# Modeling Stateless Transport Protocols in ns-3

Dmitry Ju. Chalyy
P.G. Demidov Yaroslavl State University
Yaroslavl, Russia
chaly@uniyar.ac.ru

*Abstract*—**Development of transport protocols have received a great deal of attention of network research community at several past decades. One of the general directions of such an effort was to improve a congestion control mechanism of the TCP (Transmission Control Protocol), which is tightly bounded with other components of the protocol responsible, for example, for robust delivery of data and loss detection. Such a solid architecture complicates innovations in this area and leads to inefficient or misleading functioning in different network settings.**

**Last decade has shown the emergence of new communication paradigms such as cloud computing, software-defined networks, sensor networks, fog networks etc. Thus, rethinking architecture of the transport protocol can be useful to comply new demands. The standard TCP approach ties the transport connection to its endpoints however approaching network applications in new network settings may demand more flexible and transparent data transfer. For example, in cloud computing architectures, servers can dynamically power on or shutdown and such a behavior must be transparent for clients. This can be difficult or even impossible to achieve if the transport protocol's state is distributed between both sides of the connection. We consider a protocol called the Trickles [1], which is one of the first efforts to migrate all connection state to one endpoint allowing its counterpart to operate without any per-connection state.**

**In this paper we describe the architecture of the model of such a stateless protocol and describe a framework that can be used to model such protocols in ns-3. Another contribution of the paper is an approach based on ideas of literate programming [3] to achieve reproducible results of analysis of network protocols.**

## I. Introduction

Since its appearance in 1980 the Transmission Control Protocol (TCP) [11] was established as a connection-oriented protocol which provides a reliable service with a congestion control in a contrast to the Unified Datagram Protocol (UDP) which does not guarantee delivery. The transport layer connection is a virtual arrangement provided by a synchronization scheme which lies under the hood of the protocol. The protocol itself is a distributed algorithm which changes the values of two sets of variables settled on each of two sides of the connection. This two sets of variables constitute the transport connection state. The transport connection is established by three-way handshake algorithm which synchronizes both endpoints. During data transfer both endpoints must maintain consistent local states. Tight synchronization of local states leads to various deficiencies which are critical for modern network applications and can lead to a number of vulnerabilities involving misbehaving endpoints, connection hijacking, connection disruption, half-open connection denial-of-service attacks etc.

Another motivation to consider stateless protocols are modern information systems which impose special demands on networking. One of the such a prominent applications are cloud computing and its enabler technology, software-defined networks. Processing units in cloud environments are highly flexible, that is can be stopped, started or moved across the cloud dynamically. This can lead to loss of the local state of the stopped endpoint, so the transport connection becomes half-open. This scenario can be considered from the other perspective. Processing unit in a cloud environment can provide a service to a large number of mobile clients each of which can dynamically connect to various hotspots obtaining different IP address, so the mobile unit can loss its local state or the state can become obsolete. This leads to a large number half-open connection at processing unit side and can overwhelm it. Such a behavior is similar also in a DoS-attack case where an attacker opens a lot of half-open connections at server side exhausting its resources. Tightly synchronized endpoints cause impossibility of finer-granular load balancing in cloud environments different from connection-level granularity. Stateless transport protocols have shown its advantages in load balancing applications using anycast communication in software-defined networks [5]. Considered examples illustrate drawbacks of existing TCP architecture.

Moving the state to one side of the connection can break such a tied synchronization and will promote more effective use cases of transport service. One of such efforts to solve this problem is the Trickles protocol [1] which promotes all the connection state to a client side leaving the server stateless. So, the server does not store any connection information, thus, it has no drawbacks considered earlier. In next section we consider the design of the stateless transport protocol and revise challenges in such a protocol development. In the section III we consider our implementation of stateless protocols in ns-3. The section IV presents a reproducible methodology for experimental network research and experimental results.

## II. Stateless Protocol Design

In a context of transport layer interaction, modern networks can be seen as a graph where nodes are computing systems and edges represent transport layer connections. Some of these nodes are high degree nodes so they have a lot of interaction with the rest of network. We will call such nodes as servers. Servers are sources of data traffic and transmit the valuable information to the other nodes in a network that we will call clients. We would like to focus attention on two essential differencies between servers and clients: first, servers have many established connections and second, they mostly transmit data to clients. We will use such an abstract but still useful client-server model in our discussion.

Transport layer connection between a server and a client using TCP protocol is established via three-way handshake

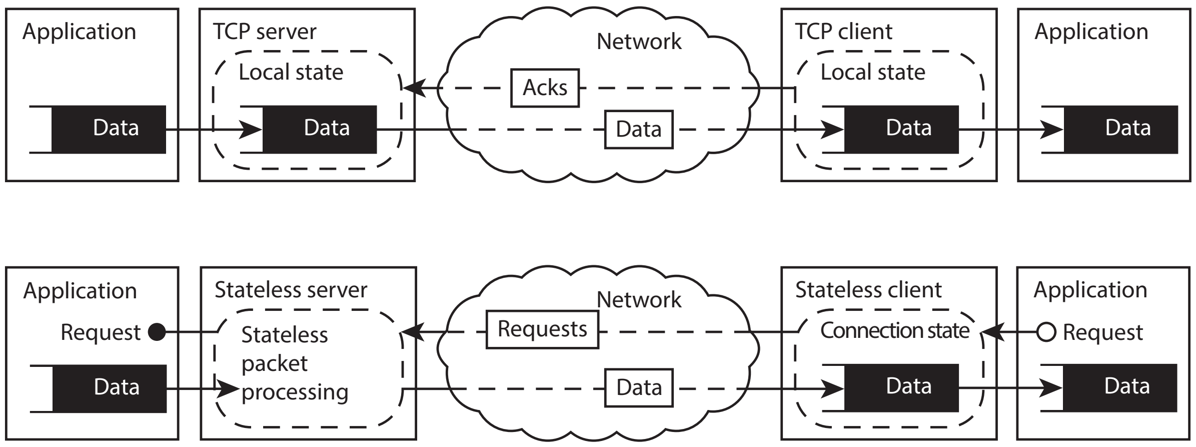

Fig. 1. TCP-like distributed state transport protocol (top); Stateless transport protocol (bottom)

algorithm [11]. Every byte in a data stream is numbered and the reception of data is acknowledged by a client. The role of the server side is to push data to the client using the sliding window protocol and a congestion control algorithm trying not to overwhelm the client and a network. The client role is rather passive; the client must acknowledge received data as soon as possible using Nagle algorithm. The big picture is shown on Fig. 1. The client and the server maintain distributed synchronized state variables in transmission control block (TCB) structures. Packets contain data and a signal (e.g. acknowledgements) information. The algorithm is distributed between the client and the server and it changes the state of the connection upon receiving a packet or when an event (e.g. retransmit timeout) occurs.

Stateless protocol model leads to substantial changes in the transport protocol architecture. Instead of pushing data to a client, the server waits for a data request. This request is originally initiated by an application at the client side. The application could request any size of data. The purpose of the stateless protocol is to cut the request into small chunks and pass these smaller requests to the server. Stateless protocol at the server side just passes the requests to server application which fulfills them with data. In this setting a server is a stateless part of the connection thus it must recover necessary state to process requests. This is done by using self-describing request packets which contain encapsulated state data. Successful delivery of requested data to the client plays the role of the acknowledgement. Despite the statelessness the server side of the protocol is responsible for making congestion control decisions as in case of TCP.

As an example of a such a stateless protocol we consider the Trickles protocol [1]. The Trickles connection exchanges data in a parallel manner using two key abstractions: a trickles and a continuation. Since the server does not tracks the state, the client must encapsulate relevant parts of the state to the request packet. Server considers received state, updates it and attaches to response packets. The piggybacked state is called a continuation since it contains all necessary information for the server to resume processing data stream later (see Fig. 2). After receiving response packets client can fuse several continuations to obtain a new global connection state. The client side is stateful providing reliable delivery of requests and data to the stateless server. The request-continuation model is not only applicable to transport layer but stateless approach encourages communication services developers to create stateless applications in a similar manner. For example, many web services using HTTP protocol can be considered in such a way.

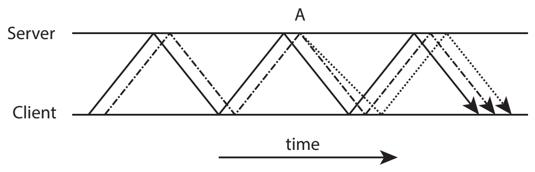

Fig. 2. A sample Trickles connection. Each line pattern corresponds to a different trickle

Transport protocol must be efficient in terms of performance. So each Trickles connection maintain multiple trickles each encapsulating partial connection state. These parallel states are the source of complexity of stateless protocols. The Fig. 2 shows the typical data exchange in transport connection. Series of related packets that trace a series of state transitions forms a trickle. A trickle is a control and data flow management unit. A server can terminate a trickle decreasing network congestion or split trickles (as shown at time $A$ on Fig. 2) trying to increase network usage.

The original Trickles protocol [1] mimics TCP Reno [8] congestion control algorithm as close as possible. The congestion control algorithm operates at the stateless server side which deals with each trickles independently. The original TCP Reno algorithm ensures that there are $cwnd$ in-flight packets in a given moment of time, so the Trickles server side tries to limit the number of simultaneous trickles in the network to $cwnd$. When a packet arrives to a server side it can permit the application to send one packet in response, which is a most common case, *continuing* the trickle; if packets were lost due to a network congestion the server may *terminate* the current trickle by not permitting the response. The server may also *split* the current trickle into $k$ response packets beginning $k-1$ new trickles.

The server statelessly performs these decisions trying to follow congestion control mechanisms in TCP Reno [8]: slow start/congestion avoidance, fast recovery, fast retransmit. Each packet has a unique sequence number $k$ and contain the following information as a partial connection state to support these mechanisms:

- $TCPBase$ — sequence number of the first packet in slow start phase;
- $startCwnd$ — initial value of $cwnd$ at $TCPBase$ packet;
- $ssthresh$ — threshold value which determines whether the protocol should do slow start or congestion avoidance;
- $SACK$ — selective acknowledgements (SACKs) [9] which are the compact representation of losses the client has incurred. The Trickles protocol relies heavily on SACKs to detect losses.

Let us briefly describe the Trickles simulation of TCP Reno congestion control.

*Slow start and congestion avoidance.* The original TCP Reno increases $cwnd$ by one per packet acknowledgement in slow start; $cwnd$ increases by one per window in congestion avoidance. Each request with sequence number $k$ is processed using the following algorithm:

1) Let $CwndDelta = TCPCwnd(k) - TCPCwnd(k-1)$. Here $TCPCwnd(k)$ is a closed-form solution of TCP simulation from [1].
2) Continue the original trickle and split $CwndDelta$ times starting from sequence number $k + TCPCwnd(k-1)$.

In particular, the $TCPCwnd(k)$ function is defined as follows:

$$TCPCwnd(k) =$$
$$= \begin{cases} startCwnd + (k - TCPBase) & \text{if } k < A \\ ssthresh & \text{if } A \leqslant k < A + ssthresh \\ F(k-A) & \text{if } A + ssthresh \leqslant k \end{cases}$$

where

$$A = ssthresh - startCwnd + TCPBase$$

and $F(N)$ is the largest integer less than the positive value of $x$ that is a zero of

$$\frac{(x-1)x - (ssthresh-1)ssthresh}{2} - N$$

The algorithm takes the assumption that no packets are lost. This assumption can easily be checked looking at the request packet SACK block. If any losses occur the overall transfer process can be partitioned at the loss positions into multiple loss-free epochs. Thus, $TCPCwnd(k)$ is valid within each individual epoch. Free parameters of the formula adapt it to each epoch. Upon recovery from loss the Trickles recovery algorithm updates values of $TCPBase$, $startCwnd$ and $ssthresh$.

*Fast retransmit/recovery.* If the client receives an out-of-order packet it put the trickles on hold. If three out-of-order packets are received the client transmits deferred requests. The SACK block of each request contains information about losses. For a request with sequence number $k$ the Trickles performs following operations:

1) Let $firstLoss$ is a number of the first lost packet. It is obtained from SACK block.
2) If $k$ is the number of the packet right after a run of losses, the Trickles retransmits the lost packets. Maximum number of retransmitted packets can be tuned using $burstLimit$ parameter. Losses beyond $burstLimit$ are handled via retransmit time-out.
3) Let $lossOffset = k - firstLoss$ and $cwndAtLoss = TCPCwnd(firstLoss - 1)$.
4) Compute the estimate of number of packets in the network
$$numInFlight = cwndAtLoss - 1.$$
5) New value of $cwnd$ will be
$$newCwnd = \lfloor numInFlight/2 \rfloor.$$
6) If $cwndAtLoss - lossOffset + 1 \leqslant newCwnd$, continue the trickle and tag it as *recovery*. Otherwise terminate the trickle.

All survived trickles have a special *recovery* tag. This tag helps the client to switch from lossy fast recovery epoch to a new loss free epoch. If all losses have been recovered using fast retransmit, the client updates new loss-free epoch variables: $TCPBase$ points to recovery point, $ssthresh = startCwnd = newCwnd.$

*Retransmit timeout.* The retransmit time-out occurs at the client. The client retransmits the last request which has special *RTO* tag. The server executes the following steps:

1) Let $firstLoss$ is a number of the first lost packet. It is obtained from SACK block.
2) Update $ssthresh = \lfloor TCPCwnd(firstLoss - 1)/2 \rfloor$.
3) Split $startCwnd - 1$ times and retransmit lost packets.

The client must remove *RTO* tag from outgoing packet if its SACK block does not contain any loss information and must update initial conditions of the next loss-free epoch.

As we can see the statelessness of the server leads to a non-trivial congestion control mechanism. Modeling and analysis of such a mechanism in a well-defined simulation environment is a necessary condition for capturing essential performance properties of the protocol.

## III. Implementation Framework

The main goal of our implementation is to make the framework as abstract as possible for further advancing research of stateless transport protocols. We consider models of ns-3 protocols as an example of such an implementation. The models of particular TCP versions are derived from abstract classes so it is possible to derive original ns-3 models of TCP without digging deep into details. Stateless protocols have a quite different from the standard TCP architecture. Thus, we have to develop abstract models which hide low level details of sockets and network layer interfaces.

### A. Applications

The stateless approach to the transport layer encourages the use of stateless applications. To proceed to the study of the family of stateless protocols we decided to implement a version of an easiest application. This application consists of a client `ns3::TricklesSink` which can request data from a socket at a given rate and a server `ns3::TricklesServer` which fulfills incoming requests right after the receipt. Applications are accompanied with corresponding helper classes for building executable ns-3 experiment setups easier.

### B. Sockets

The ns-3 provides a model of Berkeley-like socket interface to a transport layer. The original Trickles protocol contributions [1], [14] introduced a specific socket model called *minisockets*. Minisockets are descriptors that created in on-demand manner when an event occurs. Introduction of the minisockets model to ns-3 seemed to be a very complex task. So we decided to use the existing Berkeley-like socket interface but use a different semantics of application calls in a stateless case.

The standard TCP connection initiated using either a *Connect*, or a *Listen* call. Application use the *Connect* call to initiate connection to the remote side and act as a listening server using *Listen* call. Trickles protocol does not need connection establishment process, so we use these calls as just a notification of the endpoint role (will it behave as a server or a client) to the local socket.

Application can send data to a socket using a *Send* call. In the stateless application setting this call is used only by a server side to send data in response to a client's request.

The most important call in a stateless setting is the *Receive(maxSize, flags)* call. The server side uses this call to receive incoming data requests from a local socket. The socket notifies the server application of incoming request right after its receipt. The client side can execute two different kinds of the *Receive* call. The first one is obvious — if there is data in the socket the application can read this data from. The second kind of call is specific to the client only. It places a request of a new portion of data from the server. The request is queued in the socket and the protocol will cut it into smaller requests and send them to a server. These two kinds of *Receive* call are differentiated by the value of parameter *flags* which exists in the implementation of the *Receive* call in ns-3.

### C. Packets

The network communication between two Trickles endpoints is a stream of packets. The original Trickles protocol proposal [1] states that the Trickles protocol encapsulates transport continuation within the TCP packet which is extended by means of variable options [11]. It is not very convenient to redefine the existing TCP Header class (`ns3::TcpHeader`), so we decided to introduce a new header class `ns3::TricklesHeader` which encapsulates all relevant information to a generic stateless protocol, for example, packet number, SACK options, request size, recovery tag and a timestamp. One of the main topics in transport protocols research is a congestion control mechanism analysis which allows us to measure the performance of the protocol. Such mechanisms may be of a different nature and may use different sets of control variables. For the reference implementation of the stateless protocol which mimics TCP Reno [1] we decided to introduce one more header type called `ns3::TricklesShiehHeader` which carry the congestion control data like $TCPBase$, $startCwnd$ and $ssthresh$. We beleive that many more congestion control mechanisms (for example, based on the rate estimation [4]) can be proposed in a stateless setting so hard coding congestion control into a generic header is not a solution.

Every packet which is involved in a networked interaction contains three headers. The `ns3::TcpHeader` on top of the packet which contains necessary socket information and a starting sequence number of data which must be get from the server. Below is a `ns3::TricklesHeader` which contains generic stateless protocol data and if the model of the reference protocol is used then under this header the `ns3::TricklesShiehHeader` is settled.

### D. Trickles implementation

Let us consider the implementation details of the Trickles protocol model. The overall idea of the implementation is to follow TCP implementation in ns-3. The hierarchy of classes implementing the Trickles protocol is shown on Fig. 3. The central class in the hierarchy is the `ns3::TricklesL4Protocol` which models transport layer protocols in the TCP/IP stack. Each node in the network contain a unique object of this class which is responsible for passing packets to a network layer and demultiplexing packets to sockets The object contains references to existing Trickles sockets which are the objects of class `ns3:TricklesSocketBase`. This class models a behaviour of a Trickles socket interpreting application calls using semantics we have defined earlier in section III-B. The `ns3::TricklesSocketBase` class provides a service which is common for all stateless protocols. It manages queues of sent requests and arrived data. It notifies the application if a new data or a request have arrived. The important part of the socket model is a round trip time estimation which is made using timestamps and RTTM mechanisms [10]. The `ns3::TricklesSocketBase` provides a couple of virtual methods which should be overloaded when implementing the specific stateless protocol:

- The method `::NewRequest` is called whenever socket receives a request on a new portion of data from the client application;
- `::ProcessTricklesPacket` is called if a new packet is received from a network.

The `ns3::TricklesShieh` class is derived from the abstract `ns3::TricklesSocketBase` class. That class implements the congestion control algorithm that we have described in section II. So the `ns3::TricklesShieh` is an implementation of the specific stateless transport protocol defined in [1].

We have updated the source code for `ns3::InternetStackHelper` class which is responsible

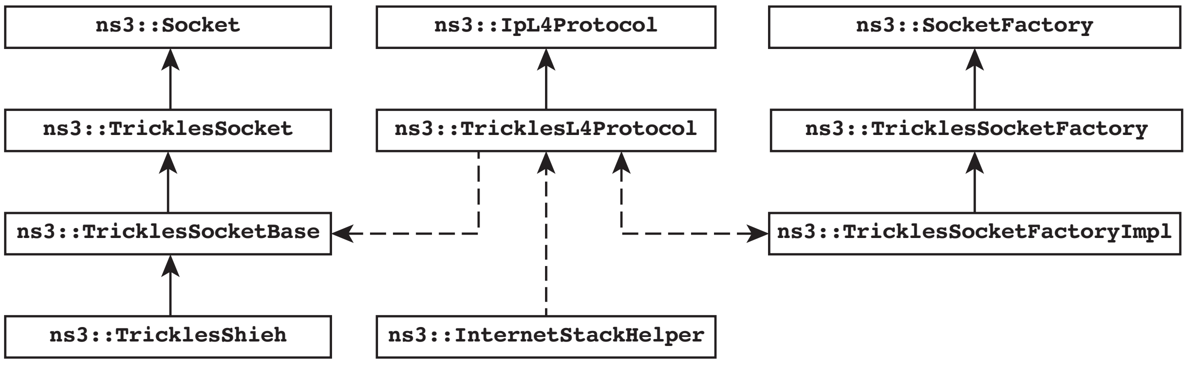

Fig. 3. Class hierarchy of the Trickles model in ns-3

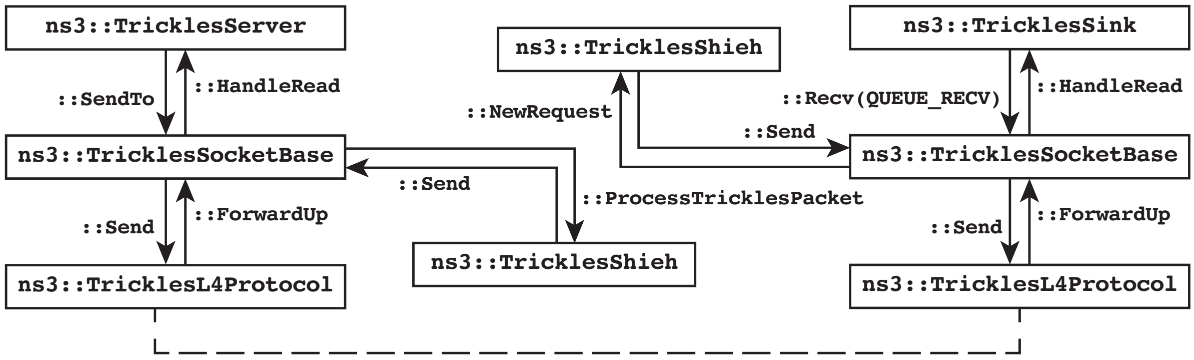

Fig. 4. Flowcharts of an arbitrary packet through the ns-3 protocol stack

for creation of TCP/IP stack on a given node to add the Trickles protocol to the stack.

The flow of a single data portion is shown on Fig. 4. Let us follow the picture from right to left. The instance of the `ns3::TricklesSink` class plays role of the client and initiates data transfer by requesting a new portion of data. This is done by executing `::Recv` call on the instance of the `ns3::TricklesSocketBase` class with the parameter *flags* equals to `QUEUE_RECV`. This means that the client requests a new data which is expected right after all previous requests be fulfilled. The `ns3::TricklesShieh` is the actual implementation that handles congestion control by default in a current version so the processing besides the generic one is done using method `::NewRequest`. When the congestion control allows to emit new request the method `::Send` is triggered. The relevant headers are attached to the packet and the packet is sent using `ns3::TricklesL4Protocol` instance.

The server side receives the packet with incoming data request in `ns3::TricklesL4Protocol` object which forwards the packet to the specific instance on `ns3::TricklesSocketBase` based on destination port information in `ns3::TcpHeader`. The actual implementation of the congestion control algorithm, the `ns3::TricklesShieh` object processes the packet, updates headers to form a correct continuation and then methods of `ns3::TricklesSocketBase` immediately notify the `ns3::TricklesServer` application that the request is received. The application reads the packet from socket, adds the data to the packet and sends it back to the socket which passes the packet to the network via `ns3::TricklesL4Protocol`.

The reception of continuation with data at the client side is almost the same as the reception of the request at the server side. The data is placed to the data queue and the `ns3::TricklesSocketBase` notifies `ns3::TricklesSink` that new data arrived via handler `::HandleRead`.

## *E. Issues*

It is hard to get the reference for comparing developed model of the Trickles protocol even though there is an implementation of the protocol in Linux kernel [14]. There are several reasons: the implementation itself can not be a model of the protocol since it has a lot of subtle low-level details which are not the part of the Trickles reference contribution [1]; the implementation does not has any documentation; it is hard to map the behaviour of the protocol in a real setting to what we achieve under simulation. So we decided to create own experiment and make sure that behaviour of the model is the same as expected.

## IV. Experiments

### A. Methodology

Transport protocols are the important part of the Internet infrastructure, so the protocol research must be reproducible. Reproducibility does not guarantee the correctness of the results but can make sure that the findings can be replicated by an independent researcher. Nevertheless, reproducibility is not a common practice in computer science [20] but achieved a considerable amount of attention in recent years [7]. Most research in protocol community is done using simulations as a source of raw data which is explored using statistics and exploratory data analysis. We think that the approach based on data science criteria of reproducible research [2] will be fruitful in achieving reproducible results. These criteria states that every research consists of the following:

- *Raw data.* The process of obtaining raw data must be explicit and be accessible. However, some research is unique and can not provide means to obtain raw data on demand.
- *Clean data.* Raw data can be represented in various formats. Clean data is organized as a table data where each column represents a variable and every row represents an individual measurement.
- *Analytic code.* This code processes clean data and provide answers to scientific problems which are in the center of the research.
- *Presentation code.* The code is used in exploratory data analysis and is used to make visual representations of data and results.

In a context of our research, the developed ns-3 model of a stateless protocol and a code implementing particular experiments are the sources of raw data in our research. The result of execution of experiments are the traces which are text files including time series of events occurred in the network under consideration. The source code of our framework is available for ns-3 (ver. 3.20) as Git repository at [12]. The repository is organized as straightforward as possible: you should put the code to ns-3 source code tree and compile. Our implementation contains several unit tests which validates the build.

Our approach is based on using R [15] as a programming platform for cleaning data, making statistical and exploratory data analysis. The R provides convenient tools for reading and cleaning files as well as a solid statistical computing platform which can be extended with vast amount of packages available at CRAN [16]. The code for reading and cleaning ns-3 trace files is available at repository [13].

The holy grail of reproducible research are reports which presents the results can be compiled automatically from the raw data using robust tools. Donald Knuth proposed a methodology called *literate programming* [3] which promotes the idea that the same program can be compiled both into executable and into documentation. We decided to organize our experiments in a spirit of literate programming paradigm. The key tool in this undertaking is *knitr* package [19] for R. The source file for *knitr* is a simple text file written in Markdown [18] which can contain R code for data processing. *knitr* compiles the source file executing the R code, starting individual experiments, and produce html or LaTeXfile which contains full protocol of the experiment.

### B. Experiments

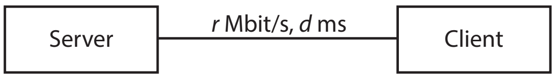

Fig. 5. Performance comparison of the Trickles model to the Reno model in ns-3

For all experiments[1] we use a simple network topology shown on Fig. 5. The topology consists of a single client and a server. The transport interaction between these counterparts in our setting can be managed using the Trickles and the TCP Reno protocols. For the Trickles case we have used `ns3::TricklesSink` application at the client side and the `ns3::TricklesServer` at the server side. For the TCP Reno protocol we use `ns3::PacketSink` at the client side and `ns3::OnOffApplication` at the server side. In both cases we configure the experiments in such a way that applications continuously push data from the server to the client.

The network configuration has several parameters. The link is a point-to-point link with a bandwidth and delay specified individually for a single experiment. The bandwidth $r$ can be 1.0, 2.0 or 3.0 Mbit/s and the delay $b$ can be 10, 55 or 100 ms. We also vary the queue length at the server and the client nodes from 70 packets to 100 packets with step of 5 packets.

Both TCP Reno and Trickles protocols are configured in a similar way. They both start with $cwnd$ equals to 2 packets and $ssthresh$ value equals to 65 packets. The data payload size of each packet is set to 1000 bytes.

The results of the experiments are shown on Fig. 6. The x-axis shows the queue length and the y-axis shows the performance achieved by each protocol in different network settings. We can see that the Trickles shows lower performance in contrast to the TCP Reno as expected since it has a more sophisticated request-response algorithm than TCP. With more queue length the performance of the Trickles protocol becomes close to the Reno results as expected. In general the experimental results confirm the hypothesis that the Trickles performance must be close to the Reno performance.

## V. Conclusion

The stateless protocols is the interesting family of transport layer protocols. The modern applications such as cloud environments, high-load applications, software-defined networks, sensor networks can benefit from stateless protocol architectural advantages and absence of hard synchronization between endpoints. Nevertheless each protocol must be thoroughly investigated before its adoption in real networks. The main methodology of transport protocol research is simulation and experimental analysis. Building an executable model of a stateless protocol in the ns-3 simulator is a first step for

[1]the source code of the experiment and its *knitr* protocol is available at [12]

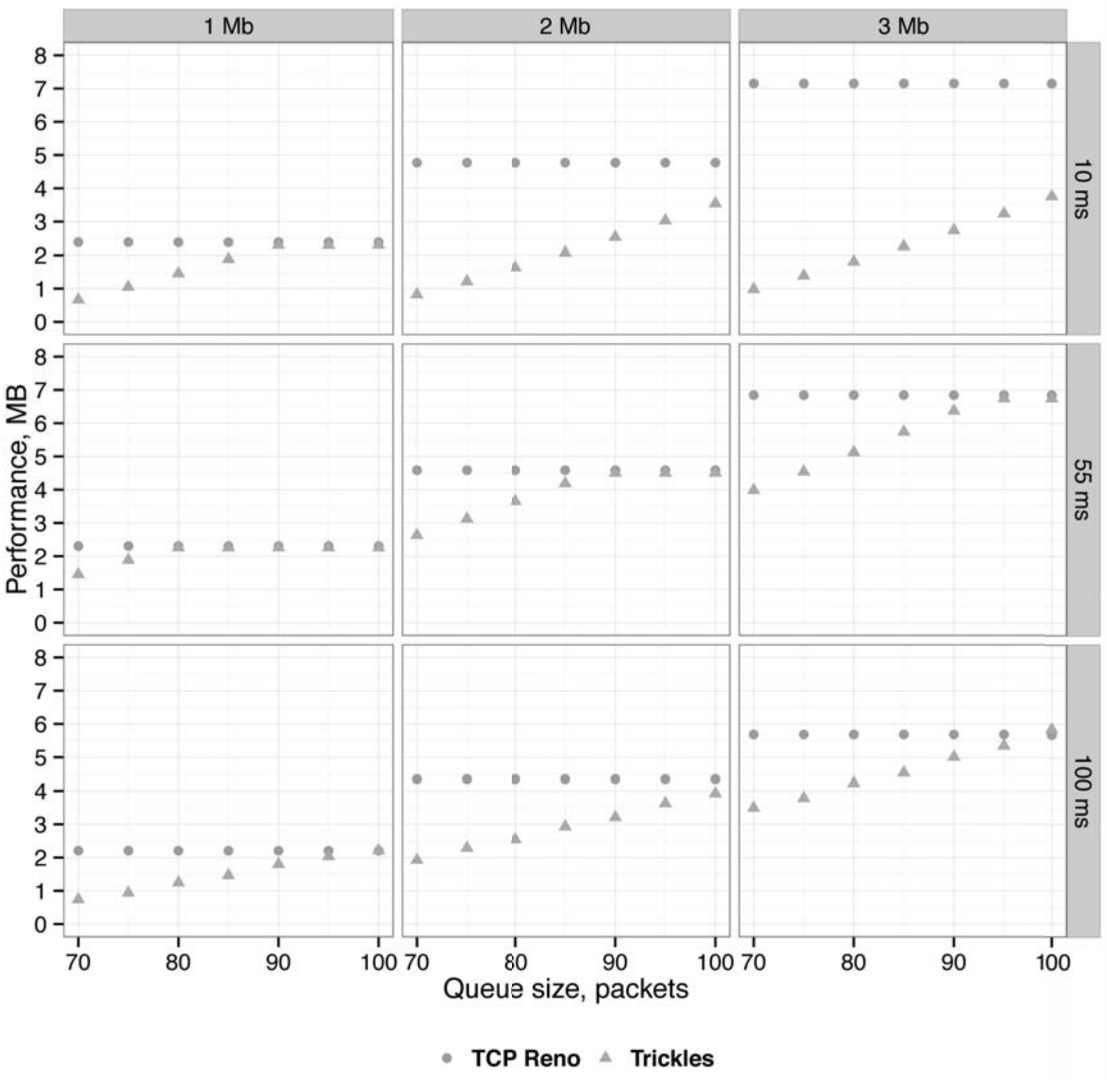

Fig. 6. Performance comparison of the Trickles model to the Reno model in ns-3

encouraging research in this direction. The paper presents such a model which source code is available at [12]. Of course the model must be further investigated and reviewed by the protocol research community. More to be done in the experimental analysis of the stateless protocols in various network settings. The reproducibility of such a research is essential. The paper advocates a methodology based on the ideas of literate programming and using R statistical framework to achieve high level of reproducibility. The tools which provide interface between ns-3 and R are available at [13] and must be further developed.

## ACKNOWLEDGMENT

The authors would like to appreciate the Russian Fund for Basic Research (RFBR) for partial support under project 14-01-31539.

The research was conducted at P.G. Demidov Yaroslavl State University using financial support of the Ministry of Education and Science of the Russian Federation under contract ID RFMEFI57414X0036.

## REFERENCES

[1] A. Shieh, A.C. Myers, E.G. Sirer, "A Stateless Approach to Connection-Oriented Protocols", *ACM Trans. Comput. Syst."*, Vol. 26, No 3, September, 2008, 50 p.

[2] R.D. Peng, "Reproducible research in computational science", *Science*, Vol. 334, No 6060, 2011, pp. 1226-1227.

[3] , D. Knuth, "Literate Programming", *The Computer Journal*, Vol. 27, 1984, pp. 97-111.

[4] I.V. Alekseev, V.A. Sokolov, "ARTCP: Efficient Algorithm for Transport Protocol for Packet Switched Networks", *Proc. of Parallel Computing Technologies (PaCT-2001)*, September, 2001, pp. 159-174.

[5] M.A. Nikitinskiy, I.V. Alekseev, "A stateless transport protocol in software defined networks", *Science and Technology Conference (Modern Networking Technologies) (MoNeTeC)*, October, 2014, pp. 108-114.

[6] A.C. Snoeren, D.G. Andersen, H. Balakrishnan, ”Fine-Grained Failover Using Connection Migration”, *Proc. of 3rd USENIX USITS*, March, 2001.

[7] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, N. McKeown, ”Reproducible Network Experiments Using Container-Based Emulation”, *Proc. of ACM CoNEXT-12*, December, 2012, pp. 253-264.

[8] M. Allman, V. Paxson, W. Stevens, ”RFC 2581: TCP Congestion Control”, 1999.

[9] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, ”RFC 2018: TCP Selective Acknowledgement Option”, 1996.

[10] V. Jacobson, R. Braden, D. Borman, ”RFC 1323: TCP Extensions for High Performance”, 1992.

[11] DARPA, ”RFC 793: Transmission Control Protocol”, 1981.

[12] Dmitry Ju. Chalyy, ”Git repository of stateless transport protocol models for ns-3”, *URL: https://github.com/dchaly/stateless*, Accessed: 2015-02-28.

[13] Dmirty Ju. Chalyy, ”Git repository of ns-3 trace analysis code in R”, *URL: https://github.com/dchaly/ns3r*, Accessed: 2015-02-28.

[14] Alan Shieh, ”Trickles source code for Linux Kernel”, *URL: http://www.cs.cornell.edu/w8/ ashieh/trickles-release/*, Accessed: 2015-02-28.

[15] The R Project for statistical computing, *URL: http://www.r-project.org*, Accessed: 2015-02-28.

[16] The comprehensive R archive network”, *URL: http://cran.r-project.org*, Accessed: 2015-02-28.

[17] ns-3 network simulator, *URL: http://www.nsnam.org*, Accessed: 2015-02-28.

[18] Daring Fireball: Markdown”, *URL: http://daringfireball.net/projects/markdown/*, Accessed: 2015-02-28.

[19] knitr: elegant, flexible and fast dynamic report generation with R, *URL: http://yihui.name/knitr/*, Accessed: 2015-02-28.

[20] C. Collberg, T. Proebsting, G. Moraila, A. Shankaran, Z. Shi, A.M. Warren, ”Measuring Reproducibility in Computer Systems Research”, *URL:http://reproducibility.cs.arizona.edu/v1/tr.pdf*, Accessed: 2015-02-28.