Fed4FIRE+ Experiment Report

Joint Experimentation of Modern Internet Application Protocols with SDN (Go-Quick)

Date of preparation of your proposal:	13/02/2017
Version number *(optional):*	
Your organisation name:	Eight Bells Ltd
Your organisation address:	Agion Omologiton 15, Nicosia 1080, Cyprus
Name of the coordinating person:	Dr. Ioannis Giannoulakis
Coordinator telephone number:	+306997060964
Coordinator email:	giannoul@8bellsresearch.com

Contractual Date of Delivery:	30/09/2017
Editor:	Dr. Ioannis Giannoulakis
Authors:	Dr. Ioannis Giannoulakis, Dr. Jordi Ferrer Riera, Mr. Michalis Tzifas
Distribution / Type:	Public (PU) / Report (R)
Version:	1.2
Total Number of Pages:	51
File:	Go-Quick Final Report

Table of Contents

Table of Figures

Section A Project Summary

This section provides an executive summary of the experiment objectives, implementation and main results. Remark: The information in this section will be used in public documents and reports by the Fed4FIRE consortium. The length of this section is restricted to 1 page.

Go-Quick proposal aims to evaluate the performance of QUIC protocol, as compared to HTTP and SPDY, through deploying virtual overlay networks on Fed4FIRE+ OpenFlow SDN-enabled infrastructure. Since sophisticated congestion avoidance and packet error correction mechanisms are used in all protocols, the actual network scenarios of the Go-Quick experiments determine which protocol performs best in each case.

By emulating various network conditions, performance metrics, like web page load time, throughput is assessed for the three protocols HTTP, SPDY, and QUIC. These metrics are used to indicate the cross-layer approach that is the most efficient, under a set of characteristic network conditions.

Extensive experiments indicate that the QUIC protocol, although not in an optimized and “mature” publicly available software version, performs well in case of congested networks, where losses and delays are not negligible. On the other hand, HTTP and HTTP/2 (the successor of SPDY) are favoured by large companies like Google and have efficiently implemented clients and servers. These protocols are remarkably fast, especially when network conditions are relatively good.

Section B Detailed Description

B.1 Concept, Objectives, Set-up and Background

There is no page limit for this section as you are invited to describe the concept, objectives and setup in as much detail as you wish to do. Please also include graphs and figures were needed.

B.1.1 Concept & objectives

Go-Quick goal is to evaluate the performance of QUIC, as compared to HTTP and SPDY, using the powerful and flexible virtual machines and/or switches provided by the Fed4FIRE+ OpenFlow SDN-enabled infrastructure of the i2CAT OFELIA testbed, in Spain. Indeed, since sophisticated congestion avoidance and packet error correction mechanisms are used in all candidate protocols, the actual network scenarios applied during the Go-Quick experiments determine which protocol performs best in each case, so as to encourage the adoption of Web 2.0 technologies.

All three candidate protocols are trying to tackle with diverse problems appearing in upper layer protocol behavior and thus are exhibiting remarkable differences in their nature. For example, HTTP is opening several TCP/IP channels between the web server and the user, SPDY tries to deliver all page content using only one TCP channel, while the QUIC protocol tries to get rid of the time-consuming quality assurance (connection oriented) mechanism of the TCP protocol and thus, chooses to build its functionality on the UDP protocol.

By emulating various network conditions, performance metrics, like web page load time, throughput is assessed for the three protocols HTTP, SPDY, and QUIC. These metrics are used to indicate the cross-layer approach that is the most efficient, under a set of characteristic network conditions.

Web service providers and operators need to enhance their networks and effectively manage transport protocols and other network parameters so as to deliver top quality of experience to their subscribers. So far, QUIC has managed to provide very good results. According to Google [1], QUIC has helped to realize a 3% improvement in the mean page load times on Google Search. That may not seem high but one needs to recall that Google search is already about as optimized as possible. Other sites, and especially latency-heavy web apps, will likely see better improvements. Users who connect to YouTube over QUIC report about 30% fewer re-buffers when watching videos and because of QUIC's improved congestion control and loss recover over UDP, users on some of the slowest connection also see improved page load times.

This improved speed and optimization comes at the perfect time. Applications, videos and games have continued to dominate Internet usage. More specific, video has skyrocketed over the past years with Gartner Research reporting an estimated 59% increase in mobile video traffic in 2015 [2]. Nevertheless, solutions must also enable operators to manage and optimize future traffic patterns and also web service providers must have access to an easy to implement framework that allows them to manage new types of content and protocols. Go-Quick proposes an experiment that will consider a range of network conditions in order to provide quantitative results on the performance of some key cross-layer protocols and the experimental outcomes will provide useful insights for the most appropriate network regimes that certain transport protocols can operate. The results can thus be further leveraged by network operators and Content Service Providers (CSPs) through the distribution channels of 8BELLS.

B.1.2 Set-up of the experiment

Hypertext Transfer Protocol (HTTP) is one of the most widely used protocols, responsible for delivering webpages, videos and web applications to billions of devices of all sizes, and from desktop computers

to smartphones. Fueled by the explosive growth of video traffic and HTTP infrastructure (e.g., Content Delivery Networks (CDNs), proxies, web caches), HTTP has become the de-facto protocol for deploying new services and applications.

However, from the publication of HTTP/1.1, websites and Internet traffic have changed dramatically and at the same time, users connect mostly through mobile devices instead of using fixed access. Therefore, many researchers consider HTTP the narrow waist of the future Internet [3]. As a result, web service providers (e.g., Google, Akamai, etc.) and academics worldwide investigate techniques to address inefficiencies of HTTP/1.1.

More specifically, in 2009, Google launched a new web transfer protocol, called SPDY [4] (20-25% of all Internet traffic passes through its servers and Google Chrome browser has 40% of market share [5]). Apart from Chrome browser and Google servers, SPDY is also launched on popular sites like Facebook and Twitter and it is supported from latest versions of Firefox and Internet Explorer web browsers. The limitation of a HTTP client that can fetch only one resource at time is solved in SPDY through multiplexing HTTP requests on a single TCP socket; SPDY also supports a client-server 'push-based' communication model, HTTP header compression and traffic prioritization among parallel requests.

In the case of SPDY, the shortcomings and limitations come from the Transmission Control Protocol (TCP) that it is based upon. SPDY relies on only one TCP socket to deliver resources to the client and thus, a single packet loss in the underlying TCP connection triggers a congestion window decrease for all the multiplexed SPDY streams.

Moreover, an out-of-order packet delivery from TCP leads to Head of Line (HOL) blocking for all SPDY streams multiplexed on the TCP connection. Finally, since SPDY works over TCP, start-up latency is at least one Round-Trip-Time (RTT) and may reach three RTT if SSL/TLS encryption is adopted. For the above reasons, Google proposed a new protocol called QUIC (Quick UDP Internet Connection) [6] in 2013, which adopts UDP (User Datagram Protocol) for the transport layer instead of TCP. Having already deployed QUIC over its data servers (like the ones that power YouTube) and to Chrome browser, Google plans to switch one fourth from total Internet traffic from HTTP over TCP to QUIC over UDP [7], [8].

Go-Quick proposal focuses on investigating and evaluating the performance of QUIC in comparison with HTTP and SPDY, under various networking conditions. For this purpose, the Fed4FIRE+ experimentation infrastructure will be used for the deployment of virtual network topologies over the physical infrastructure. This will pave the way for Go-Quick to quantify the improvements introduced by QUIC, as compared to existing protocols. The comparative analysis of QUIC, SPDY and HTTP will require understanding sophisticated methods for handling congestion control and packet error correction, which depend heavily on network parameters [9], [10], [11] like e.g., available bandwidth, RRT, packet error rate, bandwidth delay-product, etc.

Go-Quick makes use of the Fed4FIRE+ SDN testbed, namely the OFELIA island. The OpenFlow Aggregate Manager (OFAM) of OFELIA will allow the deployment of virtual overlay networks on the physical infrastructure for easily shaping the data traffic, according to the network scenarios of the experiments. This overall design governing the experiments has to take into consideration all relevant developed technologies of 8BELLS and Fed4FIRE+, thus maximizing the usage of the available project resources and adding value to those outcomes.

OpenFlow SDN-enabled switches will allow data traffic routing based on the latency of the physical links and queue assignment (either classless or classful) per application that are expected to have severe impact on TCP-based protocols (i.e., HTTP, SPDY). The congestion avoidance and packet error

correction mechanisms of QUIC protocol are expected to be also heavily influenced from the above settings, as well as the dynamic allocation of bandwidth and data traffic prioritization.

The OFELIA testbed provides a powerful island of VMs that can be used to host the various web server and client flavors as well as the necessary machines emulating the various network congestion cases.

According to the initial plans, detailed design and configuration of the planned experiments had to take place. The Chromium browser code-base4 [11] which incorporates the server and the client with the QUIC protocol had to be compiled over a Linux machine running Ubuntu. For SPDY, the Google implementation of HTTP/2.0 nghttp2 [12] was to be applied, while for the HTTP/1.1 over SSL/TLS an Apache/2.4.10 server with TLS 1.2 was to be employed. The candidate key performance metrics under consideration might be Goodput, Channel Utilization, and Page Load Time, and potentially others to be identified at that stage.

Deviations from the initial plans were expected and actually happened, as newer versions of the necessary software, for running the application protocols and/or testing them, made the scene while the old ones were becoming deprecated. Indeed, the overall experimentation design and setup was a real challenge in order to be able the various (HTTP, SPDY, QUIC) server and client software to be properly installed on the remote virtual machines.

For instance, automating the client – server interaction requires command line tools and appropriate scripting development, which was not always easy to be done. In some cases, the actual servers, available at the various Google sites, or graphical interface implementations had to be used, for most representative results or because of the lack of their command line counterparts. Furthermore, due to the huge amount of space the full compilation of the SPDY source code required (approx. 100 GB), a partial compilation process had to be applied to reduce the overall size bellow to 1 GB. Apart from this, newer versions for Apache server were available and had to be customized upon our testbed. Added to this, SPDY stopped being supported by Chromium (i.e., by Google), in favor of the HTTP/2 protocol [13].

B.1.2.1 Topology

The topology finally selected for the experiments consists of one web server machine, one web client machine and the intermediate network which has two routers[1], all in series. All machines have access to the Internet through the second interface of the web server machine. Indeed, the latter is necessary for easy installation and fine tuning of the software packages being required. Furthermore, in some cases, the actual Google web servers had to be used and thus, the connection to the outer world was necessary. The network topology of the VMs comprising the basic testbed is depicted in Figure 1.

The Virtual Machines used to simulate each computer should be powerful enough to support the needs of the software being hosted and eliminate computational power related bottlenecks. Furthermore, the space required for compilation/installation of the necessary software was not of small size, especially in case of the QUIC software. More specifically, the machines being allocated and used had the following characteristics:

[1] Although physical (NEC IP8800) SDN switches were available, having direct access through the SDN controller might have indirectly interfered with other experiments, created traffic loops or other conditions. For this scenario we preferred to deploy an overlay virtual topology to simulate their functionalities.

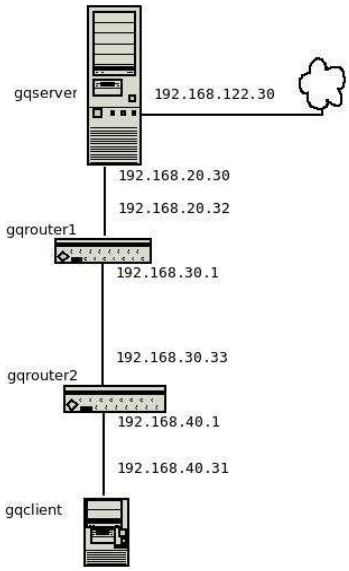

Figure 1: The final topology of the testbed

B.1.2.2 The VM hardware

VM specifications: 4-core CPU, 4 GB RAM for gqserver, gqclient, 2-core CPU, 2 GB RAM for router, with 2 available networks, one external and one internal of gigabit speed. More specifically:

- gqserver: The machine serving QUIC, HTTPS and HTTP2, (SPDY) protocols. VM specifications: 4-core, 4 GB RAM, 1 network interfaces.
- gqrouter1, gqrouter2: Traffic shaping machine to simulate network speed, errors and latency. VM specifications : 2-core, 2GB RAM, 2 network interfaces.
- gqclient : Client for QUIC, HTTPS and HTTP2 protocols, with simple GUI for Chrome client. VM specifications : 4-core CPU, 4 GB RAM, 1 network interface.

B.1.2.3 Networking Configuration Details

All machines are running Ubuntu 16.04 LTS linux distribution. The following steps have been followed in both server and client machine cases to perform the basic setup process, in terms of networking.

First we enabled the root user by creating a password

```
sudo passwd
```

Configure /etc/hostname as gqserver and gqclient respectively

Add to /etc/hosts for both server vm and client vm

```
192.168.20.30    www.example.org
```

Install ssh

```
apt-get install ssh
```

Enable ssh root connections on /etc/ssh/sshd_config

```
PermitRootLogin yes
```

Restart SSH

```
service ssh restart
```

Distribution upgrade

```
apt-get dist-upgrade -y
```

Configure static ips on /etc/network/interfaces.

For example this is the configuration file for server VM, if wan router for internet is 192.168.2.1

```
auto ens18
iface ens18 inet static
address 192.168.2.30
netmask 255.255.255.0
gateway 192.168.2.1

auto ens19
iface ens19 inet static
address 192.168.20.30
netmask 255.255.255.0
For client vm
auto ens18
iface ens18 inet static
address 192.168.2.31
netmask 255.255.255.0
gateway 192.168.2.1

auto ens19
iface ens19 inet static
address 192.168.20.31
netmask 255.255.255.0
```

Install needed software

```
apt-get install lynx wget emacs bc parallel ncdu linuxinfo
htop iftop iotop hdparm ethtool
```

Purge resolvconf and reboot

```
apt-get purge resolvconf; rm /etc/resolv.conf; shutdown -r 0
```

Configure /etc/resolv.conf with dns server, for example

```
echo "nameserver 192.168.2.1" >> /etc/resolv.conf
```

Install proto-quic library

```
cd /opt
git clone https://github.com/google/proto-quic.git
cd proto-quic
export PROTO_QUIC_ROOT=`pwd`/src
export PATH=$PATH:`pwd`/depot_tools
```

Proto quic library include ninja v1.6. This version will not compile quic_server and quic_client

We install ninja make tool [14] from git.

```
git clone git://github.com/ninja-build/ninja.git && cd ninja
git checkout release
./configure.py --bootstrap
```

Replace ninja v1.6 from proto-quic with 1.7.2

```
cd /opt/proto-quic/depot_tools/
mv ninja-linux64 ninja-linux64.bak
cp /opt/ninja/ninja ninja-linux64
```

Compilation will take a while (30 minutes on a dual core 4790S 3 Ghz), so we may need to shutdown VM, increase cores and then boot again.

On server VM we compile quic_server. We compile also quic_client for initial tests

```
./proto_quic_tools/sync.sh
./src/build/install-build-deps.sh
cd src
gn gen out/Default && ninja -C out/Default quic_server
quic_client net_unittests
```

On the client VM, we compile only quic_client

```
./proto_quic_tools/sync.sh
./src/build/install-build-deps.sh
cd src
gn gen out/Default && ninja -C out/Default quic_client
net_unittests
```

On both server and client VMs, we create a directory to put our scripts in and add it to PATH

```
mkdir -p /opt/goquic/bin
echo "PATH=\$PATH:/opt/goquic/bin" > /etc/profile.d/goquic.sh
```

Each machine, except gqclient must act as a router as well. The routing capabilities are enabled by adding the following lines to /etc/rc.local configuration file:

```
/etc/rc.local on gqserver, gqrouter1 and gqrouter2
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -t nat -A POSTROUTING --out-interface ens18 -j
MASQUERADE
iptables -A FORWARD --in-interface ens19 -j ACCEPT
```

On gqrouter1 and gqrouter2

```
apt-get purge resolvconf
rm /etc/resolv.conf
echo "nameserver 8.8.8.8" > /etc/resolv.conf
```

On gqrouter1, inside the the /etc/network/interfaces file

```
auto ens18
iface ens18 inet static
address 192.168.20.32
```

```
gateway 192.168.20.30

auto ens19
iface ens19 inet static
address 192.168.30.1
```

On gqrouter2, inside the /etc/network/interfaces file

```
auto ens18
iface ens18 inet static
address 192.168.30.33
gateway 192.168.30.1

auto ens19
iface ens19 inet static
address 192.168.40.1
```

On gqclient, inside the /etc/network/interfaces file

```
auto ens18
iface ens18 inet static
address 192.168.40.31
gateway 192.168.40.1
```

We enable execution of /etc/rc.local file on boot

```
systemctl enable rc-local.service
```

On the host machine we make sure that

- gqserver bridges ens18 with vmbr0 and ens19 with vmbr1
- gqrouter1 bridges ens18 with vmbr1 and ens19 with vmbr0
- gqrouter2 bridges ens18 with vmbr0 and ens19 with vmbr1
- gqclient bridges ens18 with vmbr1 and ens19 with vmbr0

For verification reasons, traceroute command is executed on gqclient

```
traceroute www.example.org
traceroute to www.example.org (192.168.20.30), 30 hops max, 60
byte packets
 1  192.168.40.1 (192.168.40.1)  0.124 ms  0.118 ms  0.112 ms
 2  192.168.30.1 (192.168.30.1)  0.242 ms  0.238 ms  0.234 ms
 3  www.example.org (192.168.20.30)  0.322 ms  0.317 ms  0.313
ms
```

B.1.2.4 Setting up the QUIC variant of web server software

Prepare test data and Create test domain data [15]:

```
mkdir /opt/quic-data
cd /opt/quic-data
wget -p --save-headers https://www.example.org
```

Add the X-Original-Url: https://www.example.org/ to /opt/www.example.org/index.html

```
sed -i "\#Connection:#i X-Original-Url:
https://www.example.org/index.html" index.html
```

Install the necessary certificates. Since quic is running on secure ssl connection we need to generate certificates. The script generates a 3 days valid certificate. To increase that we modify /opt/proto-quic/src/net/tools/quic/certs/generate-certs.sh code and set it to 30

```
cd /opt/proto-quic/src/net/tools/quic/certs/
sed -i "s/-days 3/-days 30/g" generate-certs.sh
./generate-certs.sh
cd /opt/proto-quic/src/
```

We add the certificate to root user on server VM

```
certutil -d sql:$HOME/.pki/nssdb -A -t "C,," -n goquic -i
/opt/proto-quic/src/net/tools/quic/certs/out/2048-sha256-
root.pem
```

If any problems, we can delete and reapply. Some more useful commands are:

List certificates

```
certutil -d sql:$HOME/.pki/nssdb -L
```

Delete certificate

```
certutil -d sql:$HOME/.pki/nssdb -D -n goquic
```

Copy certificates from server to client machine:

chmod g+r,o+r -R /opt/goquic/certs/
scp /opt/proto-quic/src/net/tools/quic/certs/out/* 192.168.20.31:/opt/goquic/certs

On client

```
mkdir /opt/goquic/certs/
chmod g+r,o+r -R /opt/goquic/certs/
certutil -d sql:$HOME/.pki/nssdb -A -t "C,," -n goquic -i
/opt/goquic/certs/2048-sha256-root.pem
```

Finally, the quic_server software [17] is activated by the following commands:

```
/opt/proto-quic/src/out/Default/quic_server --
quic_response_cache_dir=/opt/quic-data/www.example.org   --
certificate_file=/opt/proto-
quic/src/net/tools/quic/certs/out/leaf_cert.pem   --
key_file=/opt/proto-
quic/src/net/tools/quic/certs/out/leaf_cert.pkcs8
```

B.1.2.5 Setting up the QUIC variant of web client software (Command Line)

For starting, a good practice is to initially install and run the client software on the server VM [16]. The following test assures that the quic server installation is well done.

```
/opt/proto-quic/src/out/Default/quic_client --host=127.0.0.1 -
-port=6121 https://www.example.org
```

At second stage, the main part of the installation of the client software is performed [17], starting with the creation of a necessary GUI user (i.e., a user account equipped with graphical user interface capabilities):

```
adduser gquser
```

After that, the installation of the necessary certificates for the “root” user and for the “gquser” user is performed:

```
chown user -R /opt
certutil -d sql:$HOME/.pki/nssdb -A -t "C,," -n goquic -i
/opt/goquic/certs/2048-sha256-root.pem
su - gquser
certutil -d sql:$HOME/.pki/nssdb -A -t "C,," -n goquic -i
/opt/goquic/certs/2048-sha256-root.pem
```

After successful installation of the quic_client software on the server VM, we repeat the process on the client VM for both “root” user and “gquser”

```
/opt/proto-quic/src/out/Default/quic_client --
host=192.168.20.30 --port=6121 https://www.example.org
apt-get install lightdm xfce4
shutdown -r 0
```

B.1.2.6 Setting up the QUIC variant of web client software (Chrome version)

On Ubuntu 16.04, the default packaged chromium-browser was too buggy, when used from command line, so the latest Chrome version had to be installed [18], as follows:

```
apt-get install libgconf2-4 libnss3-1d libxss1
wget https://dl.google.com/linux/direct/google-chrome-
stable_current_amd64.deb
dpkg -i google-chrome-stable_current_amd64.deb
rm -f dpkg -i google-chrome-stable_current_amd64.deb
```

After that, one can login by the graphical interface as gquser, open a linux terminal and execute:

```
google-chrome   --user-data-dir=/tmp/chrome-profile   --no-
proxy-server --enable-quic   --origin-to-force-quic-
on=www.example.org:443   --host-resolver-rules='MAP
www.example.org:443 192.168.2.30:6121'
```

B.1.2.7 Enabling HAR capability

A good practice during the experiments is to have the timing with Chrome automated, as client is using headless Chrome and the har-specific info is captured by the Chrome-har-capturer. For this reason, the chrome-har-capturer is installed:

```
apt-get install npm
npm install chrome-har-capturer
echo "PATH=\$PATH:/root/node_modules/.bin" >>
/etc/profile.d/goquic.sh
Install node.js
curl
https://raw.githubusercontent.com/isaacs/nave/master/nave.sh |
bash -s -- usemain latest
```

After that, for testing reasons, the har files are captured by the following steps:

a) Run chrome as a service headless:

```
google-chrome --remote-debugging-port=9222 --headless
```

b) Capture har:

```
chrome-har-capturer -o example.har
https://www.example.org/1MB/index1.html
```

The corresponding file, example.har, contains all the important information of the URLs being loaded by Chrome along with the corresponding load times.

B.1.2.8 Setup Apache Server

The Installation/Configuration process requires the generation of the necessary certificate [20] (a self-signed certificate):

```
cd /etc/apache2
/usr/lib/ssl/misc/CA.sh -newcert
Enter PEM pass phrase:1234
Verifying - Enter PEM pass phrase:1234
Country Name (2 letter code) [AU]:GR
State or Province Name (full name) [Some-State]:Athens
Locality Name (eg, city) []:Athens
Organization Name (eg, company) [Internet Widgits Pty
Ltd]:Dimokritos
Organizational Unit Name (eg, section) []:GoQuic
Common Name (e.g. server FQDN or YOUR name) []:www.example.org
Email Address []:
..
cp newkey.pem newkey.pem.org
openssl rsa -in newkey.pem.org -out newkey.pem
chmod 400 newkey.pem newcert.pem
```

From server VM, we copy certificates to client

```
scp /etc/apache2/*.pem 192.168.2.31:/opt/goquic/certs/
```

On client VM we import certificates as “root” and as “gquser” (GUI) user

```
mkdir /opt/goquic/certs/
chmod g+r,o+r -R /opt/goquic/certs/
certutil -d sql:$HOME/.pki/nssdb -A -t "C,," -n goquicapache -i
/opt/goquic/certs/newcert.pem
su - gquser
certutil -d sql:$HOME/.pki/nssdb -A -t "C,," -n goquicapache -i
/opt/goquic/certs/newcert.pem
```

Configure Apache

```
apt-get install apache2
mkdir -p /opt/apache-data/www.example.org/
```

We create the following configuration file /etc/apache2/sites-available/goquic.conf.

The setup process is the same for both the HTTP and the HTTPS case. The ssl directives had to be load to the self-generated keys being created.

Headers are applied to all the files, in order always seem expired and disable cache features of the clients to allow our tests to always download files and never use cache.

```
<VirtualHost *:80>
        ServerName www.example.org
        DocumentRoot /opt/apache-data/www.example.org
        <Directory /opt/apache-data/www.example.org>
                AllowOverride All
                Require all granted
        </Directory>
        ErrorLog ${APACHE_LOG_DIR}/error.log
        CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>


<VirtualHost *:443>
        ServerName www.example.org
        DocumentRoot /opt/apache-data/www.example.org
        <Directory /opt/apache-data/www.example.org>
                AllowOverride All
                Require all granted
        </Directory>
        ErrorLog ${APACHE_LOG_DIR}/error.log
        CustomLog ${APACHE_LOG_DIR}/access.log combined
        SSLEngine on
        SSLCertificateFile /etc/apache2/newcert.pem
        SSLCertificateKeyFile /etc/apache2/newkey.pem
        #SSLCACertificateFile /etc/apache2/newca.pem
        FileETag None
        <IfModule mod_headers.c>
            Header unset ETag
            Header set Cache-Control "max-age=0, no-cache, no-
store, must-revalidate"
            Header set Pragma "no-cache"
            Header set Expires "Wed, 12 Jan 1980 05:00:00 GMT"
        </IfModule>
</VirtualHost>
```

To enable the whole site the following command should be executed:

```
a2dissite 000-default.conf a2ensite goquic.conf mkdir
/opt/apache-data/www.example.org chgrp www-data -R
/opt/apache-data/www.example.org a2enmod ssl service apache2
restart
```

B.1.2.9 Setup SPDY (HTTP/2) in Apache Server

Installation/Configuration: First enabling [21] the http2 functionality [22] is required. Apache on Ubuntu 16.04 does not support this built-in feature so it is necessary to reinstall the latest version of the apache software [23], [24] as follows:

```
add-apt-repository ppa:ondrej/apache2
```

```
apt-get update
apt-get dist-upgrade
echo 'LoadModule http2_module
/usr/lib/apache2/modules/mod_http2.so
  <IfModule http2_module>
  LogLevel http2:info
  </IfModule>' > /etc/apache2/mods-available/http2.load
a2enmod http2
service apache2 restart
```

However, there is another problem to be solved, as indicated by the logs:

```
[http2:warn] [pid 18365] AH10034: The mpm module (prefork.c) is
not supported by mod_http2. The mpm determines how things are
processed in your server. HTTP/2 has more demands in this regard
and the currently selected mpm will just not do. This is an
advisory warning. Your server will continue to work, but the
HTTP/2 protocol will be inactive.
```

This problem means that Apache 2.4.27 /2 is not supported in prefork mpm. To solve this, a different mpm [25] like event has to be selected. This was done by the following steps (details in https://http2.pro/doc/Apache):

```
apt-get install php7.1-fpm
apt-get install php7.0-fpm
a2enmod proxy_fcgi setenvif
a2enconf php7.0-fpm
a2dismod php7.0
a2dismod mpm_prefork
a2enmod mpm_event
service apache2 restart
```

As SPDY comes as a module on apache 2.4, several dependencies had to be installed, in order to compile it:

```
apt-get -y install git g++ libapr1-dev libaprutil1-dev curl
patch binutils make devscripts
```

The latest SPDY module was downloaded from git repository, compiled:

```
mkdir /opt/spdy
cd /opt/spdy
git clone https://github.com/eousphoros/mod-spdy.git
cd mod-spdy/src
./build_modssl_with_npn.sh
chmod +x ./build/gyp_chromium
make BUILDTYPE=Release
```

After that the mod_spdy module had to be installed and thus configured and applied to apache:

```
cp out/Release/libmod_spdy.so
/usr/lib/apache2/modules/mod_spdy.so
echo "LoadModule spdy_module
/usr/lib/apache2/modules/mod_spdy.so" > /etc/apache2/mods-
available/spdy.load
```

```
echo "SpdyEnabled on" > /etc/apache2/mods-available/spdy.conf
a2enmod spdy
service apache2 restart
```

To test if mod_spdy works, the URL https://www.example.org/1MB/index1.html has to be opened from Firefox and by using Firebug Network/All and selecting the index1.html one shall see:

```
X-Firefox-Spdy: h2
```

B.1.3 Background / Motivation

Web service providers and operators need to enhance their networks and effectively manage transport protocols and other network parameters so as to deliver top quality of experience to their subscribers. So far, QUIC has managed to provide very good results. According to Google [1], QUIC has helped to realize a 3% improvement in the mean page load times on Google Search. That may not seem high but one needs to recall that Google search is already about as optimized as possible. Other sites, and especially latency-heavy web apps, will likely see better improvements. Users who connect to YouTube over QUIC report about 30% fewer re-buffers when watching videos and because of QUIC's improved congestion control and loss recover over UDP, users on some of the slowest connection also see improved page load times.

This improved speed and optimization comes at the perfect time. Applications, videos and games have continued to dominate Internet usage. More specific, video has skyrocketed over the past years with Gartner Research reporting an estimated 59% increase in mobile video traffic in 2015 [2]. Nevertheless, solutions must also enable operators to manage and optimize future traffic patterns and also web service providers must have access to an easy to implement framework that allows them to manage new types of content and protocols.

Go-Quick proposes an experiment that will consider a range of network conditions in order to provide quantitative results on the performance of some key cross-layer protocols and the experimental outcomes have provided useful insights for the most appropriate network regimes that certain transport protocols can operate. Therefore the company's strategy is to exploit the results of Go-Quick targeting to Network Operators and Content Service Providers (CSPs) through the distribution channels and the of 8BELLS.

B.2 Technical Results & Lessons learned

After setting up the all the necessary VMs, the corresponding web client should start requesting content from the server via the two router nodes, under different network conditions. Three basic client – server pairs should be tested: QUIC client – QUIC server, HTTP client – HTTP server, HTTP/2 client – HTTP/2 server. Each of these categories might have further variants, e.g., depending on the existence of command line client tool (better automation) or not. The following report on the different cases being setup and tested.

B.2.1.1 Initial Tests (Server VM - Generate test)

We copy various files [26] for our tests from the folder /opt/goquic/images. The files are approximately 10MB, 1MB, 100KB and 10KB

```
ls -l /opt/goquic/images/
```

- `rw-r–r– 1 root root 103250 Σεπ 6 18:50 testimage100KB.jpg`
- `rw-r–r– 1 root root 10215 Σεπ 6 18:50 testimage10KB.jpg`
- `rw-r–r– 1 root root 10438161 Σεπ 6 18:50 testimage10MB.png`
- `rw-r–r– 1 root root 1069168 Σεπ 6 18:50 testimage1MB.jpg`

Before placing the files in the /opt/quic-data/www.example.org/ folders, for QUIC to serve them, we need to copy them under /opt/apache-data/www.example.org/ on respective folders, e.g.:

```
mkdir /opt/apache-data/www.example.org/1MB/
cp /opt/goquic/images/testimage1MB.jpg /opt/apache-
data/www.example.org/1MB/
```

Download them to quic folder together with the headers

```
mkdir /opt/apache-data/www.example.org/1MB/
wget --save-headers http://127.0.0.1/1MB/testimage1MB.jpg -O
/opt/quic-data/www.example.org/1MB/
```

Add extra header needed for quic:

```
sed -i "\#Connection:#i X-Original-Url:
https://www.example.org/1MB/testimage1MB.jpg /opt/quic-
data/www.example.org/1MB/testimage1MB.jpg"
```

To automate this procedure along we created a script which does also the following:

- Generates duplicates of the test file with different names to test multiple downloads. E.g. n files named testimage1MB-n.jpg
- Creates various indexn.html files that load multiple files on a single page e.g. index10.html loads 10 images testimage1MB-1.jpg .. testimage1MB-10.jpg

For example, to create 100 1MB files and various index$n.html files (index1.html, index2.html, index4.html, etc.):

```
testGen -x -i /opt/goquic/images/testimage1MB.jpg -f
1MB/testimage1MB.jpg -n 100 -g "1 2 4 8 10 15 20 30 40 50 60
80 100"
```

All tests are generated by the script “testGenAll”.

```
#!/bin/bash

testGen -x -i /opt/goquic/images/testimage10MB.png -f
10MB/testimage10MB.png -n 30 -g "1 2 4 8 10 15 20 30"

testGen -x -i /opt/goquic/images/testimage1MB.jpg -f
1MB/testimage1MB.jpg -n 100 -g "1 2 4 8 10 15 20 30 40 50 60
80 100"
```

```
testGen -x -i /opt/goquic/images/testimage100KB.jpg -f
100KB/testimage100KB.jpg -n 1000 -g "1 2 4 8 10 15 20 30 40 50
60 80 100 200 500 1000"

testGen -x -i /opt/goquic/images/testimage10KB.jpg -f
10KB/testimage10KB.jpg -n 1000 -g "1 2 4 8 10 15 20 30 40 50
60 80 100 200 500 1000"
```

We also use a script to facilitate starting/stopping server as follows:

```
/opt/goquic/bin/doService -s quic -c start
```

and we add it to /etc/rc.local to start on boot.

B.2.1.2 Retrieving Content using the quic_client (Command Line)

To receive a file from the quic_client the following command is executed:

```
/opt/proto-quic/src/out/Default/quic_client --
host=192.168.20.30 --port=6121
https://www.example.org/1MB/testimage1MB-1.jpg>
1MB/testimage1MB-1.jpg
```

However, this file is downloaded along with header info that we need to strip to reconstruct the original file

```
destfile=1MB/testimage1MB-1.jpg
sed -i '/Response:/,$!d' $destfile
sed -i '/body: /,$!d' $destfile
sed -i "s/body: //g" $destfile
sed -i "/trailers/d" $destfile
sed -i "/Request succeeded/d" $destfile
perl -pi -e 'chomp if eof' $destfile
```

To automate this procedure a suitable script was created performs the following steps, among others:

- Retrieves all files sequentially or in parallel. We tested in parallel (-p all)
- Queues concurrent download so as not to exceed a maximum always 10 concurrent downloads (-p 10) of 30 total downloads (-n 30).
- Each specific test is run 3 times (-t 3) and we get the average in Mbits/sec (-b Mb)
- To facilitate the process of importing data in a spreadsheet for analysis, the summary of the results is appended to a TAB delimited file (-e results.txt)
- The same script is used for all protocols. For QUIC (-s quic), for HTTPS (-s https). Not used for SPDY/HTTP2 since wget does not support it

For example, to receive from QUIC server 10 concurrent files of 1MB size, executing the test 2 times, the following commands are necessary:

```
cd /opt/goquic

getFiles -x -p all -t 2 -b Mb -e results.txt -s "quic" -i
192.168.20.30 -u https://www.example.org/1MB/testimage1MB.jpg
-d 1MB -n 10
```

Both quic_server and quic_client executables, been as provided by Google, are not optimized and do not scale (see note in red in [17]). Indeed, the corresponding software uses only 1 thread that is limited by single thread CPU performance and cannot handle too many connections per machine.

Via experimentation, the maximum file number, plausible to download should be set to:

- Up to 20 concurrent files of 10MB size
- Up to 40 concurrent files of 1MB size
- Up to 60 concurrent files of 100KB size
- Up to 10 concurrent files of 10KB size

This issue can be solved by limiting concurrent download number and queuing. As example, we invoke 100 downloads to happen, through 30 concurrent connections. We called the script as follows

```
getFiles -x -p 30 -t 3 -b Mb -e results.txt -s "quic" -i
192.168.20.30 -u https://www.example.org/1MB/testimage1MB.jpg
-d 1MB -n 100
```

This also worked for 100KB files. We requested 1000 files, while applying queuing by up to 15 concurrent connections.

```
getFiles -x -p 15 -t 3 -b Mb -e results.txt -s "quic" -i
192.168.20.30 -u
https://www.example.org/100KB/testimage100KB.jpg -d 100KB -n
1000
```

However we cannot increase, by using this method, the number of concurrent connections for very small files (10KB) or very big files (10MB).

After reached the concurrent download limit for each case (e.g. over 10 concurrent files of 10KB size) failures are starting to occur.

```
Request failed (404).
```

or, sometimes,

```
[0904/183522.840864:WARNING:proof_verifier_chromium.cc(452)]
Failed to verify certificate chain:
net::ERR_CERT_AUTHORITY_INVALID

Failed to connect to 192.168.20.30:6121. Error:
QUIC_PROOF_INVALID
```

On failures, we did a restart on server and client, to clean any lingering connections, prior retrying with a smaller number of concurrent connections.

These quic_client related limitations were not observed when using the native Google Chrome browser as a client.

B.2.1.3 Retrieving Content using the Chrome client for QUIC (with GUI)

We try to get a full page with all images from **one client call**, since the QUIC protocol might behave better if we fetch all images **in same session**. So we try from Chrome (which has better performance than quic_client). To make Chrome to act as a **quic** client, the following command should be executed:

```
google-chrome --user-data-dir=/tmp/chrome-profile   --no-
proxy-server --enable-quic   --origin-to-force-quic-
on=www.example.org:443   --host-resolver-rules='MAP
www.example.org:443 192.168.20.30:6121'
https://www.example.org/1MB/index.html
```

We press **ctr-shift-I** to get to Tools/Developer/Network and press **F5** to reload and see load times. To verify the protocol:

- Right clicking on the columns we can add Protocol column or
- We open a new tab Chrome:////net-internals/#quic. We may need to reload again previous tab to see active quic sessions

As a normal **HTTPS over TCP** client:

```
google-chrome –user-data-dir=/tmp/chrome-profile
https://www.example.org/1MB/index.html
```

We press Ctrl-Shift-I to get to Tools/Developer/Network and press F5 to reload and see load times.

As mentioned above, the web page content to be retrieved by the client consisted of a variable number of objects (photo files) of variable number. The most typical cases representing characteristic real world conditions included:

- Few large photos of 1MB each, per web page
- A moderate number of photos of 100KB each, per web page
- A large number of small photos of 10KB each, per web page

Figure 2 depicts an indicative case of web page characteristics, during the experiments, containing file objects (photos) of 100 KB each.

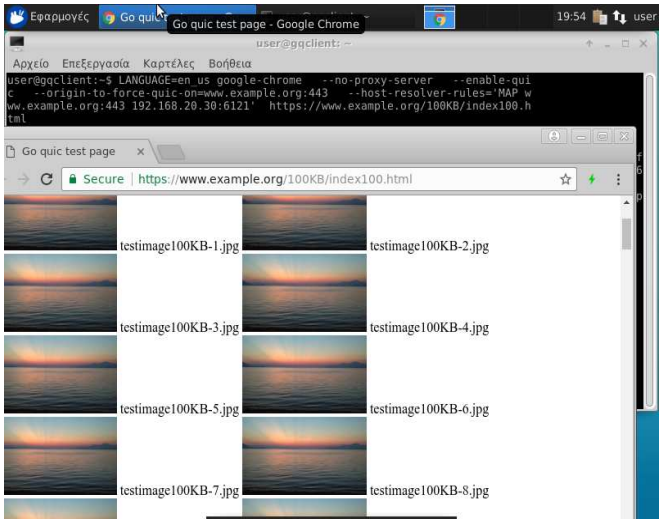

Figure 2: An indicative web page case during the experiments, containing file objects (photos) of 100 KB each

To verify the protocol:

- Right clicking on the columns we can add Protocol column or
- we open a new tab Chrome:////net-internals/#quic. We may need to reload again previous tab to see active quic sessions

There is also a nice Chrome extension which shows if a page has HTTP2 or QUIC support. Chrome has abandoned the support for SPDY over HTTP2.

By enabling HTTP2 on apache server we can test again using HTTP2:

```
a2enmod http2; a2enmod spdy; service apache2 restart
```

B.2.1.4 Apache2 tests: Retrieving files with wget (Command Line)

To retrieve content using the HTTP (or the HTTP2 protocol) [27], [28] the following is executed:

```
wget --no-check-certificate
https://www.example.org/10MB/testimage10MB.png
```

To automate all tests we used getFiles script with exactly the same parameters replacing only **-s quic** with **-s https**. For example to receive from apache server 40 concurrent files of 1MB run:

```
cd /opt/goquic
getFiles -x -p all -t 3 -b Mb -e results.txt -s "https" -i
192.168.20.30 -u https://www.example.org/1MB/testimage1MB.jpg
-d 1MB -n 40
```

Or, for 100 downloads, limited by up to 30 concurrent connections:

```
getFiles -x -p 30 -t 3 -b Mb -e results.txt -s "https" -i
192.168.20.30 -u https://www.example.org/1MB/testimage1MB.jpg
-d 1MB -n 100
```

B.2.1.5 Retrieving Content with HAR

We can automate testing with Chrome [29] using Chrome-har-capturer and running Chrome headless. We have automated this test also and embedded it to getFiles script as an option.

This test was used for both https and https/2 requests.

However Chrome seems not to support QUIC when running headless. Passing the same option along with headless options did not work and searching for documentation or information on the Internet on this matter yielded no results.

In Figure 3, as an example, the typical command line method is depicted, invoking a series of automated Chrome tests using the getFiles script.

```
root@gqclient:/opt/goquic# getFiles -x -p all -t 3 -b Mb -e results.txt -c chrome -s https -u https:
//www.example.org/100KB/index500.html -d 100KB
cleanup directory 100KB
- https://www.example.org/100KB/index500.html
Download time : 3.514343173 secs
Downloaded bytes : 49.23 MB
Download speed : 279.91 Mb/sec
Success!
- https://www.example.org/100KB/index500.html
Download time : 3.299073358 secs
Downloaded bytes : 49.23 MB
Download speed : 302.95 Mb/sec
Success!
- https://www.example.org/100KB/index500.html
Download time : 3.569315720 secs
Downloaded bytes : 49.23 MB
Download speed : 302.72 Mb/sec
Success!
Tests run 3 times. Results in Mb/sec
279,91  302,95  302,72
Average speed : 295,19
root@gqclient:/opt/goquic#
```

Figure 3: Instance of HTTP request generation, through command line, using Chrome as client, asking for 100KB file content from the web

In Figure 4, an example of graphical client view is depicted. More specifically, progressive downloading of a test page containing 100 KB file objects with (bottom) and without (up) security issues to override.

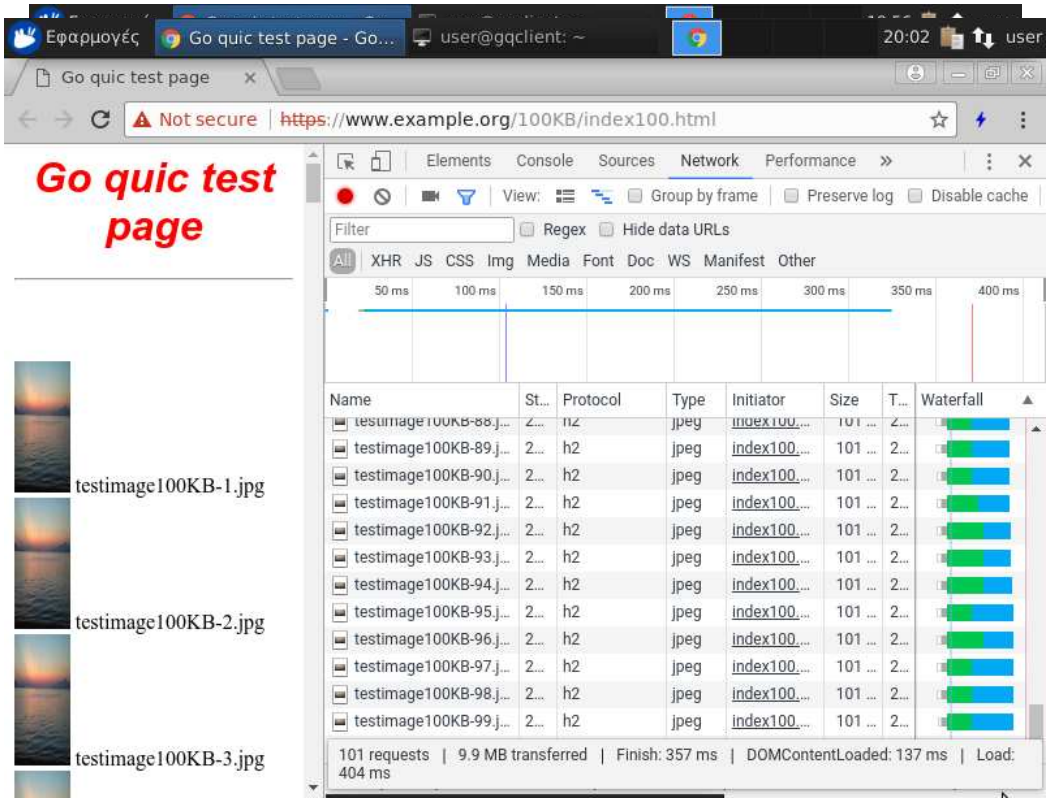

Figure 4: Progressive downloading of a test page containing 100 KB file objects with (bottom) and without (up) security issues to override

B.2.1.6 Performance of the candidate protocols

After installation and exhaustive testing of the various flavors of each protocol, more causes of differentiation in performance made the scene, as not all protocols are of the same "maturity" in terms of code architecture and some of them are more favored by the developing organization policy than the others. Indeed, Google has stopped supporting the QUIC protocol and promoted the HTTP/2 instead [13] in its web servers worldwide, as a successor.

Initial tests on uncongested network

Initial tests focused on the goodput rate characterizing different client – server implementations, while trying to get concurrent files (photos) of specific size each at very good network conditions. Figure 5Figure 6,Figure 7,Figure 8 are depicting the behavior of several variants of the web protocols implementations under testing, while asking for file objects (photos) of 10 MB, 1 MB, 100 KB and 10 KB, respectively.

After a meticulous gathering and study of results, similar to the ones depicted in Figure 5Figure 6,Figure 7,Figure 8, it is certain that the HTTP-based web protocol implementations (i.e., pure HTTP and HTTP2 – former SPDY) are exhibiting really fast results which are close in performance. Indeed, their code has been optimized for years and an efficient multithreading schema is used to serve the concurrent file object requests from the web pages.

One the other hand, the QUIC protocol implementations, especially the public available one, are using less threading (even only one thread) to tackle with web page downloading issues. Things are better while working with the actual QUIC servers that Google is providing. The built-in packet pacing mechanism the QUIC implementations have, is another, not negligible, cause of poor performance in case of almost uncongested networks.

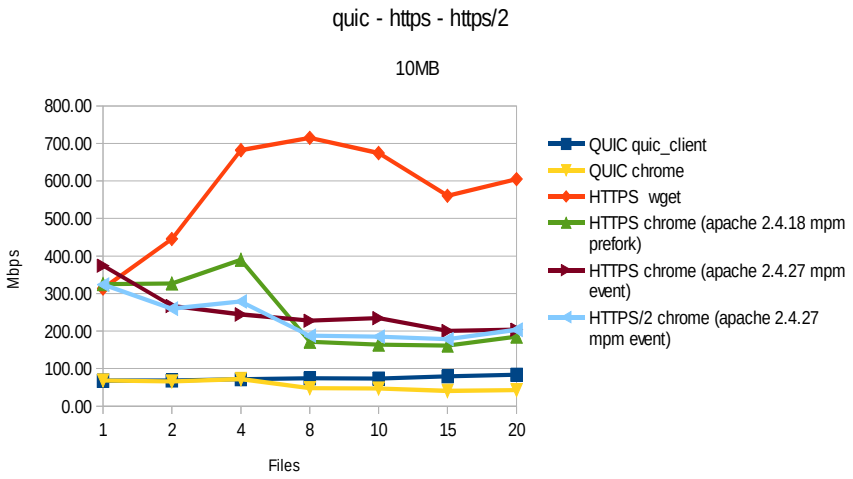

Figure 5: The performance of several variants of the web protocols implementations under testing, while asking for file objects (photos) of 10 MB, in size

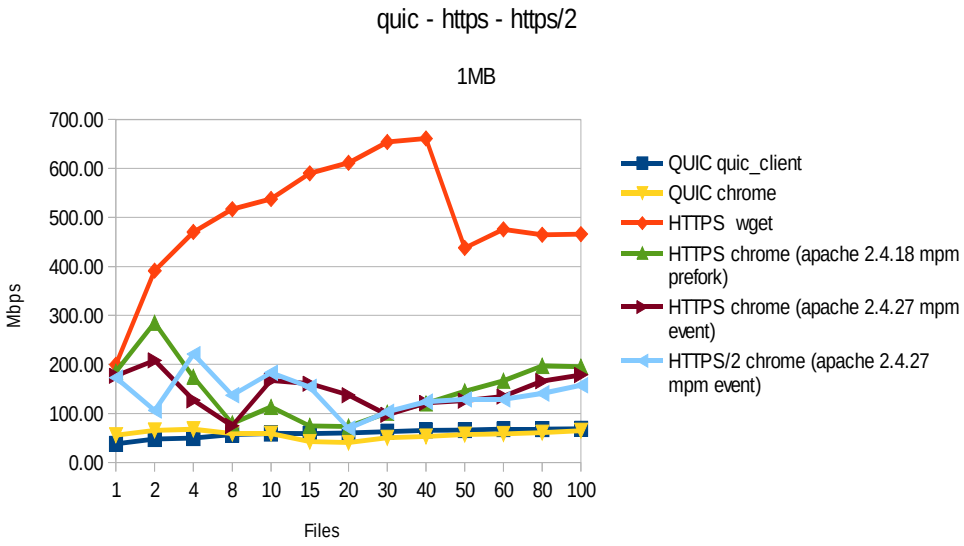

Figure 6: The performance of several variants of the web protocols implementations under testing, while asking for file objects (photos) of 1 MB, in size

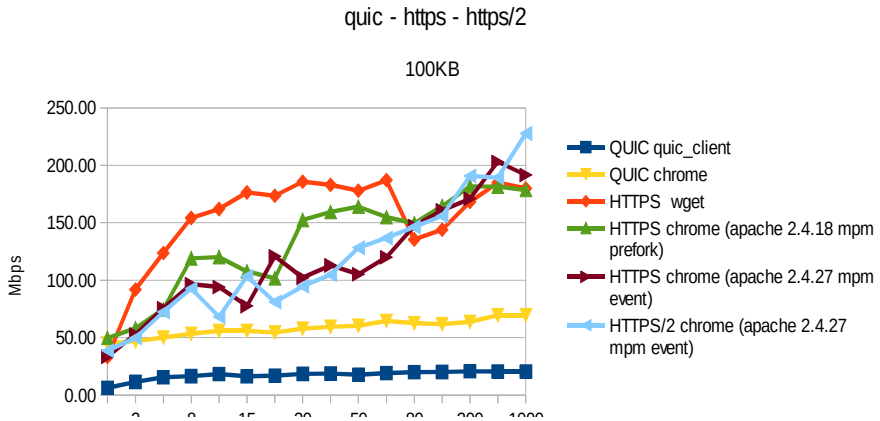

Figure 7: The performance of several variants of the web protocols implementations under testing, while asking for file objects (photos) of 100 KB, in size

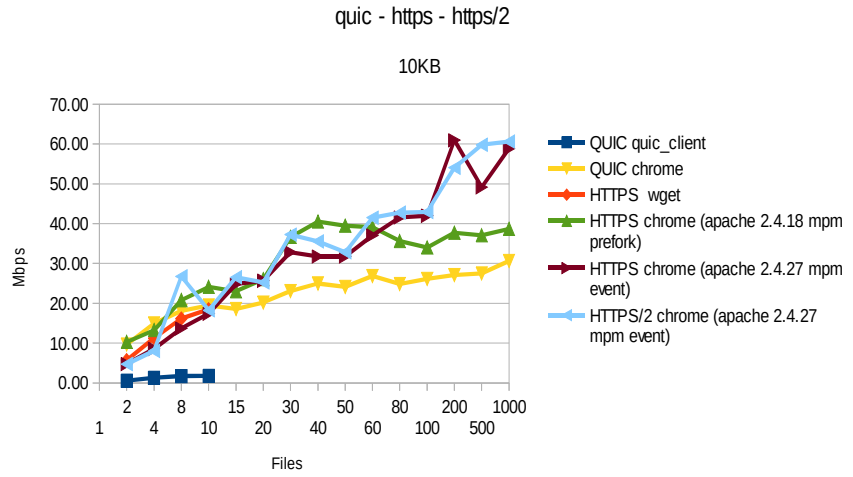

Figure 8: The performance of several variants of the web protocols implementations under testing, while asking for file objects (photos) of 10 KB, in size

As depicted in Figure 5Figure 6,Figure 7,Figure 8, the command line implementations (i.e., like wget) perform faster as they are less demanding in terms of computational power or memory to be allocated. Command line implementations allow for better automation during the experiments and thus lead to faster result acquisition. Unfortunately, graphical client-based implementations are closer to real world situations and provide more complete set of options for testing, so this type of experiments cannot be omitted.

Finally, Figure 5Figure 6,Figure 7,Figure 8 highlight the drop in performance, in terms of goodput rate, as file size to deliver is becoming smaller and its number is increasing. This issue will be further studied into the following session.

B.2.1.7 Further tests on congested networks

In order to recreate the suitable network congestion conditions we used, upon router machines, the netem function of the TC (Traffic Control) package and custom scripts to manipulate the bandwidth, the delay and the packet loss of the underlying network connection, and thus emulating a real network. For each characteristic case of network conditions, we performed a set of typical web content download actions, from the web server machine to the client machine through the router

machines, including from requests for a small number of large size web objects (i.e., photos) to requests for a large number of small web page objects (i.e., photos).

More specifically, the netem function creates policies that are applied by default only to the outgoing queue of a specific network interface. The fastest way to tackle with this idiosyncrasy was to apply the desired policy to both interfaces of each router (i.e., to the one towards the server site and to the one towards the client site). Apart from the delay and loss values a small statistical jitter was added to better emulate an actual network conditions (this jitter was following Pareto distribution as this is more likely to describe Internet network traffic conditions). A custom tool based on TC as well has been used to modify the bandwidth provided by each router network interface.

A typical usage for the netem function of the TC has as follows:

```
# tc qdisc add dev ens4 root netem loss 4% delay 100ms 10ms
distribution pareto

# tc qdisc change dev ens3 root netem loss 2% delay 50ms 5ms
distribution pareto

# tc qdisc change dev ens3 root
```

After performing series of measurements with diverse network characteristics and web client – server combinations we distinguished five indicative cases of network congestion:

- Negligible delays and losses. In this case, the only delay is caused by the interfaces and routing software of the Gigabit LAN of our topology.
- Comparatively low values for delays and losses. In this case, delay of 12.5 ms is added and losses of about 0.5%, on both upstream and downstream directions.
- Moderate values for delays and losses. In this case, delay of 25 ms is added and losses of about 1%, on both upstream and downstream directions.
- Moderate values for delays and losses. In this case, delay of 50 ms is added and losses of about 2%, on both upstream and downstream directions.
- Comparatively high values for delays and losses. In this case, delay of 100 ms is added and losses of about 4%, on both upstream and downstream directions.

Figure 9 depicts a screenshot containing typical network condition modification actions, during the experiments.

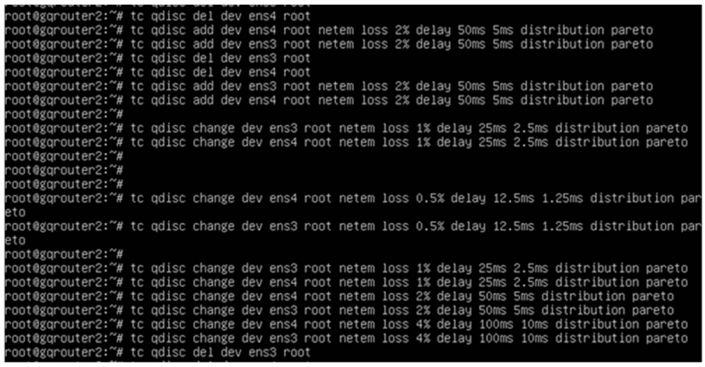

Figure 9: A Screenshot containing typical network condition modification actions, during the experiments

The ping [31] utility was used for fast verification of each network quality settings set being applied. It must be noted that ping provides round trip statistics, and this fact had to be taken account.

A typical example of ping command, as used during the experiments being performed has as follows:

```
#ping –I ens3 192.168.20.30 –i 0.01 –c 10000 –s 100
```

The above command will send, through interface ens3, 1000 consecutive packets of size 100 bytes each, with inter-packet distance of 10 ms. Figure 10 depicts a screenshot containing simple network condition verification process using the ping utility, during the experiments.

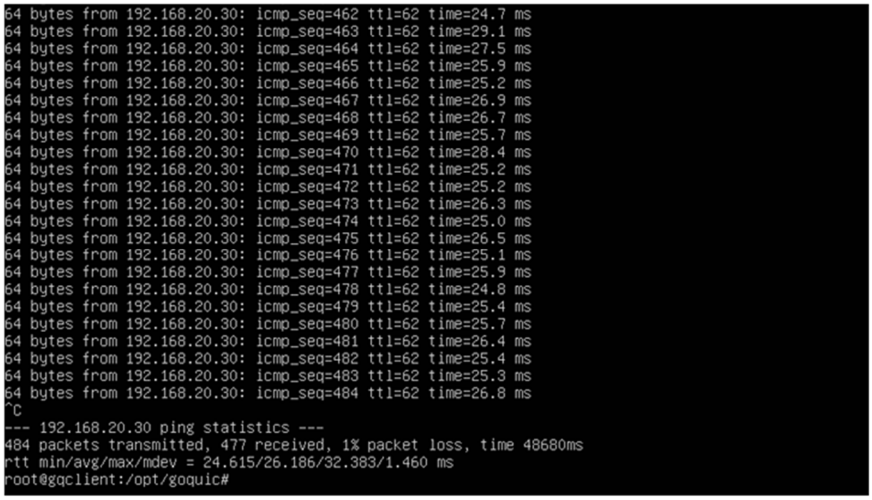

Figure 10: A Screenshot containing simple network condition verification process using the ping utility, during the experiments

Our experiments indicate that the HTTP – based web client and server flavors are faster in cases when network conditions tend to be ideal. This is merely due to the fact these versions are using more mature code which has been better optimized than its QUIC counterpart, which remains of experimental quality, at least, in its publicly available version. Indeed, by inspecting the CPU utilization of the machines running the application protocols under testing, we conclude that the QUIC-based implementations are much more CPU intensive than the HTTP–based ones. Another reason for that difference (i.e., for the underperformance of the QUIC protocol on good network conditions) is the packet pacing mechanism that the QUIC implementation has. No remarkable differences have been recorded between HTTPS and HTTP/2 (former SPDY) performance evaluation process (Indeed, the HTTP/2 tends to be slightly better under moderate network conditions than the native HTTPS).

Further experimentation reveals that, as network conditions deteriorate (i.e., network traffic delay and losses are obtaining higher values) the performance of the HTTP flavors is becoming drastically worse. On the other hand, the QUIC implementations are not so rapidly affected by the network deterioration and are exhibiting great tolerance to moderate delay and loss values. Finally, the QUIC protocol is the winner for relatively high values of network losses and delays.

This behavior is quite common for all object (photo files) size cases, from the few and larger ones to the many and small ones. Indeed, for smaller files, the QUIC protocol comes first at slightly higher values of network deterioration conditions.

Indicative results are shown in Table 1, while Figure 11 provides the corresponding graphical representations of these results. More specifically, it is depicted, in detail, the performance (goodput rate in logarithmic scale) of the most indicative variants of the web protocols implementations under testing, while asking for web page objects (photos) of 1MB, 100KB, 10KB, as function of the underlying network conditions (i.e., delay and loss in the form <delayvalue>_<lossvalue>).

Table 1: The performance (goodput rate) of the most indicative protocol variants under testing, for diverse network conditions and download object patterns

Files of 1MB			
delay_loss	https_1MB	http2_1MB	quic_1MB
0_0.0	493	509.67	49.45
12.5_0.5	83.38	81.62	30.72
25_1.0	29.05	31.96	22.86
50_2.0	10.87	11.9	11.93
100_4.0	3.83	3.86	5.98
Files of 100KB			
delay_loss	https_100KB	http2_100KB	quic_100KB
0_0.0	96.2	91.82	12.15
12.5_0.5	25.22	27.81	9.98
25_1.0	12.46	11.71	8.28
50_2.0	4.92	4.98	5.59
100_4.0	2.57	2.46	3.52
Files of 10KB			
delay_loss	https_10KB	http2_10KB	quic_10KB
0_0.0	9.12	9.15	1.33
12.5_0.5	3.05	2.98	1.01
25_1.0	1.46	1.86	0.92
50_2.0	0.81	0.76	0.68
100_4.0	0.33	0.31	0.46

Finally, we tried to measure the difference in performance that a specific web protocol is experiencing (based on goodput measurements) compared with the best performing case (of the same protocol), in terms of objects file pattern to download.

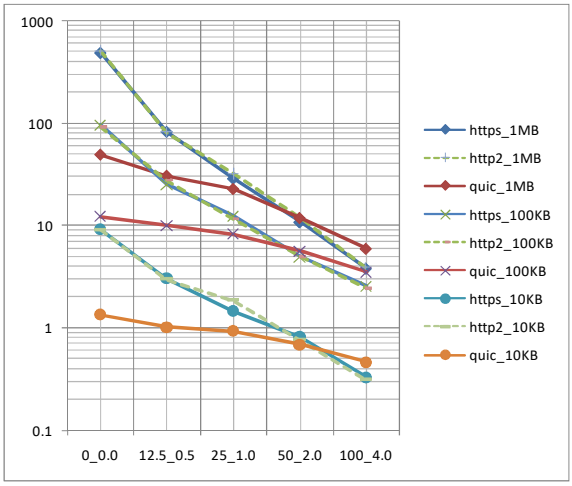

Figure 11: The performance (goodput rate) of the most indicative variants of the web protocols implementations under testing, as function of the underlying network conditions

Figure 12 illustrates the relative performance, as a percentage (in logarithmic scale) of the goodput rate value of the best performing case of the same protocol (i.e., of 1MB downloads), for the most indicative variants of the web protocols implementations under testing, for various network conditions, as function of the size of the file objects to be downloaded. The file size of objects to be downloaded (horizontal axis) varied from 10 KB to 1000 KB (1 MB).

The characteristic network conditions during these experiments as similar to the ones in case of experiments depicted in Figure 11, i.e.:

- Negligible delays and losses. In this case, the only delay is caused by the interfaces and routing software of the Gigabit LAN of our topology.
- Moderate values for delays and losses. In this case, delay of 50 ms is added and losses of about 2%.
- Comparatively high values for delays and losses. In this case, delay of 100 ms is added and losses of about 4%.

As depicted in Figure 11, and reassured in Figure 10, the file pattern of the objects to download (i.e., few large ones or many small ones) can drastically affect (for more than 50 times) the performance (i.e., the goodput rate) of the web protocols, favoring data with larger sizes. This difference in performance is comparable or even larger than the difference in performance invoked by the network conditions deterioration factor. Apparently, web application protocols are performing better when delivering fewer number of larger objects as such a case requires fewer number of connections to be treated or objects to be packed/tagged/assembled.

Despite this fact, the changes in performance over the file pattern changes of the QUIC protocol implementation compared to the changes in performance of the HTTP protocol seem not to follow too different patterns.

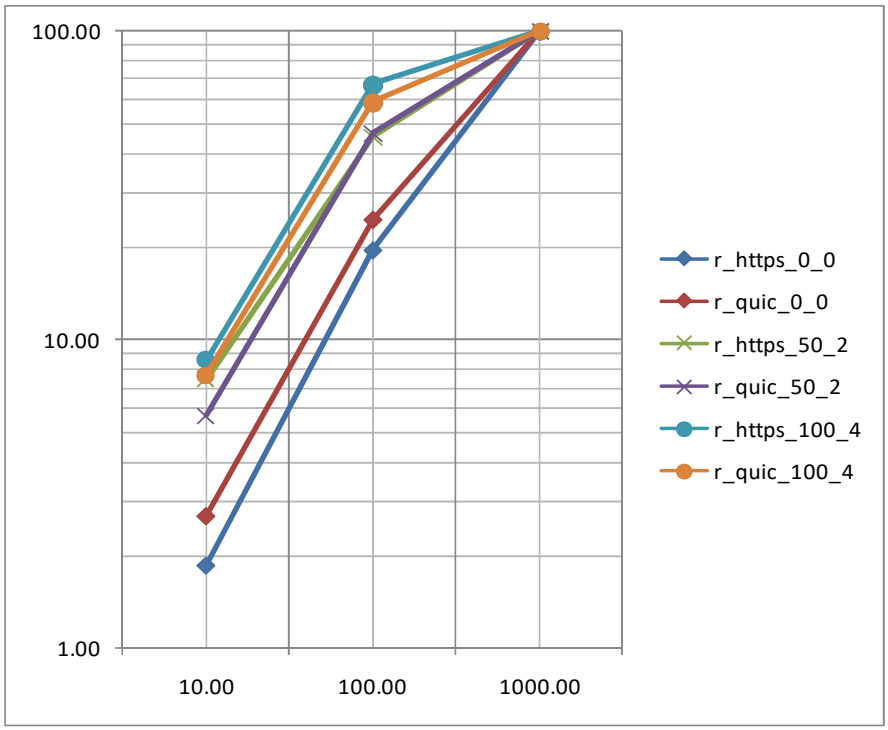

Figure 12: Relative performance (percentage) of the most indicative web protocols under testing, for various network conditions, as function of objects file pattern

Any differences here noticed are more apparent in case of HTTP-based implementations where they have to treat one TCP channel per object while the QUIC-based implementations use only one channel. As network conditions deteriorate, the difference between the best (few large files) and the worst performing cases (many small files) becomes smaller. The degree that this deterioration is affecting the HTTP implementations, compared with their QUIC counterparts, seems not to be of a remarkable difference.

All the above-mentioned results are in accordance with the ones presented in other similar works like in [10], [24].

B.2.2 Conclusions and Feedback to the Fed4FIRE Project Consortium

Go-Quick was targeted in evaluating the performance of the QUIC protocol, highlighting the need of real-life, small-scale experiments and under several network conditions. The procedures and code used for the experiment have been provided and reported so as to increase awareness of the performed innovations as well as to properly evaluate and assess the achieved outcomes.

Extensive experiments indicate that the QUIC protocol, although not in an optimized and "mature" publicly available software version, performs well in case of congested networks, where losses and delays are not negligible. On the other hand, HTTP and HTTP/2 (the successor of SPDY) are favored by large companies like Google and possess highly efficiently implemented clients and servers. These protocols are remarkably fast, especially when network conditions are relatively good.

The network conditions characterizing the link between the web server and the client remain the dominant factor of differences in performance of the QUIC-based implementations compared with the HTTP-based ones. More specifically, the QUIC-based implementations, although not so well optimized as their HTTP-based counterparts, exhibit a remarkably good performance upon network links of poor quality, as they don't have to deal with the "heavy" connection-oriented quality assurance mechanism the TCP/IP has. On the other hand, on almost uncongested networks, their performance is drastically slower than their HTTP-based counterparts and this is due not only to the poor optimization degree of their code but also to the presence of the packet pacing mechanism.

Furthermore, the file pattern of the file objects to be downloaded affects drastically the performance of both the QUIC-based implementations and the HTTP-based ones. Nevertheless, the relative changes in performance of the candidate protocol implementations, over the object file pattern to download, cannot be characterized by a remarkable diversity.

Another important issue is that the experiment has gathered a considerable amount of valuable experience and feedback remarks about the functional components and the platform, during the experiment setup and validation. Therefore, any type of error or imperfection encountered during the setup and validation phase was directly communicated to theFed4FIRE+ members. Go-Quick is now in position to provide its view on the best practices that should be pursued, as well as other recommendations as regarding the involvement in the development and validation of innovations to the current federated testbed. This testbed was proved powerful enough to support the required VMs and flexible enough to adapt to the diverse topology demands, during the experiments.

The accurate responses regarding the setup were beneficial for Go-Quick, as they assisted in fast deployment of experiments and Fed4FIRE+ took some important feedback for the improvements required on the current infrastructure (i.e., reporting potential bugs, flaws, etc.).

8BELLS has a strong motivation on comparing the three protocols under study, mainly targeting to offer various cross-layer approaches under specific network conditions, which are valuable

evaluations to meet clients' requirements. We consider that the Go-Quick experiment outcomes fit well to the company's value chain, providing both consumptive and productive use values.

Finally, 8BELLS is now ready to communicate and disseminate the results of Go-Quick raising awareness and impact on stakeholders and the wider community, as it has assessed the received feedback. The creation and exploitation of future synergies with other projects, that 8BELLS may participate as a partner will be investigated and may be capitalized. The participation of 8BELLS to these projects poses an excellent chance to communicate Fed4FIRE+ results directly to the European research and market communities.

B.2.3 References

[1] Google Wants To Speed Up The Web With Its QUIC Protocol, https://techcrunch.com/2015/04/18/google-wants-tospeed-up-the-web-with-its-quic-protocol/
[2] What's Driving Mobile Data Growth?, http://www.gartner.com/newsroom/id/2977917
[3] L. Popa et al., "Http as the narrow waist of the future internet", in Proc. of ACM SIGCOMM Workshop on HotNets, pages 6:1-6:6, Monterey, California, 2010.
[4] SPDY: An experimental protocol for a faster web, http://www.chromium.org/spdy/spdy-whitepaper
[5] Desktop Browser Version Market Share, https://www.netmarketshare.com/browser-marketshare. aspx?qprid=2&qpcustomd=0
[6] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk, "QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2," [Online]. Available: https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02
[7] Half of Chrome Traffic Now Runs over QUIC, https://www.bizety.com/2016/01/26/half-of-chrome-to-google-trafficnow-run-over-quic/
[8] Google's massive grip on the web, http://techland.time.com/2013/08/19/when-google-goes-down-it-apparently-takes-40-of-internet-traffic-with-it/
[9] G. Carlucci, L. D. Cicco, S. Mascolo, "HTTP over UDP: an Experimental Investigation of QUIC", in ACM SAC' 15, April 13-17 2015, Salamanca, Spain.
[10] P. Megyesi, Z. Kramer, S. Molnar, "How quick is QUIC?", in Proc. of IEEE ICC 2016, 22-27 May 2016.
[11] Chromium project, https://cs.chromium.org
[12] HTTP/2 C library and tools, https://nghttp2.org
[13] Chrome ends spdy support , https://blog.chromium.org/2016/02/transitioning-from-spdy-to-http2.html
[14] Ninja tool, https://ninja-build.org/
[15] Quic protocol with chrome, https://www.chromium.org/quic/playing-with-quic
[16] Chrome/Chromium Enable quic, https://ma.ttias.be/enable-quic-protocol-google-chrome/
[17] Quic protocol, https://github.com/google/proto-quic
[18] Quic server material, https://groups.google.com/a/chromium.org/forum/#!topic/proto-quic/zvwT9QSuxIU
[19] Chrome/Chromium Install Chrome, https://askubuntu.com/questions/510056/how-to-install-google-chrome
[20] Certificates, https://chromium.googlesource.com/chromium/src/+/master/docs/linux_cert_management.md
[21] Indicators SPDY, https://blog.cloudflare.com/tools-for-debugging-testing-and-using-http-2/, https://chrome.google.com/webstore/detail/http2-and-spdy-indicator/mpbpobfflnpcgagjijhmgnchggcjblin
[22] Http2, https://http2.pro/doc/Apache
[23] Spdy software, https://developers.google.com/speed/spdy/mod_spdy/

[24] Spdy software coexistent with apache, https://devops.profitbricks.com/tutorials/how-to-install-mod_spdy-with-apache-on-ubuntu-1604-1/
[25] Apache mpm prefork vs worker vs event, https://serverfault.com/questions/383526/how-do-i-select-which-apache-mpm-to-use
[26] Google Sites for testing, https://sites.google.com
[27] Apache performance, https://www.datadoghq.com/blog/monitoring-apache-web-server-performance/
[28] Online tools for testing http vs http2, https://www.dareboost.com/en/website-speed-test-http2-vs-http1
[29] Online tools for testing har, https://toolbox.googleapps.com/apps/har_analyzer/, https://github.com/cyrus-and/chrome-har-capturer, https://github.com/parasdahal/speedprofile/
[30] Linux tc functionality, https://www.sitespeed.io/documentation/browsertime/, https://wiki.linuxfoundation.org/networking/netem
[31] The ping tool, http://www.tutorialspoint.com/unix_commands/ping.htm

B.3 Business impact

Describe in detail how this experiment may impact your business and product development.

B.3.1 Value perceived

What is the value you have perceived from this experiment (return on investment)?
E.g. gained knowledge; acquired new competences; practical implementation solutions such as scalability, reliability, interoperability; new ideas for experiments/products; etc.

8BELLS designed and executed the Go-Quick experiment for analysing and quantifying the performance of QUIC, as compared to HTTP and SPDY. The outcomes will be used as inputs to further analysis targeting to existing network operators and CSPs. Thus, the innovative approach of Go-Quick experiment consists a basis for a technical modelling framework in the existing company's portfolio.

What was the direct or indirect value for your company / institution? What is the time frame this value could be incorporated within your current product(s) range or technical solution? Could you apply your results also to other scenarios, products, industries?

There are various ways by which that Go-Quick experiment outcomes can be used directly for the benefit of the Company. For example, 8BELLS experimented using a small set of common well-documented experimenter tools which increased simplicity (since those tools can hide many of the testbeds' complexities), a single federated interface, and uniform input/output from different systems. All these benefits result in a lower entry barrier for a start-up like 8BELLS.
The time frame of incorporating the technical solution can be characterized as a "mid-term" basis. The results obtained could be adopted by several industries in the ICT domain apart from network operators, like for example content service providers, over the top players, equipment manufacturers, etc.

If no federation of testbed infrastructure would be available, how would this have affected your product / solution? What would have been the value of your product / solution if the experiment was not executed within Fed4FIRE? What problems could have occurred?

Go-Quick project leveraged on i2CAT's OpenFlow OFELIA testbed for the small-scale experiment on real equipment. 8BELLS has involved also in its business plan they study of network services and their interactions, therefore the company searches for cutting edge technology environments for transport layer applications. Therefore, 8BELLS is very satisfied that it had the opportunity to apply its methods for experimentation also in a testbed that belongs to the FIRE ecosystem.

Are there any follow-up activities planned by your company/institution? New projects or funding thanks to this experiment? Do you intend to use Fed4FIRE facilities again in the future?

The ambition of 8BELLS analysts was to find into the Fed4FIRE+ experimentation testbed an SDN-capable platform a diverse set of capabilities and experimentation tools, for testing and benchmarking the novel QUIC protocol against its predecessors, under various network conditions. A working group inside the company has already been established investigating all relevant parameters of such an ambitious action. Besides, the Fed4FIRE facilities could be used again, to the extend they are in favour of open source solutions.

B.3.2 Funding

Was the allocated budget related to the experiment to be conducted high enough (to execute the experiment, in relation to the value perceived, etc.)?

The budget allocated to the experiment, taking into account also the effort required for setting up and running the experiment can be characterized as tight. The number of travels required can also be decreased.

Did you receive other funding for executing this experiment besides the money from the Fed4FIRE open call (e.g. internal, national, …)?

No.

Would you (have) execute(d) the experiment without receiving any external funding?

No.

Would you even consider to pay for running such an experiment? If so, what do you see as most valuable component(s) to pay for (resources, support, …)?

No.

Section C Feedback to Fed4FIRE

This section contains valuable information for the Fed4FIRE consortium and describes your experiences by running your experiment on the available testbeds. Note that the production of this feedback is one of the key motivations for the existence of the Fed4FIRE open calls.

C.1 Resources & tools used

C.1.1 Resources

Describe the testbeds you have been using and specify the resources used.

Infrastructures	*Used?*	*Specify the type and amount of the resources used*
Wired testbeds		
• Virtual Wall (iMinds)		
• PlanetLab Europe (UPMC)		
• Ultra Access (UC3M, Stanford)		
Wireless testbeds		
• Norbit (NICTA)		
• w-iLab.t (iMinds)		
• NITOS (UTH)		
• Netmode (NTUA)		
• SmartSantander (UC)		
• FuSeCo (FOKUS)		
• PerformLTE (UMA)		
OpenFlow testbeds		
• UBristol OFELIA island		
• i2CAT OFELIA island	Yes	2 SDN OpenFlow-enabled NEC IP8800 switches in order to allow the deployment of virtual topologies over the physical network. 1 physical server for hosting VMs.
• Koren testbed (NIA)		
• NITOS testbed		
Cloud computing testbed		
• EPCC and Inria cloud sites (members of the BonFIRE multi-cloud testbed for services experimentation)		
• iMinds Virtual Wall testbed for emulated networks in BonFIRE		
Community testbeds		
• C-Lab (UPC)		

Did you make use of all requested testbed infrastructure resources, as specified in your open call proposal? If not, please explain.

Yes.

What was the ratio between time reserved vs time actually used for each resource? Why does it differ that much (e.g. for interference reasons, other)?

The value of the ratio is at about 1 so no big deviations have been observed from the initial planning.

C.1.2 Tools

Describe in detail the tools you have been using, resources used, how many nodes, …

Tools	*Used?*	*Please indicate your experience with the tools. What were the positive aspects? What didn't work?*
Fed4FIRE portal	Yes	Easy to use, stable
JFed	Yes	User friendly GUI, not so many flavours of Ubuntu VMs
Omni		
SFI		
BonFIRE portal		
BonFIRE API		
Ofelia portal		
OMF		
NEPI		
JFed timeline		
OML		
Please list below other tools used		

C.2 Feedback based on design/set-up/running your experiment on Fed4FIRE

Describe in detail your experiences concerning the procedure and administration, set-up, Fed4FIRE portfolio, documentation and support, experimentation environment, and experimentation execution and results. This feedback will help us for future improvement.

C.2.1 Procedure / Administration

How do you rate the level of work for administration / feedback / writing documents / attending conference calls or meetings compared to the timeframe of the experiment?

Normal

C.2.2 Setup of the experiment

How much effort was required to set up and run the experiment for the first time? Did you need to install additional components before your were able to execute the experiment (e.g. install hardware / software components)?

Normal. We needed to install another version of Linux of the one supported already.

How do your rate the experience as user that you only had to deal with a single service provider (i.e. single point of contact and service) instead of dealing with each testbed itself?

Very positive.

C.2.3 Fed4FIRE portfolio

Was the current portfolio of testbeds provided by the federation, with access to a large set of different technologies (sensors, computing, network, etc.) provided by a large amount of testbeds, sufficient to run your experiment?

Yes.

Was the technical offering in line with the expectations? What were the positive and negative aspects? Which requirements could not be fulfilled?

Yes, the technical offering was very much in line with what expected. 8BELLS does not have any negative comments on this.

Could you easily access the requested testbed infrastructures?

Yes.

Could you make use of all requested resources at the different testbeds as was proposed in the description of the experiment? If not, how many times did this fail? What were the main reasons it failed (e.g. timing constraints, technical failures, etc.)?

All requested resources were available and used as proposed in the description of the experiment.

Did you use a lot the combination of resources over different testbeds? Did it all work out nicely? Were they interoperable?

Go-Quick experiment took place into one NFV-PoP (OFELIA, i2CAT). So no combination of testbeds occurred.

C.2.4 Documentation and support

Was the documentation provided helpful for setting up and running the experiment? Was it complete? What was missing? What could be updated/extended?

Yes it was complete.

Did you make use of the first level support dashboard?

No.

Did you contact the individual testbeds for dedicated technical questions?

No.

C.2.5 Experiment environment

Was the environment trustworthy enough for your experiments (in terms of data protection, privacy guarantees of yourself and your experiment)?

Not sure if it is in accordance with GDPR framework.

Did you have enough control of the environment to repeat the experiment in an easy manner?

Yes.

Did you experience that the Fed4FIRE environment is unique for experimentation and goes beyond the lab environment and enables real world implementation?

Yes, the environment provided by Fed4FIRE is far more evolved that other similar initiatives.

Did you share your experiment and/or results with a wider community of experimenters (e.g. for greater impact of results, shared dissemination, possibility to share experience and knowledge with other experimenters)? If not, would you consider this in the future?

8BELLS will write a scientific paper to involve engineers and the wider community for its outcomes.

C.2.6 Experiment execution and results

Did you have enough time to conduct the experiment?

Yes, the time was adequate.

Were the results below / in line with / exceeding your initial goals and expectations?

The results were in line with the theoretical framework.

What were the hurdles / bottlenecks? What could not be executed? Was this due to technical limits? Would the federation or the individual testbeds be able to help you solving this problem in the future?

A hurdle / bottleneck has been that the testbed cannot access external servers through Internet.

C.2.7 Other feedback

If you have other feedback or comments not discussed before related to the design, set-up and execution of your experiment, please note them below.

None.

C.3 Why Fed4FIRE was useful to you

Describe why you chose Fed4FIRE for your experiment, which components were perceived as most valuable for the federation, and your opinion what your would liked to have had, what should be changed or was missing.

C.3.1 Execution of the experiment

Why did you choose Fed4FIRE for your experiment? Was it the availability of budget, easy procedure, possibility to combine different (geographically spread) facilities, access to resources that otherwise would not be affordable, availability of tools, etc.? Please specify in detail.

Being an SME, the openness provided by Fed4FIRE was a motivation. Also, the additional tools and the support provided during the experiment helped a lot.

Could you have conducted the experiment at a commercially available testbed infrastructure?

No.

C.3.2 Added value of Fed4FIRE

Which components did you see as highly valuable for the federation (e.g. combining infrastructures, diversity of available resources, tools offered, support and documentation, easy setup of experiments, etc.)? Please rank them in order of importance.

- jFED was very intuitive, user friendly and with many functionalities. With the combination of CLI an experimenter can easily fulfill his/her targets.

Which of these tools and components should the federation at least offer to allow experimentation without funding?

jFED is very attractive platform for experimentation.

C.3.3 What is missing from your perspective?

What would you have liked to have had within Fed4FIRE (tools, APIs, scripts, …)? Which tools and procedures should be adapted? What functionality did you really miss?

A traffic generator tool would help a lot.

Which (types of) testbed infrastructures (and resources) would have been very valuable for you as experimenter within the Fed4FIRE consortium?

For next experimentation, 8BELLS will target to 5G end to end networks.

Is there any other kind of support that you would expect from the federation, which is not available today e.g. some kind of consultancy service that can guide you through every step of the process of transforming your idea into an actual successful experiment and eventually helping you to understand the obtained results?

A consultancy service for entering the market would be of high value.

C.3.4 Other feedback

If you have further feedback or comments not discussed before how Fed4FIRE was useful to you, please note them below.

None.

C.3.5 Quote

We would also like to have a quote we could use for further dissemination activities. Please complete the following sentence.

Thanks to the experiment I conducted within Fed4FIRE ...

… 8BELLS will be able to come closer to its target audience.

Appendix A

Contents of the web content (test) generating script:

```
#!/bin/bash

# File must already be in a folder under quic-data/www.example.org
directory
# e.g. testGen -x -f 18MB/test.iso # just clean
# e.g. testGen -x -i /opt/goquic/images/testimage1MB.jpg -s /opt/quic-
data/www.example.org -f 1MB/testimage1MB.jpg -n 10

OPTIND=1
while getopts pxvf:s:i:g:n: OPTION; do
        case $OPTION in
                p) optparallel=1 ;; # not yet implemented
                               x) optclean=1 ;;
                               v) optverbose=1 ;;
                s) sitepath=$OPTARG ;;
                               i) inputfile=$OPTARG;;
                               f) filepath=$OPTARG ;;
                               g) optgenerateindexes=$OPTARG ;;
                n) numoffiles=$OPTARG ;;
                ?) echo "Usage: testGen [-p] [-x] [-s sitepath]  [-f
filepath] -n numoffiles"
                   exit 2 ;;
        esac
done
shift $(($OPTIND - 1))
#default values

[ "$sitepath" = "" ] && sitepath=/opt/quic-data/www.example.org
[ "$apachesitepath" = "" ] && apachesitepath=/opt/apache-
data/www.example.org

if ! [ -e "$sitepath" ]; then
      echo "Error : site path $sitepath not found, exiting .."
      exit 3
fi
if ! [ -e "$apachesitepath" ]; then
   echo "Error : site path $sitepath not found, exiting .."
   exit 4
fi

dirname=$(dirname $sitepath/$filepath)
apachedirname=$(dirname $apachesitepath/$filepath)

mkdir -p $dirname $apachedirname
cp $inputfile $apachesitepath/$filepath

if ! [ -e "$inputfile" ]; then
   echo "Error : file $inputfile not found, exiting .."
   exit 5
elif ! [ -e "$apachesitepath/$filepath" ]; then
   echo "Error : file $apachesitepath/$filepath not found, exiting .."
   exit 6
else
      if [ "$optclean" ]; then
      fullpath="$sitepath/$filepath"
            apachefullpath="$apachesitepath/$filepath"
```

```
            echo "cleanup files $dirname/* ${apachefullpath%.*}-*"

            #rm -rf $fullpath.*
            rm -rf ${apachefullpath%.*}-* $dirname/*
            ! [ "$numoffiles" ] && exit 0 # only clean
      else
            if [ "$numoffiles" ]; then
                  echo "Error! Please supply -n numoffiles, exiting .."
                  exit 7 # we were supposed to generate files
            fi
      fi
fi

mkdir -p $dirname
filesmsg="index.html"
for i in $optgenerateindexes; do
      filesmsg="$filesmsg, index$i.html"
done
echo -e "generate $filesmsg\nand $numoffiles files from $(basename
$inputfile) in folders $dirname and $apachedirname"
for i in $(seq 1 $numoffiles);do
      fullpath="$sitepath/$filepath"
   apachefullpath="$apachesitepath/$filepath"
      dirname=$(dirname $fullpath)
      apachedirname=$(dirname $apachefullpath)
      filename=$(basename $fullpath)
      filepref=${filename%.*}
      filesuf=${filename##*.}
      newfile=$apachedirname/$filepref-$i.$filesuf
      newfilequic=$dirname/$filepref-$i.$filesuf
      fileshtml="$fileshtml <img width=\"20%\" height=\"20%\"
src=\"https://www.example.org/${filepath%.*}-$i.${filepath##*.}\">
$filepref-$i.$filesuf"
      #newfile=$sitepath/$filepath.$i

      if [ "$(echo $optgenerateindexes|grep -w $i)" ]; then
            cp /opt/goquic/index.html $apachedirname/index$i.html
            sed -i "s#FLAGFILESFLAG#$fileshtml#g"
$apachedirname/index$i.html
            wget -A.html -q --no-check-certificate -p --save-headers
https://www.example.org/$(dirname $filepath)/index$i.html -O
$dirname/index$i.html
            sed -i "\#Connection:#i X-Original-Url:
https://www.example.org/$(dirname $filepath)/index$i.html"
$dirname/index$i.html
      fi
      wget -q --no-check-certificate -p --save-headers
https://www.example.org/$filepath -O $newfilequic
      #cp $sitepath/$filepath $newfilequic;
   cp $apachesitepath/$filepath $newfile;

   sed -i "\#Connection:#i X-Original-Url:
https://www.example.org/${filepath%.*}-$i.${filepath##*.}" $newfilequic;
   [ "$optverbose" ] && diff -du -a $sitepath/$filepath $newfile
done

#Generate index.html
indexhtmlfile=$apachesitepath/$(dirname $filepath)/index.html
indexhtmlfilequic=$sitepath/$(dirname $filepath)/index.html
```

```
cp /opt/goquic/index.html $indexhtmlfile
sed -i "s#FLAGFILESFLAG#$fileshtml#g" $indexhtmlfile
wget -A.html -q --no-check-certificate -p --save-headers
https://www.example.org/$(dirname $filepath)/index.html -O
$indexhtmlfilequic

sed -i "\#Connection:#i X-Original-Url: https://www.example.org/$(dirname
$filepath)/index.html" $indexhtmlfilequic;

echo
doService -s quic -c restart
```

Appendix B

Contents of the "doService" script:

```
#!/bin/bash

progname=$0

OPTIND=1
while getopts s:c: OPTION; do
        case $OPTION in
               s) srv=$OPTARG ;;
                                c) cmd=$OPTARG ;;
        esac
done
shift $(($OPTIND - 1))

case $srv in
      quic)
                  case $cmd in
                        start)
                                        /opt/proto-
quic/src/out/Default/quic_server --quic_response_cache_dir=/opt/quic-
data/www.example.org   --certificate_file=/opt/proto-
quic/src/net/tools/quic/certs/out/leaf_cert.pem --key_file=/opt/proto-
quic/src/net/tools/quic/certs/out/leaf_cert.pkcs8&    ;;
                        stop)
                                        killall quic_server ;;
                        restart)
                                        $progname -s $srv -c stop
                                        sleep 2
                                        $progname -s $srv -c start    ;;
                  esac ;;
       https)
                  case $cmd in
            start|restart)
                                a2dismod http2; a2dismod spdy; service
apache2 restart  ;;
            stop)
                  service apache2 stop ;;
         esac     ;;
       "https/2")
                  case $cmd in
            start|restart)
               a2enmod http2; a2enmod spdy; service apache2 restart  ;;
            stop)
               service apache2 stop ;;
         esac  ;;

esac
```

Appendix C

Contents of the "getFiles" script:

```
#!/bin/bash

srv=$1
client=$2
rtimes=$3

[ "$rtimes" = "" ] && rtimes=3
if [ ""$srv"" == "" ]; then
      echo please call as $0 service , where service : quic/https
      exit 1
fi

sep1="\n\n\n\n"
sep2="\n\n\n\n"
sep3="\n\n"
resultsfile="results.txt"

if [ "$client" == "chrome" ]; then
      for i in 1 2 4 8 10 15 20; do
            getFiles -x -t $rtimes -b Mb -e $resultsfile -c $client -s
https -u https://www.example.org/10MB/index$i.html -d 10MB
      done
      echo -n -e "$sep1" >> $resultsfile
      for i in 1 2 4 8 10 15 20 30 40 50 60 80 100; do
            getFiles -x -t $rtimes -b Mb -e $resultsfile -c $client -s
https -u https://www.example.org/1MB/index$i.html -d 1MB
      done
      echo -n -e "$sep2" >> $resultsfile
      for i in 1 2 4 8 10 15 20 30 40 50 60 80 100 200 500 1000; do
            getFiles -x -t $rtimes -b Mb -e $resultsfile -c $client -s
https -u https://www.example.org/100KB/index$i.html -d 100KB
      done
      echo -n -e "$sep3" >> $resultsfile
   for i in 2 4 8 10 15 20 30 40 50 60 80 100 200 500 1000; do
      getFiles -x -t $rtimes -b Mb -e $resultsfile -c $client -s https -u
https://www.example.org/10KB/index$i.html -d 10KB
   done
      exit 0
fi

#10MB
for numoffiles in 1 2 4 8 10 15 20; do
        getFiles -x -p all -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/10MB/testimage10MB.png -d 10MB -n
$numoffiles
done
echo -n -e "$sep1" >> $resultsfile

#1MB
for numoffiles in 1 2 4 8 10 15 20 30 40; do
        getFiles -x -p all -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/1MB/testimage1MB.jpg -d 1MB -n
$numoffiles
done
getFiles -x -p 30 -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/1MB/testimage1MB.jpg -d 1MB -n 50
```

```
getFiles -x -p 30 -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/1MB/testimage1MB.jpg -d 1MB -n 60
getFiles -x -p 30 -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/1MB/testimage1MB.jpg -d 1MB -n 80
getFiles -x -p 30 -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/1MB/testimage1MB.jpg -d 1MB -n 100
echo -n -e "$sep2" >> $resultsfile

#100KB
for numoffiles in 1 2 4 8 10 15 20 30 40 50 60; do
        getFiles -x -p all -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/100KB/testimage100KB.jpg -d 100KB
-n $numoffiles
done
getFiles -x -p 20 -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/100KB/testimage100KB.jpg -d 100KB
-n 80
getFiles -x -p 20 -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/100KB/testimage100KB.jpg -d 100KB
-n 100
getFiles -x -p 20 -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/100KB/testimage100KB.jpg -d 100KB
-n 200
getFiles -x -p 15 -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/100KB/testimage100KB.jpg -d 100KB
-n 500
getFiles -x -p 15 -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/100KB/testimage100KB.jpg -d 100KB
-n 1000
echo -n -e "$sep3" >> $resultsfile

#10KB
for numoffiles in 2 4 8 10; do
        getFiles -x -p all -t $rtimes -b Mb -e $resultsfile -s "$srv" -i
192.168.20.30 -u https://www.example.org/10KB/testimage10KB.jpg -d 10KB -n
$numoffiles
done
```

Appendix D

Contents of the "rate-limiting" script:

```
#!/bin/bash

# To set max speed to 1Mbit/sec on interface ens18 (TODO detect
automatically or by IP) on port 6121
# e.g. trafficGen -c restart -s 1 -i ens18 -p 6121
# To clear rules
# e.g. trafficGen -c stop
# To show status
# e.g. trafficGen -c show



OPTIND=1
while getopts xvc:s:i:p: OPTION; do
        case $OPTION in
                x) optclean=1 ;;
                v) optverbose=1 ;;
                                c) cmd=$OPTARG ;;
                s) speed=$OPTARG ;;
                i) iface=$OPTARG ;;
                p) port=$OPTARG ;;
                ?) echo "Usage: trafficGen -c start -s maxspeed -i iface -p
port"
                   exit 2 ;;
        esac
done
shift $(($OPTIND - 1))


if [ "$iface" == "" ]; then
      echo "Error : -i interface needed"
   exit 3
fi

if [ "$cmd" == "stop" ] || [ "$cmd" == "restart" ]; then
      /sbin/tc qdisc del dev $iface root
fi


if [ "$cmd" == "show" ]; then
#     watch /sbin/tc -s qdisc ls dev $iface
      watch /sbin/tc -s -d class show dev $iface
fi


if [ "$speed" = "" ]; then
      if [ "$cmd" == "start" ] || [ "$cmd" == "restart" ] ; then
            echo "Error : -s speed needed"
            exit 4
      else
            exit 0
      fi
fi
if [ "$port" = "" ]; then
   if [ "$cmd" == "start" ]  || [ "$cmd" == "restart" ] ; then
      echo "Error : -p port needed"
```

```
      exit 5
   else
      exit 0
   fi
fi

ip="$(ifconfig $iface | grep 'inet addr:' | cut -d: -f2 | awk '{ print
$1}')"

if [ "$cmd" == "start" ] || [ "$cmd" == "restart" ] ; then
      #speedKB=$(echo "scale=0; $speed * 125" | bc)
      #speedKBceil=$(echo "scale=0; $speedKB * 1.1" | bc)
      speedceil=$(echo "scale=0; $speed * 1.1" | bc)

      #setup bandwidth limit (1st way - not working yet)
      #/sbin/tc qdisc add dev $iface root handle 1: htb
      #/sbin/tc class add dev $iface parent 1: classid 1:2 htb rate
1024kbps #set 1Gibabit parent speed (it is enough)
      #/sbin/tc class add dev $iface parent 1:2 classid 1:5 htb rate
$speed""mbit ceil $speedceil""mbit prio 0
      #/sbin/tc filter add dev $iface parent 1:0 prio 0 protocol ip handle
6 fw flowid 1:5
      #/sbin/iptables -A INPUT -t mangle -p tcp --sport $port -j MARK --
set-mark 6 #INPUT if called on server

      #etup bandwidth limit (2nd way)
      U32="/sbin/tc filter add dev $iface protocol ip parent 1:0 prio 1
u32"

      /sbin/tc qdisc add dev $iface root handle 1: htb default 30
      /sbin/tc class add dev $iface parent 1: classid 1:1 htb rate
$speed""mbit
      /sbin/tc class add dev $iface parent 1: classid 1:2 htb rate
$speed""mbit
      $U32 match ip dst $ip/32 flowid 1:1
      $U32 match ip src $ip/32 flowid 1:2
fi
```