A Real-Time Video Streaming System over IPv6+MPTCP Technology

Yu Luo, Xing Zhou, Thomas Dreibholz, and Hanbao Kuang

Abstract Today, a steadily increasing number of users are not just passively consuming Internet content, but also share and publish content. Users publish text, photos and videos. With the availability of 5G high-speed, low-latency mobile broadband networks, real-time video streaming will also be possible. We believe this will become a very popular application in the coming years. But the more popular a service is, the higher the need for resilience. In this paper, we introduce our work-in-progress live video streaming platform for future mobile edge computing scenarios, which makes use of MPTCP+IPv6 to support multi-homing for resilience and multi-path transport for load balancing. As a proof of concept, we will show that the platform is (1) compatible with IPv6, (2) utilizes load balancing when possible and (3) provides robustness by network redundancy.[1][2]
Keywords: Video Streaming Platform, IPv6, MPTCP, Load Balancing, Resilience

1 Introduction

With the rapid development of the Internet, more and more real-time services are deployed in the network, such as live video broadcast, video conferencing and other key real-time businesses. These applications have stricter requirements for bandwidth, latency and jitter. 5G networks, together with cloud infrastructure for mobile edge computing (MEC), are a key enabler for such applications. But in addi-

Yu Luo, Xing Zhou (corresponding author), Hanbao Kuang
Hainan University, College of Information Science and Technology, Haikou, Hainan, China,
e-mail: 18636038948@163.com,xingzhou50@126.com,1521793003@qq.com

Thomas Dreibholz
SimulaMet, Oslo, Norway, e-mail: dreibh@simula.no

[1] This work has been funded by the NSFC of China (No. 61363008/61662020), CERNET NGI Technology Innovation Project (No. NGII20160110) and the Research Council of Norway (Forskingsrådet) (prosjektnummer 208798/F50).

[2] The authors would like to thank Ted Zimmerman for his comments.

tion, there is also need for network and transport improvements: Today, IPv4 addresses are a scarce resource, compared with new applications like the Internet of Things (IoT). IPv6 is therefore an important development step of the next generation Internet. With the rapid development of the IoT, various network services and applications based on IPv4 will definitely migrate to IPv6 [26]. But the single-path transmission patterns of TCP cannot handle multiple addresses per endpoint or address changes, which are usual for IPv6. Therefore, another milestone is Multi-Path TCP (MPTCP) [16, 24], providing transport redundancy by multi-homing without changing the existing network infrastructure. It also supports load balancing over heterogeneous networks connections [15, 17, 27, 30]. But MPTCP is not only useful when an endpoint is connected to multiple Internet connections. [20, 22] measured and compared the stability of IPv4 and IPv6 Internet paths. With IPv4/IPv6 dual-stack endpoints, MPTCP may already provide additional redundancy by just using non-congruent IPv4 and IPv6 paths.

In this paper, we introduce our work in progress on a live video streaming service based on MPTCP and IPv6, as a proof-of-concept for a resilient service in MEC. We furthermore present initial evaluation results.

2 Overview of Relevant Technologies

The major limitation of IPv4 is its limited address space: with its 32-bit addresses, it theoretically provides almost 4.3 billion (2^{32}) unique Internet addresses, while only around 3.7 billion addresses are usable host addresses. IPv6 [3] therefore extends the address space to 128 bits, theoretically allowing around 340 undecillion (2^{128}) addresses. This allows for a very systematic assignment of subnets [26]. IPv6 furthermore provides some improvements for more efficient handling of packets [2], like an optimized header format.

TCP establishes a connection between two IPv4 or IPv6 addresses. Port numbers are used to identify the connection, i.e. the 4-tuple (source address/port, destination address/port) uniquely identifies the connection. TCP neither supports multiple address per endpoint nor address changes during connection lifetime. The MPTCP extension [16] adds this multi-homing feature. Addresses are negotiated during connection establishment. Furthermore, addresses may change during connection lifetime. MPTCP supports IPv4 and IPv6; boths protocols can be used simultaneously in the same connection. Furthermore, MPTCP can utilize multiple subflows simultaneously (multi-path transport) and balance load among the paths.

MPTCP establishes so-called subflows, which look – for middlebox devices like firewalls – like regular TCP flows. MPTCP therefore works in existing networks, with existing middlebox devices. That is, unlike for other protocols like the Stream Control Transmission Protocol with Concurrent Multipath Transfer extension (CMT-SCTP, [5]), it is compatible with old networking devices. Only the two endpoints of an MPTCP connection need MPTCP support. This ensures a smooth deployment [1, 15, 18]. MPTCP provides the same Socket API interface [19, 25] to the Application Layer as standard TCP. That is, applications interact with MPTCP

Socket API Function	Functional Description
inet_pton()	Convert address string to address
inet_ntop()	Convert address to address string
gethostbyname()	Translate name/address string to IP address(es) – deprecated!
gethostbyaddr()	Translate IP address to name/address string – deprecated!
getaddrinfo()	Translate name/address string to IP address(es)
freeaddrinfo()	Free memory allocated by getaddrinfo()
getnameinfo()	Translate IP address to name/address string
socket()	Create socket
bind()	Bind socket to address
listen()	Put socket into listening mode (TCP, SCTP)
accept()	Accept incoming connection (TCP, SCTP)
connect()	Establish new connection
send()	Send data (TCP, SCTP)
sendto()	Send data (UDP)
recv()	Receive data (TCP, SCTP)
recvfrom()	Receive data (UDP)
close()	Close socket

Table 1 Socket API Functions and IPv6 Communication

as they would do with TCP. In most cases, the application does not even have information about an underlying MPTCP instead of TCP.

3 Using Software with IPv6 and MPTCP

In the following, we introduce how to adapt systems with IPv6 and MPTCP support.

3.1 Making Software Compatible with IPv6

Network and Transport Layer protocols are used in applications by the operating system's Socket API [19, 25]. The basic functions of the Socket API are summarized in Table 1. These functions can be grouped into address handling and protocol handling. In the first group, the function

```
int inet_pton(int family, const char* src, void* dst)
```

converts the string `src` to the IP address `dst`. `dst` is addr_in for IPv4, or addr_in6 for IPv6; the type depends on the `family` parameter setting: AF_INET (IPv4) or AF_INET6 (IPv6). The reverse direction is provided by

```
const char* inet_ntop(int family, const void* src,
                      char* dst, socklen_t size)
```

Listing 1 The addrinfo Structure

```
struct addrinfo {
   int              ai_flags;
   int              ai_family;
   int              ai_socktype;
   int              ai_protocol;
   socklen_t        ai_addrlen;
   struct sockaddr* ai_addr;
   char*            ai_canonname;
   struct addrinfo* ai_next;
};
```

It converts the IP address `src` to the string `dst`, with a maximum length of `size`. Note, both functions only handle addr_in and addr_in6 – *not* sockaddr_in and sockaddr_in6. The application therefore has to assemble the latter structures by itself! Similar to converting strings and addresses, it is also possible to resolve DNS names and reverse-lookup DNS names for IP addresses. This is provided by the functions gethostbyname() (DNS name resolution, like e.g. www.nntb.no) and gethostbyaddr() (DNS reverse lookup). Originally, these functions have only been defined for IPv4. Most operating systems, however, provide extended IPv6-capable functions as well. This makes platform-specific code in multi-platform applications necessary. Furthermore, gethostbyname() and gethostbyaddr() are not thread-safe, creating further platform-specific variants. Therefore, the usage of these functions is now deprecated. gethostbyname() and gethostbyaddr() have been replaced by new functions, which also handle the address/string conversion provided by inet_pton() and inet_ntop(), in a platform-independent way.

The getaddrinfo() function provides the translation of an address or DNS name – as well as a port number or service name – into IP addresses, e.g. sockaddr_in (IPv4) or sockaddr_in6 (IPv6) [19, 25] as necessary:

```
int getaddrinfo(const char*           host,
                const char*           serv,
                const struct addrinfo* hints,
                struct addrinfo**      res)
```

That is, getaddrinfo() takes an address or DNS name string as `host` parameter, a port number string or service name as `serv` parameter, an addrinfo structure as `hints` parameter, as well as a pointer `res` to store the result (in form of an addrinfo structure allocated by getaddrinfo()). The corresponding structure addrinfo is shown in Listing 1. The `hints` structure can provide some specific information for the address structure to be created, particularly the ai_family (AF_INET or AF_INET6 – but in most cases AF_UNSPEC to support any IP protocol). Furthermore two flags in ai_flags can enforce the usage of IP address only (AI_NUMERICHOST, i.e. no DNS name allowed) and port number only (AI_NUMERICSERV, i.e. no service name allowed). We will later make this clearer with examples. The resulting addrinfo structure contains the address in ai_addr (i.e. sockaddr_in for IPv4 or sockaddr_in6 for IPv6) and its length in ai_addrlen. If there are multiple addresses (e.g. an IPv6

Listing 2 An Example for getaddrinfo() with Address String and Port

```
const char*      addressString = "2001:700:4100:2::179";
const char*      portString    = "443";
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));
hints.ai_family   = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags    = AI_NUMERICHOST|AI_NUMERICSERV;
struct addrinfo* result = NULL;
if(getaddrinfo(addressString, portString,
               (const struct addrinfo*)&hints, &result) == 0) {
   // success; address is in result->ai_addr!
   freeaddrinfo(result);
}
```

and an IPv4 address), addrinfo structures may be chained using a pointer to the next addrinfo structure in ai_next. Since getaddrinfo() allocates storage space for the resulting addrinfo structure(s), it is necessary to free them. This is realized by calling

```
void freeaddrinfo(struct addrinfo* res)
```

The reverse direction, i.e. address to string or DNS name, is provided by:

```
int getnameinfo(const struct sockaddr* addr, socklen_t addrlen,
                char*                  host, socklen_t hostlen,
                char*                  serv, socklen_t servlen,
                int                    flags)
```

It takes a sockaddr_in (IPv4) or sockaddr_in6 (IPv6) structure in `addr`, its length in `addrlen` and flags in `flags`. The result is written to buffers provided by `host` for address string/hostname (with maximum length `hostlen`) and `host` for port number string/service name (with maximum length `servlen`). Similar to getaddrinfo(), the flag NI_NUMERICHOST turns off DNS reverse lookup (the result will be an address string); NI_NUMERICSERV turns off service name lookup (the result will be a port number string). We will make this clearer with examples below.

The Socket API functions to create, bind and connect sockets, as well as to write to and receive from addresses, are using sockaddr_in (for IPv4) and sockaddr_in6 (for IPv6). Since these basics are part of various publications, like e.g. [25], we do not go into details here. To write protocol-independent code, it is therefore useful to let getaddrinfo() create all protocol-specific information and let getnameinfo() make this information human-readable again. Ideally, the application itself does not have to contain any IP-protocol-specific code.

In order to make the explanations above clearer, Listing 2 shows an example to translate an IPv6 address string and a port number string to the corresponding sockaddr_in6 structure. getaddrinfo() obtains this information, in form of an addrinfo structure (see Listing 1), pointing to the actual sockaddr_in6 structure in ai_addr.

Working with Internet applications, some servers are already using IPv6, while some others still use IPv4. It is therefore convenient to use DNS names instead

Listing 3 An Example for getaddrinfo() with DNS Name and Service String

```
const char*      nameString    = "www.nntb.no";
const char*      serviceString = "https";
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));
hints.ai_family   = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
struct addrinfo* result   = NULL;
if(getaddrinfo(nameString, serviceString,
   (const struct addrinfo*)&hints, &result) == 0) {
   for(const struct addrinfo* ai = result;
       ai != NULL; ai = ai->ai_next) {
      // Iterate over all addresses returned
   }
   freeaddrinfo(result);
}
```

Listing 4 An Example for getnameinfo()

```
const struct sockaddr* address       = result->ai_addr;
const socklen_t        addressLength = result->ai_addrlen;
char                   addressString[NI_MAXHOST];
char                   portString[NI_MAXSERV];
if(getnameinfo(address, addressLength,
               (char*)&addressString, sizeof(addressString),
               (char*)&portString, sizeof(portString),
               NI_NUMERICHOST|NI_NUMERICSERV) == 0) {
   printf("Address: %s, Port: %s\n", addressString, portString);
}
```

of addresses. The example in Listing 3 therefore shows how to use getaddrinfo() with DNS name ("www.nntb.no") and service name ("https") instead. Calling getaddrinfo() is similar to Listing 2, except for *not* setting the AI_NUMERICHOST and AI_NUMERICSERV flags (since we want to use DNS and service lookup now). Since hosts may have multiple addresses in the DNS, the example also iterates over all returned addresses chained by the ai_next pointer of the addrinfo structures. All sockaddr_in and sockaddr_in6 will contain the port number of service "https" (which is 443 – resolved from `/etc/services`).

Finally, Listing 4 presents the usage of getnameinfo() to translate an address (here: as result from getaddrinfo()) to a string, including the port number. Obviously, not setting the flags NI_NUMERICHOST and NI_NUMERICSERV would try to reverse-lookup the DNS name and convert the port number to a service name.

3.2 Configuration for MPTCP

In order to use MPTCP on a Linux system, it is necessary to install an MPTCP-capable kernel [29]. Currently, the mainline Linux kernel does not provide MPTCP, so it is necessary to build a custom one based on the MPTCP developers' patch[3].

In addition to using a kernel with MPTCP, it is furthermore necessary to extend the routing configuration. Usually, all outgoing packets to an Internet destination take a configured default route. This is, on a system being connected to two different networks, all packets take the default route over one of the networks (the one with the lowest metric). The other one is not used, unless the primary one becomes unavailable. This is not desirable for MPTCP, which should use both networks for load balancing with multi-path transport. It is therefore necessary to configure multiple IP routing tables and select one of these tables for a routed packet based on its source address.

To give an example, let a system have two IPv6 addresses *A* (in Network 1, via interface NIC1) and *B* (in Network 2, via interface NIC2). Let both networks be /64 networks for simplicity. Network 1 has router *G*1 as default gateway, Network 2 has router *G*2 as default gateway. Therefore, two routing tables (1000 and 2000 for Network 1 and Network 2) are configured:

```
ip -6 rule add from $A table 1000
ip -6 rule add from $B table 2000
ip -6 route add $A/64 scope link dev $NIC1 table 1000
ip -6 route add default via $G1  dev $NIC1 table 1000
ip -6 route add $B/64 scope link dev $NIC2 table 2000
ip -6 route add default via $G2  dev $NIC2 table 2000
```

The first two lines configure the IP routing rules. If a packet is from source address *A*, routing table 1000 is used. If it is from *B*, routing table 2000 is used. The following lines configure local network and default route for both tables. Now, MPTCP is able to send via both networks – by just using different source addresses.

4 Our Live Video Streaming System

For our live video system, we choose to use the Real-Time Messaging Protocol (RTMP) [23] over TCP. For this protocol, there is already a widely used Open Source implementation available: SRS[4]. Particularly, SRS supports the structure of components we need:

- Push: the source of the live video. This could e.g. be a mobile user, filming with his phone camera.
- Server: a server taking the video from the Push user, possibly adapting it and distributing streams to viewers. In particular, this can be a cloud service in an

[3] MPTCP Git repository: https://github.com/multipath-tcp/mptcp.

[4] SRS Git repository: https://github.com/ossrs.

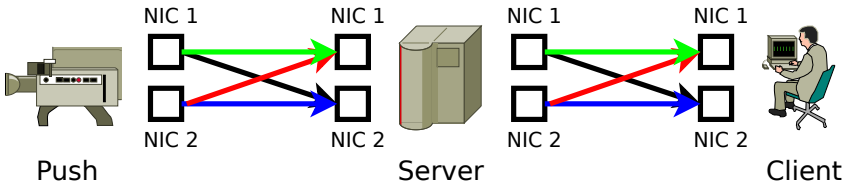

Fig. 1 The Structure of the Streaming Platform

MEC setup, to provide low-latency processing, so that viewers get an (almost) live stream.
- Client: a viewer, receiving the media stream from the server.

However, SRS was solely based on TCP over IPv4, i.e. without support for IPv6. As part of the work for this paper, we added IPv6 support as well. The changes for this support have been submitted as pull request[5] to the upstream project. It has already been merged into the upstream SRS sources, i.e. IPv6 support is now available for *all* SRS users. Basically, the idea of the changes for SRS was to replace all IPv4-specific code by IP-protocol-independent code based on getaddrinfo() and getnameinfo() (see Section 3). That is, getaddrinfo() is now used to convert address strings (see Listing 2) or resolve host names (see Listing 3) to either sockaddr_in (IPv4) or sockaddr_in6 (IPv6). Then, the application code can create sockets of the right family (AF_INET or AF_INET6 – returned by getaddrinfo() in ai_family of the addrinfo structure; see Listing 1). In the reverse direction, socket addresses are translated into strings by getnameinfo() (see Lising 4). By making the code IP-protocol-independent, we added IPv6 support. This not only simplified the original code, but also made it more portable as well.

We furthermore adapted `bwm-ng`[6], a tool to record the data statistics of network interfaces, with an improved accuracy of the recorded information. This change had also been provided as pull request[7] and been merged by the upstream project.

Finally, for the video display at the client, we use `ffplay`. It is a playback component of the well-known FFMPEG[8] tools.

5 Proof of Concept Evaluation

To evaluate the live broadcast system with SRS over MPTCP, we have set up the scenario as shown in Figure 1: it consists of RTMP Push, Server and Client. In the Push-Server network, as well as in the Server-Client network, each device is

[5] IPv6 support for SRS: https://github.com/ossrs/srs/pull/988 (pull request).

[6] `bwm-ng` Git repository: https://github.com/vgropp/bwm-ng.

[7] CSV file output improvement: https://github.com/vgropp/bwm-ng/pull/20 (pull request).

[8] FFMPEG: https://www.ffmpeg.org.

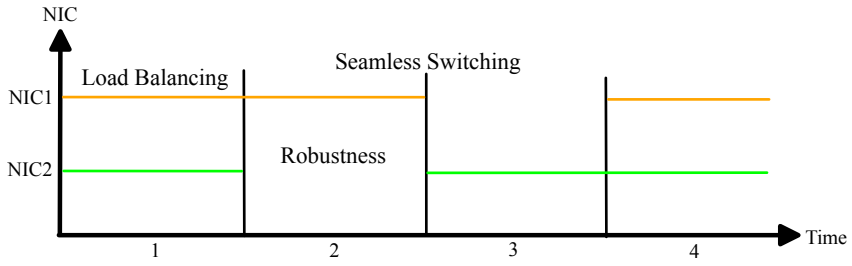

Fig. 2 Testing Scenario Sequence Diagram

equipped with two network interface cards (NIC). Each NIC can be configured with an IPv4 or IPv6 address. Between Push and Server, as well as between Server and Client, MPTCP can therefore establish 4 subflows. We consider NICs corresponding to different Internet service providers (ISP), i.e. all NIC1s correspond to ISP1 and all NIC2s correspond to ISP2. Then, the setup corresponds to a 2-ISP mobile edge setup, with the Push device e.g. a user sending a live video stream, the Server device a media server in a (mobile edge) computing centre and the Client device another user watching the live video. The video is observed by a human user, to check for disruptions and quality problems. To assess the performance of our setup, we apply the testing sequence as shown in Figure 2. It consists of 4 phases: (1) load balancing with all interfaces up; (2) ISP2 is down (all traffic for NIC2 of *all* devices is blocked); (3) ISP1 goes down, at the same time ISP2 comes up (all traffic of the NIC1s blocked); (4) load balancing with all interfaces up again. With these phases, we want to mimic simple handover and multi-ISP load balancing in a multi-homed mobile edge scenario. In the following measurement, each phase lasts 25s. For each NIC, the outgoing and incoming amount of data is recorded by using `bwm-ng`. On the Client device, `ffplay` is used to display the streamed video.

Figure 3 presents the IPv6 results for the Push side (outgoing data rate) and Client side (incoming data rate). In the first 25 s, load balancing is performed by MPTCP. As intended, MPTCP distributes the traffic to both networks. The outage of ISPs at t=25 s (by blocking all NIC1s) is properly handled by using the other network. The data rate there increases, since the video quality remains unchanged. At t=50 s, the ISPs change (ISP1 down, ISP2 up again). Particularly at the Client side, it is shown that it takes a moment for the system to detect the working ISP2 network and to move all traffic to the NIC2s. Once MPTCP detects that the NIC2s can be used, instead of the NIC1s, the RTMP system can quickly fill up the client's playback buffer (see the data rate peak). The video client at the Client device uses a buffer of 64 MiB, i.e. this playback buffer is able to cover the time needed by MPTCP for changing the networks. The human user did not notice a disruption of his video. At t=75 s, the blocking of the NIC1s is ended. The system went back to load balancing as in the first phase.

In summary, we can show that (1) MPTCP+IPv6 can provide load balancing in multi-ISP real-time streaming setups; (2) MPTCP+IPv6 can support seamless

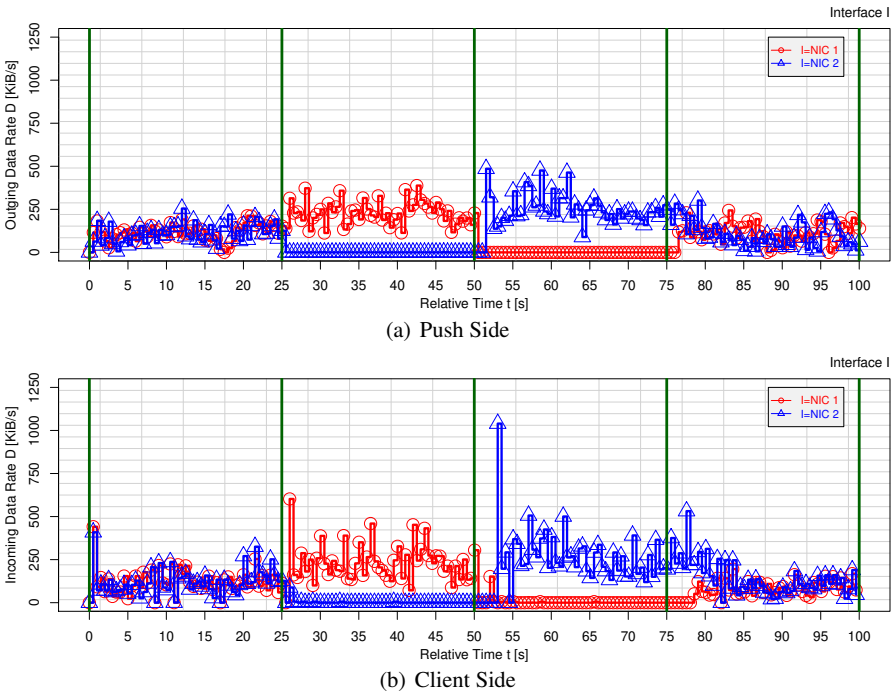

Fig. 3 Outgoing Per-Interface Data Rate with IPv6

handover in failure scenarios; (3) There is good compatibility among IPv6 in the Network Layer, MPTCP in the Transport Layer and RTMP in the Application Layer. The configuration therefore seems to be a reasonable choice for live real-time streaming with mobile edge computing.

6 Conclusions and Future Work

This paper introduced our live broadcast platform based on RTMP, MPTCP and IPv6. As a proof of concept, we showed that the system is able to handle typical load balancing, robustness and handover situations of multi-ISP scenarios.

As ongoing work on video transport over MPTCP with IPv6, we are now going to test and evaluate our setup in realistic, larger-scale Internet setups by using the NorNet Core testbed [6, 7, 9, 21]. The NorNet Core testbed consists of test nodes being located at different sites in multiple countries. Test applications can experience the same quality of service (QoS) as "normal" Internet users. The NorNet Core setup allows tests under realistic Internet conditions. We particularly plan to also combine it with MELODIC (Multi-Cloud Execution-Ware for Large-scale

Optimised Data-Intensive Computing) [10, 13], to support the cloud infrastructure for mobile edge computing. MELODIC is a middleware for application deployment in multi-cloud setups. Our idea is therefore to use MELODIC to provide a scalable Server-side setup in the MEC cloud (e.g. for video transcoding, caching and editing). The resources can then be instantiated on demand and multi-cloud support provides cost-effective deployment without vendor lock-in.

Furthermore, we want to apply NEAT (New, Evolutive API and Transport-Layer architecture for the Internet) technology [8, 11, 28]. The goal is to allow network and transport "services" provided to applications – such as reliability, low-delay communication or security – to be dynamically tailored based on application demands, current network conditions, hardware capabilities or local policies. For example, NEAT may choose to use advanced technologies like multi-path transport with MPTCP and IPv6 where available, but fall back to regular TCP in legacy networks.

Finally, with the application of Reliable Server Pooling (RSerPool) [4, 12, 14], it would also be possible to achieve high-availability by redundant Server-instances. RSerPool could then also be used to balance sessions between several Server-instances.

References

1. Becke, M., Dreibholz, T., Adhari, H., Rathgeb, E.P.: On the Fairness of Transport Protocols in a Multi-Path Environment. In: Proceedings of the IEEE International Conference on Communications (ICC), pp. 2666–2672. Ottawa, Ontario/Canada (2012)
2. Chen, J.L., Chao, H.C., Kuo, S.Y.: IPv6: More than protocol for next generation Internet. Computer Communications **29**, 3011–3012 (2006)
3. Deering, S.E., Hinden, R.M.: Internet Protocol, Version 6 (IPv6) Specification. Standards Track RFC 2460, IETF (1998)
4. Dreibholz, T.: Reliable Server Pooling – Evaluation, Optimization and Extension of a Novel IETF Architecture. Ph.D. thesis, University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems (2007)
5. Dreibholz, T.: Evaluation and Optimisation of Multi-Path Transport using the Stream Control Transmission Protocol. Habilitation treatise, University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems (2012)
6. Dreibholz, T.: NorNet – Building an Inter-Continental Internet Testbed based on Open Source Software. In: Proceedings of the LinuxCon Europe. Berlin/Germany (2016)
7. Dreibholz, T.: NorNet – The Internet Testbed for Multi-Homed Systems. In: Proceedings of the Multi-Service Networks Conference (MSN, Coseners). Abingdon, Oxfordshire/United Kingdom (2016)
8. Dreibholz, T.: A Practical Introduction to NEAT at Hainan University. Invited Talk at Hainan University, College of Information Science and Technology (CIST) (2017)
9. Dreibholz, T.: An Introduction to Multi-Path Transport at Hainan University. Keynote Talk at Hainan University, College of Information Science and Technology (CIST) (2017)
10. Dreibholz, T.: Big Data Applications on Multi-Clouds: An Introduction to the MELODIC Project. Keynote Talk at Hainan University, College of Information Science and Technology (CIST) (2017)
11. Dreibholz, T.: NEAT Sockets API. Internet Draft draft-dreibholz-taps-neat-socketapi-04, IETF, Individual Submission (2019)

12. Dreibholz, T., Becke, M.: The RSPLIB Project – From Research to Application. Demo Presentation at the IEEE Global Communications Conference (GLOBECOM) (2010)
13. Dreibholz, T., Mazumdar, S., Zahid, F., Taherkordi, A., Gran, E.G.: Mobile Edge as Part of the Multi-Cloud Ecosystem: A Performance Study. In: Proceedings of the 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). Pavia, Lombardia/Italy (2019)
14. Dreibholz, T., Zhou, X., Becke, M., Pulinthanath, J., Rathgeb, E.P., Du, W.: On the Security of Reliable Server Pooling Systems. International Journal on Intelligent Information and Database Systems (IJIIDS) **4**(6), 552–578 (2010)
15. Dreibholz, T., Zhou, X., Fu, F.: Multi-Path TCP in Real-World Setups – An Evaluation in the NorNet Core Testbed. In: 5th International Workshop on Protocols and Applications with Multi-Homing Support (PAMS), pp. 617–622. Gwangju/South Korea (2015)
16. Ford, A., Raiciu, C., Handley, M., Bonaventure, O.: TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, IETF (2013)
17. Fu, F., Zhou, X., Dreibholz, T., Wang, K., Zhou, F., Gan, Q.: Performance Comparison of Congestion Control Strategies for Multi-Path TCP in the NorNet Testbed. In: Proceedings of the 4th IEEE/CIC International Conference on Communications in China (ICCC), pp. 607–612. Shenzhen, Guangdong/People's Republic of China (2015)
18. Fu, F., Zhou, X., Tan, Y., Dreibholz, T., Adhari, H., Rathgeb, E.P.: Performance Analysis of MPTCP Protocol in Multiple Scenarios. Computer Engineering and Applications **52**(5), 89–93 (2016)
19. Gilligan, R., Thomson, S., Bound, J., McCann, J., Stevens, W.R.: Basic Socket Interface Extensions for IPv6. Informational RFC 3493, IETF (2003)
20. Golkar, F., Dreibholz, T., Kvalbein, A.: Measuring and Comparing Internet Path Stability in IPv4 and IPv6. In: Proceedings of the 5th IEEE International Conference on the Network of the Future (NoF), pp. 1–5. Paris/France (2014)
21. Gran, E.G., Dreibholz, T., Kvalbein, A.: NorNet Core – A Multi-Homed Research Testbed. Computer Networks, Special Issue on Future Internet Testbeds **61**, 75–87 (2014)
22. Livadariu, I.A., Ferlin, S., Özgü Alay, Dreibholz, T., Dhamdhere, A., Elmokashfi, A.M.: Leveraging the IPv4/IPv6 Identity Duality by using Multi-Path Transport. In: Proceedings of the 18th IEEE Global Internet Symposium (GI) at the 34th IEEE Conference on Computer Communications (INFOCOM), pp. 312–317. Hong Kong/People's Republic of China (2015)
23. Parmar, H., Thornburgh, M.: Adobe's Real Time Messaging Protocol. Tech. rep., Adobe (2012)
24. Raiciu, C., Paasch, C., Barré, S., Ford, A., Honda, M., Duchêne, F., Bonaventure, O., Handley, M.: How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI), pp. 1–14. San Jose, California/U.S.A. (2012)
25. Stevens, W.R., Fenner, B., Rudoff, A.M.: Unix Network Programming. Addison-Wesley Professional (2003)
26. Tadayoni, R., Henten, A.: From IPv4 to IPv6: Lost in translation? Telematics & Informatics **33**(2), 650–659 (2016)
27. Wang, K., Dreibholz, T., Zhou, X., Fu, F., Tan, Y., Cheng, X., Tan, Q.: On the Path Management of Multi-Path TCP in Internet Scenarios based on the NorNet Testbed. In: Proceedings of the IEEE International Conference on Advanced Information Networking and Applications (AINA), pp. 1–8. Taipei, Taiwan/People's Republic of China (2017)
28. Weinrank, F., Grinnemo, K.J., Bozakov, Z., Brunström, A., Dreibholz, T., Hurtig, P., Khademi, N., Tüxen, M.: A NEAT Way to Browse the Web. In: Proceedings of the ACM, IRTF and ISOC Applied Networking Research Workshop (ANRW), pp. 33–34. Praha/Czech Republic (2017)
29. Wischik, D., Raiciu, C., Greenhalgh, A., Handley, M.: Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI), pp. 99–112. Boston, Massachusetts/U.S.A. (2011)
30. Zhou, F., Dreibholz, T., Zhou, X., Fu, F., Tan, Y., Gan, Q.: The Performance Impact of Buffer Sizes for Multi-Path TCP in Internet Setups. In: Proceedings of the IEEE International Conference on Advanced Information Networking and Applications (AINA), pp. 9–16. Taipei, Taiwan/People's Republic of China (2017)