a) reno flows (b) cubic flows

Figure 7: Goodput rates of two clients with one client having two flows

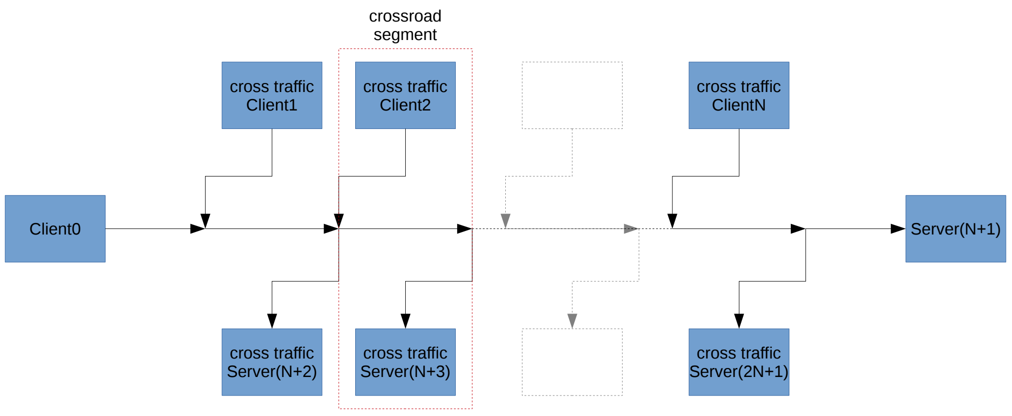
Statistics for the reno flows:
AccessInputLinkServer2Stack.CapacityPhase.meanOccupancy = 0.6701289180266726
AccessInputLinkServer3Stack.CapacityPhase.meanOccupancy = 0.32956165066667

Statistics for the cubic flows:
AccessInputLinkServer2Stack.CapacityPhase.meanOccupancy = 0.6332702373333364
AccessInputLinkServer3Stack.CapacityPhase.meanOccupancy = 0.3664343180266672

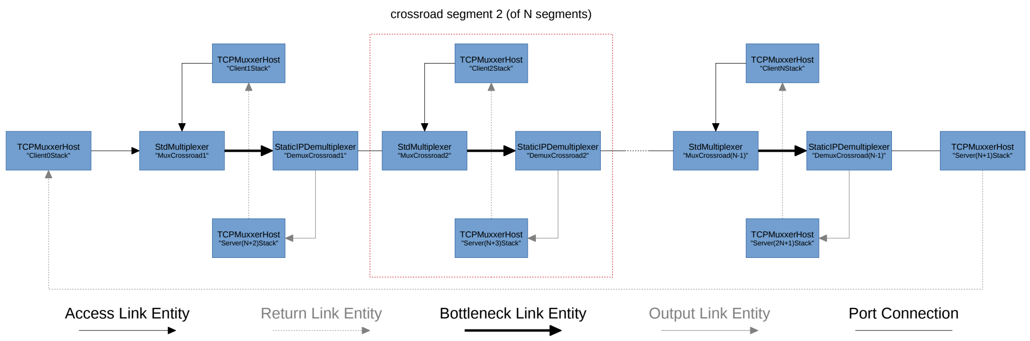
As we can see in Figure 7 and the statistics, the client with two flows gets more of the available bandwidth. In this case the available bandwidth is equally distributed over the flows, which means that every flow gets a third of it. From a flow perspective this is fair, but from a client perspective, which often is more relevant, this is highly unfair. These problems can be solved by intelligent queuing disciplines which e.g. equally distribute packet drops on clients.

2.5 Problem 5: TCP Unfairness for flows facing multiple bottlenecks

For this situation, we assume a connection between a client and a server, which passes several network nodes and therefore several links with queuing entities, similar to a connection trough the internet. In this case the flow has to share the available bandwidth of each link with other flows that pass this link. We call these flows cross traffic. If these links are the bottleneck for this cross traffic, the Active Queue Management (AQM) of this queue will (have to) drop packets of the competing flows as well as of the flow traversing several of these bottleneck queues. In our model we want to evaluate the effect of these multiple bottlenecks on a central flow. For this purpose we can use the ParkingLotModel [3] in the IKR.QEMU-example library. Figure 8a shows an overview of this topology and Figure 8b the details of ParkingLotModel [3].



(a) Overview


crossroad segment 2 (of N segments)
TCPMuxxerHost "Client1Stack"
TCPMuxxerHost "Client2Stack"
TCPMuxxerHost "ClientNStack"
TCPMuxxerHost "Client0Stack"
StdMultiplexer "MuxCrossroad1"
StaticIPDemultiplexer "DemuxCrossroad1"
StdMultiplexer "MuxCrossroad2"
StaticIPDemultiplexer "DemuxCrossroad2"
StdMultiplexer "MuxCrossroad(N-1)"
StaticIPDemultiplexer "DemuxCrossroad(N-1)"
TCPMuxxerHost "Server(N+1)Stack"
TCPMuxxerHost "Server(N+2)Stack"
TCPMuxxerHost "Server(N+3)Stack"
TCPMuxxerHost "Server(2N+1)Stack"
Access Link Entity
Return Link Entity
Bottleneck Link Entity
Output Link Entity
Port Connection

(b) Detail

Figure 8: Topology of ParkingLotModel [3]

We want the crossroad links to all have the same bandwidth and delay and the cross traffic flows to have about the same RTT as the central flow. As the packets of the central flow pass several queues, they suffer from several queue waiting times and therefore we can only estimate the RTT for the central flow. So for this problem we use the following parameters:

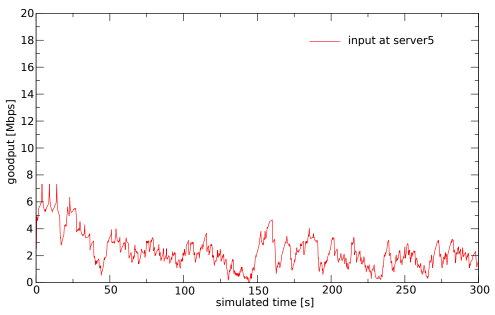
```
# Topology settings:
ParkingLotModel.NumberOfCrossRoads = 4
# Link settings:
ParkingLotModel.CentralAccessLink.PropagationDelayInSeconds = 0
ParkingLotModel.CentralAccessLink.TransmissionRateInMBitS = 100
ParkingLotModel.CentralReturnLink.PropagationDelayInSeconds = 0
ParkingLotModel.CentralReturnLink.TransmissionRateInMBitS = 100
ParkingLotModel.CentralBottleneckLink*.PropagationDelayInSeconds = 0.01
ParkingLotModel.CentralBottleneckLink*.TransmissionRateInMBitS = 20
ParkingLotModel.CrossAccessLink*.PropagationDelayInSeconds = 0
ParkingLotModel.CrossAccessLink*.TransmissionRateInMBitS = 100
ParkingLotModel.CrossOutputLink*.PropagationDelayInSeconds = 0.04
ParkingLotModel.CrossOutputLink*.TransmissionRateInMBitS = 100
ParkingLotModel.CrossReturnLink*.PropagationDelayInSeconds = 0
ParkingLotModel.CrossReturnLink*.TransmissionRateInMBitS = 100
# Possible generic outputs:
ParkingLotModel.WriteRateMeter = true
```

With these settings we get the following round trip times for the flows:

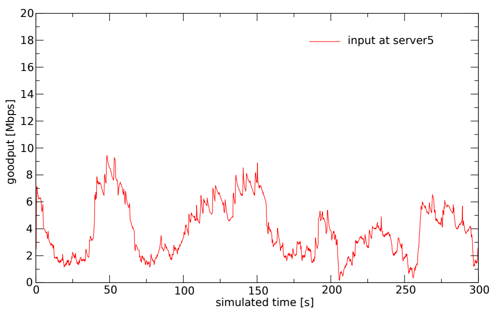
$$\begin{aligned}
RTT_{\text{central flow}} &= 0\,ms + 10\,ms + 10\,ms + 10\,ms + 10\,ms + 0\,ms + 4 \cdot queueWaitingTime_{4\,queues} \\
&= 40\,ms + 4 \cdot queueWaitingTime_{4\,queues} \\
RTT_{\text{client1toServer3}} &= 0\,ms + 10\,ms + 40\,ms + 0\,ms + queueWaitingTime_{1\,queue} \\
&= 50\,ms + queueWaitingTime_{1\,queue}
\end{aligned}$$

Keep in mind, that $0 \leq queueWaitingTime_{4\,queues} \leq 4 \cdot queueWaitingTime_{1\,queue}$!

The resulting goodput rate of the central flow measured at the input port of server5 (Server5.InRate) is plotted in Figure 9:



(a) reno flows



(b) cubic flows

Figure 9: Goodput rate of the central flow

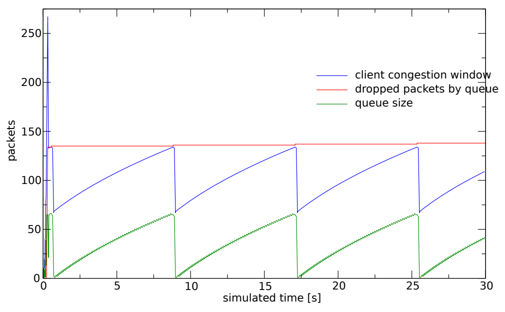
The perfectly fair result would be the central flow getting half of the available bandwidth at each bottleneck link and therefore having a goodput rate of about 10 Mbps. As we can see in Figure 9a and in Figure 9b this is not the case for both reno and cubic CC. The problem is, that both of these congestion control algorithms are based on losses (and/or on marked packets, if ECN is used) and the fact that the central flow suffers from losses in multiple queues. In this situation the central flow gets more congestion signals than the cross flows do and therefore reduces its sending rate more often.

2.6 Problem 6: Using AQM queues

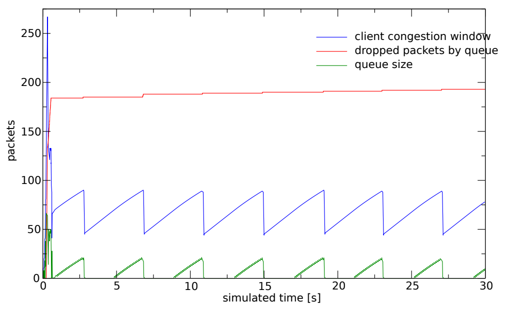
The behavior of TCP flows is not only depending on the used congestion control algorithms, but also on the queuing disciplines used in the queuing entities in front of the bottlenecks passed by a flow. For the previous problems we always used the default BoundedFIFOQDisc for the link queues. This algorithm is quite simple, because it just drops packets if the queue's buffer is full. These so called drop-tail queues have many drawbacks, e.g. global synchronization and the penalization of bursty flows. In order to reduce the overall congestion rate (i.e. the number / rate of packet drops), AQM queues start dropping packets before the buffer is full to give the sender an early signal that there is congestion in the network. This is also reflected in the name of the first and most deployed AQM: Random Early Detection (RED). This obviously requires that the queue has more buffer space available than is the lowest threshold for signalling congestion (e.g. the minimum threshold for RED). For this problem we just want to compare the results of the BoundedFIFOQDisc of Problem 1 with the results of a RED queuing discipline. Therefore we use the parameters of Problem 1 with the RedIpQDisc:

```
DumbbellModel.CentralLinkClientToServer.QueuingEntity.QueuingDiscipline =
ikr.protocolsupport.algorithms.queuingDisciplines.RedIpQDisc

*.CentralLinkClientToServer.*.QueuingDiscipline.Tracing = true
*.CentralLinkClientToServer.*.QueuingDiscipline.TraceFile = RedQueueLog.dat
```

Results are plotted in Figure 10 and Figure 11

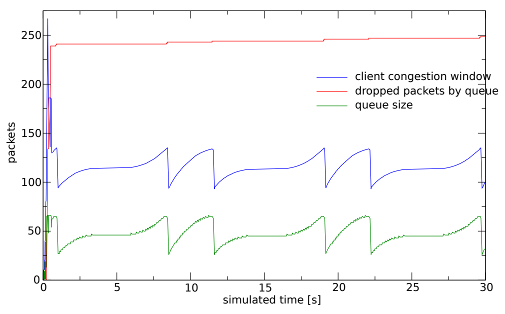


(a) FIFO Queue

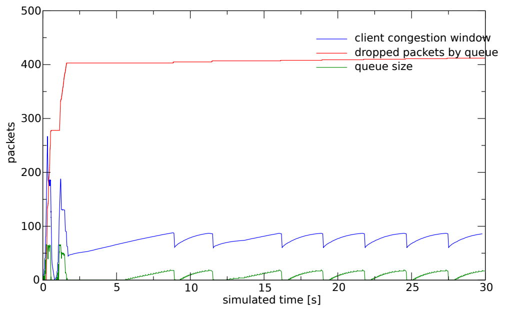


(b) RED Queue

Figure 10: Client's Congestion Window and bottleneck queue status for reno flows



(a) FIFO Queue



(b) RED Queue

Figure 11: Client's Congestion Window and bottleneck queue status for cubic flows

In Figure 10 and Figure 11 we can see that the RED queue tries to hold its queuing size at a low level by starting to drop packets at a buffer size of 25%. Therefore, usually the queuesize is chosen four times bigger than for drop tail queues.

So, always take care in choosing the right parameters for your simulation! Default parameters may exist, that just make sense for certain configuration while they are really wrong for others. In this case, we can also see that low default buffer size is by far not ideal for the simulated situation with only one flow. The queue often runs empty and so the link is not fully occupied (mean occupation statistics for reno: 0.9155606352000085, for cubic: 0.9552077923333341). But for further simulations it would be interesting to evaluate whether several flows passing a RED queue fill the link and if the average RTT is lower than with a tail drop queue.

3 Acronyms

RTT Round Trip Time

CC congestion control

AQM Active Queue Management

RED Random Early Detection

TCP Transmission Control Protocol

References

[1] Werthmann et al. - "VMSimInt: a network simulation tool supporting integration of arbitrary kernels and applications", Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques Pages 56-65, 2014. Available from `http://www.ikr.uni-stuttgart.de/Content/Publications/Archive/We_SIMUTools_2014_40209.pdf`

[2] DumbbellModel.java, Documentation `/net/arch/ikr/qemu/qemu-example-html/ikr/qemu/example/models/DumbbellModel.html`

[3] ParkingLotModel.java, Documentation `/net/arch/ikr/qemu/qemu-example-html/ikr/qemu/example/models/ParkingLotModel.html`

[4] IKR.SimLib java library, documentation `/net/arch/ikr/simlib-java/html/index.html`

[5] IKR.Protocolsupport java library, Documentation in accompanying jar-file. If release exists also at `/net/arch/ikr/protocolsupport/html/index.html`

[6] IKR.QEMU java library, Documentation in accompanying jar-file. If release exists also at `/net/arch/ikr/qemu/qemu-java-html/index.html`

[7] SimTree tool, documentation at `file:///net/arch/opt/misc/doc/SimTree.html`

[8] "IKR Simulation Library User Guide, Reference Guide" at `file:///net/arch/ikr/simlib-doc/ReferenceGuide.pdf`

[9] "IKR Simulation Library User Guide, Simulation Example" at `file:///net/arch/ikr/simlib-doc/SimulationExample.pdf`

4 Appendix

4.1 Sketching graphs

There are many different programs available to plot graphs. Just to name some of them, there are gnuplot, Matlab, grace and matplotlib. For this tutorial we used xmgrace, the gui version of grace. Its gui is not very pretty but it is helpful after you get used to it. Here are some short hints on how to use it.

At first you have to bring the data in a form that is readable for xmgrace. It needs .dat files and the data sets have to be separated by spaces. Lines beginning with # are ignored. You can either manipulate the LineWriter's output if you have access to the source code, or you replace all semicolons by spaces with the command:

```
sed 's/;/ /g' ParkingLotModel.Server5Stack.InRates > Server5Stack.InRates.dat
```

Then you start xmgrace and import the data set. You can do this on the command line, see its documentation (also loading multiple sets and appearance parameters in command line is possible).
`xmgrace ParkingLotModel.Server5Stack.InRates.dat`
or
`"Data > Import > ASCII"`
If the .dat file contains multiple data sets, you choose "`Load as NXY`".

After this you can set the appearance as you wish. For some problems there are parameter files for the xmgrace settings added, use them by loading with "`Plot > Load parameters`".

4.2 Parameter files

See the attached *.par files with comments to all used parameters

```
# Parameters for Problem 1:
# These are basically the minimalDumbbell parameters with the addition of the needed key
    parameters for Problem 1
ModelName = Tutorial1

# QEMU settings:
*.*.*.Binary = "../ikr.qemu-java/lib/qemu"          # path to the QEMU application binary (
    patched to work together with the IKR.QEMU-Java library)
*.*.*.BiosPath = "../ikr.qemu-java/lib"        # path to the bios used by the QEMU application
*.*Stack.QEMU.kernel = "../ikr.qemu-java/lib/bzImage-3.10.9"          # path to the Kernel
    image
*.*Stack.QEMU.initrd = "../ikr.qemu-java/lib/initramfs.cpio"          # path to the ram image
     the QEMU shall use for initiation
*.*Stack.QEMU.dynticks = true          # use dynamic ticks to only wake the emulated Kernel
    when it is needed (see more in ikr.qemu.QEMU.java documentation) /not important for this
    task
*.*.QEMU.LoadSnapshot = false          # don't load snapshot, instead boot Kernel /not
    important for this task

# should be faster, but doesn't work with kernel 3.5 (with 3.10 it's OK)
*.*Stack.QEMU.virtIOConsole = false          # don't use virtual QEMU console /not important
     for this task

# Link settings:
Tutorial1.CentralLink*.PropagationDelayInSeconds = 0.02          # one way propagation delay for
     the central links in seconds
Tutorial1.CentralLink*.TransmissionRateInMBitS = 20          # transmission rate for
     the central links in seconds
Tutorial1.Access*Link*.PropagationDelayInSeconds = 0.00          # one way propagation delay for
     the access links in seconds. Set to 0, but this link is important anyway for spreading
    packet bursts of the TCPMuxxerHosts. Because the Kernel in the QEMU does not run in
    simulation time (to not be hardware dependent) it can send several messages at once in
    simulation time. Therefore this access link with 0 delay but with a bandwidth is needed to
     spread these packet bursts over simulation time.
Tutorial1.Access*Link*.TransmissionRateInMBitS = 1000          # transmission rate for the
    access links in seconds. These access links can be interpreted as 1 Gbps line cards used
    by the QEMU stacks.
Tutorial1.Access*Link*.QueuingEntity.QueuingDiscipline.MaxNumberOfMessages = 11 # needed to
    accept ack + initial window

# Topology settings:
Tutorial1.NumberOfClients = 1          # only one client is needed for this task
Tutorial1.NumberOfServers = 1          # only one server is needed for this task

# Task specific settings:
Tutorial1.DefaultCongestionControl = cubic          # sets the congestion control algorithm
     all initiated stacks shall use via sysctl. For the shown results of this task we used "
    reno" and "cubic".
Tutorial1.CentralLinkClientToServer.QueuingEntity.QueuingDiscipline = ikr.protocolsupport.
    algorithms.queuingDisciplines.TracingBoundedFIFOQDisc          # we want to use a bounded
    fifo queue which implements tracing to be able to log the queue's status

# Simulation control settings:
Tutorial1.TransientDuration = 0          # due to the fact that we do not need statistical
    results we do not need a transient (warm up) phase
Tutorial1.BatchDuration = 3          # by setting the batch duration to 3 seconds we get a
    total simulation time of 3s * 10 = 30s

# Possible generic outputs:


# These are all parameters we need to set for this task. There are several other parameters
    needed to run this model which are set by default if we do not set them here.
# After you ran a simulation, you can see all used parameters by simply greping for "query" in
    the *stderr log file:
# command line example: grep query Tutorial1.stderr
```

```
Tutorial1.L4Connection*.LogSocketState = true
Tutorial1.L4Connection*.LoggedTcpInfoValues = snd_cwnd;snd_ssthresh;rtt;rttvar # sets up a
     logger which writes the selected state of the sockets, e.g. the congestion window
Tutorial1.L4Connection0OfStack0.Client.CongestionControl = reno  # apart from the model's
     default congestion control, every single sender, i.e.  BidirectionalConnectionEndpoint,
     may have its congestion control configured

Tutorial1.WriteCentralRateMeter = true
Tutorial1.SamplePeriodInSeconds = 0.02
Tutorial1.WritePcap = true
```

Listing 1: attached file Problem1.par

```
# Parameters for Problem 2:
ModelName = Tutorial2

# QEMU settings:
*.*.*.Binary = "../ikr.qemu-java/lib/qemu"              # path to the QEMU application binary (
     patched to work together with the IKR.QEMU-Java library)
*.*.*.BiosPath = "../ikr.qemu-java/lib"          # path to the bios used by the QEMU application
*.*Stack.QEMU.kernel = "../ikr.qemu-java/lib/bzImage-3.10.9"            # path to the Kernel
     image
*.*Stack.QEMU.initrd = "../ikr.qemu-java/lib/initramfs.cpio"              # path to the ram image
      the QEMU shall use for initiation
*.*Stack.QEMU.dynticks = true             # use dynamic ticks to only wake the emulated Kernel
     when it is needed (see more in ikr.qemu.QEMU.java documentation) /not important for this
     task
*.*.QEMU.LoadSnapshot = false             # don't load snapshot, instead boot Kernel /not
     important for this task

# should be faster, but doesn't work with kernel 3.5 (with 3.10 it's OK)
*.*Stack.QEMU.virtIOConsole = false               # don't use virtual QEMU console /not important
      for this task

# Link settings:
Tutorial2.CentralLink*.PropagationDelayInSeconds = 0.02           # one way propagation delay for
      the central links in seconds
Tutorial2.CentralLink*.TransmissionRateInMBitS = 20                       # transmission rate for
      the central links in seconds
Tutorial2.Access*Link*.PropagationDelayInSeconds = 0.00           # one way propagation delay for
      the access links in seconds. Set to 0, but this link is important anyway for spreading
     packet bursts of the TCPMuxxerHosts. Because the Kernel in the QEMU does not run in
     simulation time (to not be hardware dependent) it can send several messages at once in
     simulation time. Therefore this access link with 0 delay but with a bandwidth is needed to
      spread these packet bursts over simulation time.
Tutorial2.AccessOutputLink*.TransmissionRateInMBitS = 1000       # transmission rate for the
     access links in seconds. These access links can be interpreted as 1 Gbps line cards used
     by the QEMU stacks.
Tutorial2.AccessInputLink*.TransmissionRateInMBitS = 20          # set to the same as the
     central links transmission rate to be able to simply use the statistics to see the average
      bandwidth of each flow
Tutorial2.Access*Link*.QueuingEntity.QueuingDiscipline.MaxNumberOfMessages = 11 # needed to
     accept ack + initial window

# Topology settings:
Tutorial2.NumberOfClients = 2             # two clients needed for this task
Tutorial2.NumberOfServers = 2             # two servers needed for this task

# Task specific settings:
#Tutorial2.DefaultCongestionControl = reno                # sets the congestion control algorithm
      all initiated stacks shall use via sysctl. For the shown results of this task we used "
     reno" and "cubic".
Tutorial2.CongestionControl = [ reno cubic reno cubic ]          # sets the congestion control
     algorithm for all stacks separately

# Simulation control settings:
Tutorial2.TransientDuration = 0          # due to the fact that we do not need statistical
     results we do not need a transient (warm up) phase
Tutorial2.BatchDuration = 30    # by setting the batch duration to 30 seconds we get a total
     simulation time of 30s * 10 = 300s

# Possible generic outputs:
#Tutorial2.L4Connection*.LogSocketState = true           # sets up a congestion window logger
     which writes a *cwnd.csv log file
Tutorial2.WriteRateMeter = true # sets up rate meters at all stack ports


# These are all parameters we need to set for this task. There are several other parameters
     needed to run this model which are set by default if we do not set them here.
# After you ran a simulation, you can see all used parameters by simply greping for "query" in
     the *stderr log file:
# command line example: grep query Tutorial2.stderr
```

Listing 2: attached file Problem2.par

```
# Parameters for Problem 3:
ModelName = Tutorial3

# QEMU settings:
```

```
*.*.*.Binary = "../ikr.qemu-java/lib/qemu"              # path to the QEMU application binary (
     patched to work together with the IKR.QEMU-Java library)
*.*.*.BiosPath = "../ikr.qemu-java/lib"        # path to the bios used by the QEMU application
*.*Stack.QEMU.kernel = "../ikr.qemu-java/lib/bzImage-3.10.9"              # path to the Kernel
     image
*.*Stack.QEMU.initrd = "../ikr.qemu-java/lib/initramfs.cpio"              # path to the ram image
      the QEMU shall use for initiation
*.*Stack.QEMU.dynticks = true           # use dynamic ticks to only wake the emulated Kernel
     when it is needed (see more in ikr.qemu.QEMU.java documentation) /not important for this
     task
*.*.QEMU.LoadSnapshot = false           # don't load snapshot, instead boot Kernel /not
     important for this task

# should be faster, but doesn't work with kernel 3.5 (with 3.10 it's OK)
*.*Stack.QEMU.virtIOConsole = false            # don't use virtual QEMU console /not important
      for this task

# Link settings:
Tutorial3.CentralLink*.PropagationDelayInSeconds = 0.02          # one way propagation delay for
      the central links in seconds
Tutorial3.CentralLink*.TransmissionRateInMBitS = 20                       # transmission rate for
      the central links in seconds

Tutorial3.AccessOutputLink*.PropagationDelayInSeconds = 0.00                       # one way
     propagation delay for the access output links in seconds. Set to 0, but this link is
     important anyway for spreading packet bursts of the TCPMuxxerHosts. Because the Kernel in
     the QEMU does not run in simulation time (to not be hardware dependent) it can send
     several messages at once in simulation time. Therefore this access link with 0 delay but
     with a bandwidth is needed to spread these packet bursts over simulation time.

Tutorial3.AccessInputLinkClient0Stack.PropagationDelayInSeconds = 0.05   # added delay for flow
     client0ToServer2
Tutorial3.AccessInputLinkServer2Stack.PropagationDelayInSeconds = 0.05   # added delay for flow
     client0ToServer2

Tutorial3.AccessInputLinkClient1Stack.PropagationDelayInSeconds = 0             # no added
     delay for flow client1ToServer3
Tutorial3.AccessInputLinkServer3Stack.PropagationDelayInSeconds = 0             # no added
     delay for flow client0ToServer2


Tutorial3.AccessOutputLink*.TransmissionRateInMBitS = 1000       # transmission rate for the
     access links in seconds. These access links can be interpreted as 1 Gbps line cards used
     by the QEMU stacks.
Tutorial3.AccessInputLink*.TransmissionRateInMBitS = 20          # set to the same as the
     central links transmission rate to be able to simply use the statistics to see the average
      bandwidth of each flow
Tutorial3.Access*Link*.QueuingEntity.QueuingDiscipline.MaxNumberOfMessages = 11 # needed to
     accept ack + initial window


# Topology settings:
Tutorial3.NumberOfClients = 2           # two clients needed for this task
Tutorial3.NumberOfServers = 2           # two servers needed for this task

# Task specific settings:
Tutorial3.DefaultCongestionControl = reno              # sets the congestion control algorithm
      all initiated stacks shall use via sysctl. For the shown results of this task we used "
     reno" and "cubic".

# Simulation control settings:
Tutorial3.TransientDuration = 0         # due to the fact that we do not need statistical
     results we do not need a transient (warm up) phase
Tutorial3.BatchDuration = 30    # by setting the batch duration to 30 seconds we get a total
     simulation time of 30s * 10 = 300s

# Possible generic outputs:
Tutorial3.WriteRateMeter = true # sets up rate meters at all stack ports


# These are all parameters we need to set for this task. There are several other parameters
     needed to run this model which are set by default if we do not set them here.
# After you ran a simulation, you can see all used parameters by simply greping for "query" in
     the *stderr log file:
# command line example: grep query Tutorial3.stderr
```

Listing 3: attached file Problem3.par

```
# Parameters for Problem 4:
ModelName = Tutorial4

# QEMU settings:
*.*.*.Binary = "../ikr.qemu-java/lib/qemu"              # path to the QEMU application binary (
     patched to work together with the IKR.QEMU-Java library)
*.*.*.BiosPath = "../ikr.qemu-java/lib"        # path to the bios used by the QEMU application
*.*Stack.QEMU.kernel = "../ikr.qemu-java/lib/bzImage-3.10.9"              # path to the Kernel
     image
*.*Stack.QEMU.initrd = "../ikr.qemu-java/lib/initramfs.cpio"              # path to the ram image
      the QEMU shall use for initiation
*.*Stack.QEMU.dynticks = true           # use dynamic ticks to only wake the emulated Kernel
     when it is needed (see more in ikr.qemu.QEMU.java documentation) /not important for this
     task
```

```
*.*.QEMU.LoadSnapshot = false             # don't load snapshot, instead boot Kernel /not
     important for this task

# should be faster, but doesn't work with kernel 3.5 (with 3.10 it's OK)
*.*Stack.QEMU.virtIOConsole = false              # don't use virtual QEMU console /not important
      for this task

# Link settings:
Tutorial4.CentralLink*.PropagationDelayInSeconds = 0.02           # one way propagation delay for
      the central links in seconds
Tutorial4.CentralLink*.TransmissionRateInMBitS = 20                         # transmission rate for
      the central links in seconds
Tutorial4.TypeOfAccess*Link* = ikr.protocolsupport.entities.Link # because of the balanced
     scenario we will have bottlenecks before the central link, too, so we can't use NoDropLink
      i.e. NoDropBoundedFIFOQDisc
Tutorial4.Access*Link*.PropagationDelayInSeconds = 0.00           # one way propagation delay for
      the access links in seconds. Set to 0, but this link is important anyway for spreading
     packet bursts of the TCPMuxxerHosts. Because the Kernel in the QEMU does not run in
     simulation time (to not be hardware dependent) it can send several messages at once in
     simulation time. Therefore this access link with 0 delay but with a bandwidth is needed to
      spread these packet bursts over simulation time.
Tutorial4.AccessOutputLink*.TransmissionRateInMBitS = 1000        # transmission rate for the
     access links in seconds. These access links can be interpreted as 1 Gbps line cards used
     by the QEMU stacks.
Tutorial4.AccessInputLink*.TransmissionRateInMBitS = 20           # set to the same as the
     central links transmission rate to be able to simply use the statistics to see the average
      bandwidth of each flow


# Topology settings:
Tutorial4.NumberOfClients = 2             # two clients needed for this task
Tutorial4.NumberOfServers = 2             # two servers needed for this task

# Task specific settings:
Tutorial4.DefaultCongestionControl = reno                # sets the congestion control algorithm
      all initiated stacks shall use via sysctl. For the shown results of this task we used "
     reno" and "cubic".

Tutorial4.NumberOfL4Connections = 3             # number of total connections
Tutorial4.MaxNumberOfFlowsPerStack = 2  # set the maximal number of flows per stack. This is
     important for the buffer calculation of the TCPMuxxerHosts

Tutorial4.L4Connection0.Client = 0                # client of connection 0 is client0
Tutorial4.L4Connection0.Server = 2                # server of connection 0 is server2
Tutorial4.L4Connection1.Client = 0                # client of connection 1 is client0
Tutorial4.L4Connection1.Server = 2                # server of connection 1 is server2
Tutorial4.L4Connection2.Client = 1                # client of connection 2 is client1
Tutorial4.L4Connection2.Server = 3                # server of connection 2 is server3


# Simulation control settings:
Tutorial4.TransientDuration = 0          # due to the fact that we do not need statistical
     results we do not need a transient (warm up) phase
Tutorial4.BatchDuration = 30    # by setting the batch duration to 30 seconds we get a total
     simulation time of 30s * 10 = 300s

# Possible generic outputs:
Tutorial4.WriteRateMeter = true # sets up rate meters at all stack ports


# These are all parameters we need to set for this task. There are several other parameters
     needed to run this model which are set by default if we do not set them here.
# After you ran a simulation, you can see all used parameters by simply greping for "query" in
     the *stderr log file:
# command line example: grep query Tutorial4.stderr
```

Listing 4: attached file Problem4.par

```
# Parameters for Problem 5:
ModelName = Tutorial5

# QEMU settings:
*.*.*.Binary = "../ikr.qemu-java/lib/qemu"              # path to the QEMU application binary (
     patched to work together with the IKR.QEMU-Java library)
*.*.*.BiosPath = "../ikr.qemu-java/lib"          # path to the bios used by the QEMU application
*.*Stack.QEMU.kernel = "../ikr.qemu-java/lib/bzImage-3.10.9"             # path to the Kernel
     image
*.*Stack.QEMU.initrd = "../ikr.qemu-java/lib/initramfs.cpio"             # path to the ram image
      the QEMU shall use for initiation
*.*Stack.QEMU.dynticks = true            # use dynamic ticks to only wake the emulated Kernel
     when it is needed (see more in ikr.qemu.QEMU.java documentation) /not important for this
     task
*.*.QEMU.LoadSnapshot = false             # don't load snapshot, instead boot Kernel /not
     important for this task

# should be faster, but doesn't work with kernel 3.5 (with 3.10 it's OK)
*.*Stack.QEMU.virtIOConsole = false              # don't use virtual QEMU console /not important
      for this task

# Link settings:
Tutorial5.CentralAccessLink.PropagationDelayInSeconds = 0               # the access link can
     be interpreted as 100 Mbps line card used by the QEMU stack
Tutorial5.CentralAccessLink.TransmissionRateInMBitS = 100
```

```
Tutorial5.CentralReturnLink.PropagationDelayInSeconds = 0                # return link for the
     central connection, has no influence in this model
Tutorial5.CentralReturnLink.TransmissionRateInMBitS = 100

Tutorial5.CentralBottleneckLink*.PropagationDelayInSeconds = 0.01                # delay of the
     bottleneck link in a crossroad segment
Tutorial5.CentralBottleneckLink*.TransmissionRateInMBitS = 20                    # transmission
     rate of the bottleneck link in a crossroad segment

Tutorial5.CrossAccessLink*.PropagationDelayInSeconds = 0                # the access link can
     be interpreted as 100 Mbps line card used by the QEMU stack
Tutorial5.CrossAccessLink*.TransmissionRateInMBitS = 100

Tutorial5.CrossOutputLink*.PropagationDelayInSeconds = 0.04      # delay of the cross output
     link, set to 40ms to approximate the cross flow's RTT to the central flow's RTT
Tutorial5.CrossOutputLink*.TransmissionRateInMBitS = 100

Tutorial5.CrossReturnLink*.PropagationDelayInSeconds = 0                # return link for the
     cross flow connections
Tutorial5.CrossReturnLink*.TransmissionRateInMBitS = 100

# Topology settings:
Tutorial5.NumberOfCrossRoads = 4                    # number of the cross road segments

# Task specific settings:
Tutorial5.DefaultCongestionControl = reno                # sets the congestion control algorithm
      all initiated stacks shall use via sysctl. For the shown results of this task we used "
     reno" and "cubic".

Tutorial5.*Stack.BufPerFlow = 500000              # buffer size per flow, must be enough to not
     trigger kernel routines which try to influence buffer sizes of connections in order to
     save memory space
Tutorial5.*Stack.QEMU.systemMemory = 64.0M       # qemu system's memory space


# Simulation control settings:
Tutorial5.TransientDuration = 0          # due to the fact that we do not need statistical
     results we do not need a transient (warm up) phase
Tutorial5.BatchDuration = 30                      # by setting the batch duration to 30 seconds
     we get a total simulation time of 30s * 10 = 300s

# Possible generic outputs:
Tutorial5.WriteRateMeter = true          # we want to log the rates
Tutorial5.WritePcap = false                       # default is true, but not needed for this
     problem


# These are all parameters we need to set for this task. There are several other parameters
     needed to run this model which are set by default if we do not set them here.
# After you ran a simulation, you can see all used parameters by simply greping for "query" in
     the *stderr log file:
# command line example: grep query Tutorial5.stderr
```

Listing 5: attached file Problem5.par

```
# Parameters for Problem 6:
ModelName = Tutorial6

# QEMU settings:
*.*.*.Binary = "../ikr.qemu-java/lib/qemu"                # path to the QEMU application binary (
     patched to work together with the IKR.QEMU-Java library)
*.*.*.BiosPath = "../ikr.qemu-java/lib"          # path to the bios used by the QEMU application
*.*Stack.QEMU.kernel = "../ikr.qemu-java/lib/bzImage-3.10.9"            # path to the Kernel
     image
*.*Stack.QEMU.initrd = "../ikr.qemu-java/lib/initramfs.cpio"            # path to the ram image
      the QEMU shall use for initiation
*.*Stack.QEMU.dynticks = true            # use dynamic ticks to only wake the emulated Kernel
     when it is needed (see more in ikr.qemu.QEMU.java documentation) /not important for this
     task
*.*.QEMU.LoadSnapshot = false            # don't load snapshot, instead boot Kernel /not
     important for this task

# should be faster, but doesn't work with kernel 3.5 (with 3.10 it's OK)
*.*Stack.QEMU.virtIOConsole = false              # don't use virtual QEMU console /not important
      for this task

# Link settings:
Tutorial6.CentralLink*.PropagationDelayInSeconds = 0.02          # one way propagation delay for
      the central links in seconds
Tutorial6.CentralLink*.TransmissionRateInMBitS = 20                      # transmission rate for
      the central links in seconds
Tutorial6.Access*Link*.PropagationDelayInSeconds = 0.00          # one way propagation delay for
      the access links in seconds. Set to 0, but this link is important anyway for spreading
     packet bursts of the TCPMuxxerHosts. Because the Kernel in the QEMU does not run in
     simulation time (to not be hardware dependent) it can send several messages at once in
     simulation time. Therefore this access link with 0 delay but with a bandwidth is needed to
      spread these packet bursts over simulation time.
Tutorial6.Access*Link*.TransmissionRateInMBitS = 1000            # transmission rate for the
     access links in seconds. These access links can be interpreted as 1 Gbps line cards used
     by the QEMU stacks.
Tutorial6.Access*Link*.QueuingEntity.QueuingDiscipline.MaxNumberOfMessages = 11 # needed to
     accept ack + initial window
```

```
# Topology settings:
Tutorial6.NumberOfClients = 1          # only one client is needed for this task
Tutorial6.NumberOfServers = 1          # only one server is needed for this task

# Task specific settings:
Tutorial6.DefaultCongestionControl = reno              # sets the congestion control algorithm
     all initiated stacks shall use via sysctl. For the shown results of this task we used "
    reno" and "cubic".
Tutorial6.CentralLinkClientToServer.QueuingEntity.QueuingDiscipline = ikr.protocolsupport.
    algorithms.queuingDisciplines.RedIpQDisc             # We want to use the RedIpQDisc for
    this link's queue


# Simulation control settings:
Tutorial6.TransientDuration = 0        # due to the fact that we do not need statistical
    results we do not need a transient (warm up) phase
Tutorial6.BatchDuration = 3            # by setting the batch duration to 3 seconds we get a
    total simulation time of 3s * 10 = 30s

# Possible generic outputs:
Tutorial6.L4Connection*.LogSocketState = true            # sets up a congestion window logger
    which writes a *cwnd.csv log file
Tutorial6.CentralLinkClientToServer.QueuingEntity.QueuingDiscipline.Tracing = true
                                       # logs the queue's status
Tutorial6.CentralLinkClientToServer.QueuingEntity.QueuingDiscipline.TraceFile = RedQueueLog.csv
            # sets the name of the queue's log file


# These are all parameters we need to set for this task. There are several other parameters
    needed to run this model which are set by default if we do not set them here.
# After you ran a simulation, you can see all used parameters by simply greping for "query" in
    the *stderr log file:
# command line example: grep query Tutorial6.stderr
```

Listing 6: attached file Problem6.par

5 Version history

2015-06-16 David Wagner

- changes to reflect clarifications in protocolsupport-lib
- reference to VMSimInt paper
- reference to SimTree

2015-02-03 Sebastian Dörner

- initial feature-complete version (supervised by David Wagner)