Proceedings of the 17th Linux Audio Conference (LAC-19), CCRMA, Stanford University, USA, March 23–26, 2019

TPF-TOOLS - A MULTI-INSTANCE JACKTRIP CLONE

Roman Haefeli
ICST (Institute For Computer Music And Sound Technology)
Zurich University of the Arts, Switzerland
roman.haefeli@zhdk.ch

Johannes Schütt
ICST (Institute For Computer Music And Sound Technology)
Zurich University of the Arts, Switzerland
johannes.schuett@zhdk.ch

Patrick Müller
ICST (Institute For Computer Music And Sound Technology)
Zurich University of the Arts, Switzerland
patrick.mueller@zhdk.ch

ABSTRACT

Tpf-tools are used to establish bi-directional, low-latency, multichannel audio transmission between two or more geographically distant locations. The tool set consists of a server part (the tpf-server) and a client part (the tpf-client) and is heavily inspired by the JackTrip utility. It is based on the same protocol. It facilitates the handling of many concurrent audio transmissions in setups with more than two endpoints. Also, it eliminates the requirement of one endpoint having a public IP address or port forwarding configuration.

1. INTRODUCTION

The JackTrip[1] utility has proven to be a very useful and versatile tool for our research into the so-called telematic performance format (tpf), staged (musical or other kinds) events that take place simultaneously at two or more geographically distant concert venues. For these concerts, the stage is designed to blend physically present local performers with their remote counterparts, represented by means of low-latency video (UltraGrid [1]) and audio (JackTrip) transmission.

1.1. The obstacles of current IP networks

We have successfully used the JackTrip utility in many of our telematic concerts. The utility operates in two modes: client mode and server mode. For an audio transmission to take place, one end runs it in server mode listening for an inbound connection, while the other end runs it as client, thus initiating the connection. This works well so long as the client "sees" the IP address of the server. In today's Internet, most computers touched by human beings are assigned an IP address from a local area network (LAN) which is protected by a NAT router [2] . Public IP addresses are usually only assigned to headless servers and – apparently – NAT routers, but not to devices touched by humans. This topology divides the Internet in service providers and consumers and reflects the predominant capitalist ideology of today's Internet [2, Chapter 5]. At the same time, it hinders our efforts to perform telematic concerts. Running JackTrip in server mode at a concert venue requires a computer that has either a public IP assigned, or the proper port forwarding configured on the local network router. At venues where the performers are not the owners or administrators of the local network, this often bears huge administrative overheads and dealing with IT staff who may be more concerned about security than artistic achievements.

[1] Software for low-latency video transmission http://www.ultragrid.cz/

[2] NAT (network address translation) routers separate the LAN from the Internet. This increases security, because local computers are invisible from the Internet. It is also a way to deal with IPv4 address exhaustion, because all devices of a local network share one public IP address for outbound connections.

1.2. The complexity of many nodes

Another complexity we have encountered is the planning and set up of JackTrip connections when, not two, but three or (for a test situation) four venues are participating in an event. Two endpoints require one link. Three endpoints require three links, while four endpoints require six links. The number of links grows quickly with the number of endpoints. Events with more than two nodes require meticulous and careful planning.

1.3. Our motivation

We are looking for ways to streamline our processes and improve our tools in order to be able to shift our focus away from technical to more artistic aspects. JackTrip is the tool of our choice, because it is multi-platform, open source, uses JACK [3] and thus integrates well with existing professional audio software (e.g. Ardour). However, we saw an opportunity in adding a higher layer on top of the strong basis JackTrip gives us. In our efforts, we have developed a tool set that addresses the obstacles we've been experiencing:

- None of the endpoints need a public IP address.
- The client manages the audio transmissions to many endpoints and abstracts the complexity of such setups away, while presenting a simple, yet comprehensive interface to the user.

In this paper we present our tool set consisting of the tpf-client [4] (the software that is running on each participating endpoint) and the tpf-server [5] (the software that enables communication between the clients and coordinates audio transmissions).

2. VARIOUS CONNECTION MODES

2.1. Client connects to server (standard mode)

The JackTrip utility is designed so that both ends are sending similarly formatted UDP [6] packets. In server mode, it opens a listening socket that awaits for incoming connections. As soon as a packet arrives, it starts sending packets to the sender address of the incoming packets. In client mode, it immediately starts sending packets. The transmission is established as soon as both ends are up and running. This only works when the IP address of the server is visible to the client.

[3] Jack Audio Connection Kit, a sound server daemon for connecting audio applications and sound cards. http://www.jackaudio.org/

[4] The tpf-client is available at https://gitlab.zhdk.ch/TPF/tpf-client.

[5] The tpf-server is available at https://gitlab.zhdk.ch/TPF/tpf-server.

[6] User Datagram Protocol, a connectionless protocol based on the Internet Protocol that operates on the Transport Layer (Layer 4) of the OSI model. Applications with a strong focus on low latency often use it for transport.

2.2. Two clients connect to each other

A transmission can also be established when running both endpoints in client mode so long as both clients specify both bind port and peer port. The peer port of the first client matches the bind port of the second client, and vice versa.

An example of a JackTrip setup with both instances running as client:

```
$ jacktrip -c 192.168.0.12 --bindport 2000
    --peerport 3000
```

```
$ jacktrip -c 192.168.0.11 --bindport 3000
    --peerport 2000
```

This requires both ends to have an IP address visible to the other party. If one or both endpoints are hidden by a NAT-firewall, a connection cannot be established. However, this setup shows that the JackTrip design does not mandate one party to run as server.

2.3. Connection using a UDP proxy

The fact that a transmission can happen with two endpoints both running in client mode is crucial for the next step: establishing a transmission where none of the endpoints are assigned a public IP address. Since we want both endpoints to run in client mode, we need a third party that has assigned a public IP address and thus is visible for both endpoints, even when they are behind a firewall. This third party acts as proxy for both endpoints by relaying packets from client A to client B and vice versa. This technique passes most types of firewalls easily because the client initiates the connection. It works transparently for both endpoints as they do not have to know their respective peer's IP address. They simply connect to the UDP proxy. Since the JackTrip packet format is agnostic of the underlying transport protocol, all connection specific details are part of the UDP header and the payload does not contain any reference to the client address or port number. This allows the UDP proxy to relay incoming datagrams as is, without inspecting or changing the payload.

3. SUBSCRIPTION-BASED UDP PROXY

The simplest variant of a UDP proxy knows exactly two endpoint addresses and relays packets between them. However, this design mandates that each parallel transmission uses an instance of the UDP proxy, each listening on a dedicated port. The purpose of the subscription-based UDP proxy is to allow many parallel transmissions on the same port. To know which endpoints belong to a certain transmission, the endpoints send a so called token that is unique per transmission. If two clients send the same token, a transmission between those endpoints is established. This design allows an arbitrary number of transmissions to run on the same port, and each transmission is protected from intentional or unintentional interference by the token. Because of the requirement to send a token, the subscription-based UDP proxy does not work with the traditional JackTrip, at least not out-of-the-box [7] . Also, both parties intending to participate in a transmission must first agree on a common token through a separate channel.

[7] JackTrip could be wrapped into a script that first sends the token using the same bind port before it starts JackTrip

3.1. Implementation

The tpf-server presented here uses a Python [8] script as subscription-based UDP proxy. It uses two dictionaries (dicts) that are empty at start-up: a token dict and a link dict. The token dict stores the token string and sender adress when a token message is received. The token message is a UDP packet containting a string like

```
_TOKEN XXXX
```

where `XXXX` is the token string, an arbitrary string of arbitrary length. If a token message is received, its token string is looked up in the token dict. If there is no entry found, an entry is added to the token dict with the token string as key and the sender address as value. If another token message is received carrying the same token string but from a different sender address, two entries are made to the link dict. The first entry uses the address from the token dict as key and the sender address of the last token message as value. The second entry uses the same two addresses, but key and value are interchanged. After creating the entries to the link dict, the respective entry in the token dict is deleted, so that the same token may be used later by another party.

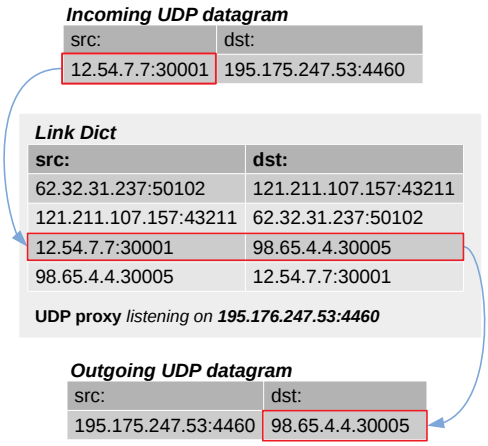

Figure 1: *Subscription-based UDP proxy.*

Since the UDP protocol does not guarantee that packets reach their destination, the client must keep sending token messages at a low rate (i.e. one message per second). When the client receives a packet for the first time, it stops sending token messages.

3.2. Considerations

Creating two entries per transmission into the link dict seems like a waste of memory, but it allows for a very quick look-up to determine the destination on an incoming packet. Keeping the latency low has the highest priority in our use case.

Although Python, as an interpreted language, is not among the fastest, it was the preferred choice for rapid prototyping and experimenting. It turned out that the UDP proxy written in Python was never the bottleneck in our performance tests and although it causes some CPU load under load, it does not seem to add a significant latency to the UDP transport. There has not yet been a pressing need to rewrite the UDP proxy in a more performant way.

[8] Python is an interpreted programming language supporting many paradigms. https://www.python.org/

4. THE TPF SERVER

The complexity of a setup increases quickly with the number of participating endpoints, as we showed before. We wanted software to manage the complex part of handling many parallel audio transmissions. The engineer should not have to deal with many terminal windows for running many JackTrip instances and know what IP address and port number each of their peer uses. Simplifying the involved processes was the main motivation for defining a protocol [3] and writing a server software implementing that protocol. It is worth noting that this part is orthogonal to the problem of audio transmission. The tpf-server is not involved in transmission of any audio data. Rather, it enables clients to know about each other and to let them initiate audio transmissions. The communication between server and clients uses TCP and runs on different ports.

4.1. Based on netpd-server and OSC

In order to reduce development efforts, the design is based on existing software – the netpd-server [9] – that was extended to implement the tpf-server presented here. The netpd-server is a relay for OSC messages and was developed for the netpd [4] project, a framework based on Pure Data (Pd) [5] that allows geographically remote clients to do electronic music together in real-time by synchronizing instrument states. The netpd framework uses OSC [6] for the communication, while OSC messages are encapsulated by SLIP [7] and transported by TCP. The OSC 1.1 specification [8] proposes SLIP to delimit OSC messages when transported by stream-oriented protocols such as TCP. While many OSC applications use UDP for transport for simplicity and speed, data integrity and correct order are crucial for the netpd framework. Also, for the tpf-server, whose purpose is to coordinate clients and allow them to share data, and which is not involved in the audio transmission directly, reliability trumps speed. TCP has a notion of connection, so for a server using TCP, there is no ambiguity in knowing when a client joins or leaves. With UDP it is much harder to clearly determine a client's state (e.g. joined or left).

4.2. netpd-server

The netpd-server defines rules about how incoming OSC messages are forwarded to the connected clients. This allows clients to send messages to specific peer clients, broadcast messages to all clients, or send messages to the server itself. The netpd-server forwards OSC messages according to the first element of the OSC path. The set of supported values for this field is listed here:

field	forwarding action
b	message is broadcast to all connected clients
s	message is intended for the server itself (not forwarded)
<*int*>	message is forwarded to the client with ID <*int*>

Table 1: List of valid receivers

4.3. The tpf-server internals

The tpf-server loads the netpd-server as an abstraction [9]. It reserves the OSC name space `/s/tpf`, which means all received messages whose OSC address starts with `/s/tpf` are handled by the tpf-server. The protocol is built on top of the protocol of the netpd-server. The exact protocol specification is part of the tpf-server package [3]. Since the protocol is based on OSC, it is agnostic of any software framework or programming language. It could be implemented in any language where libraries exist to deal with network sockets and the OSC protocol. It was implemented in Pure Data, because it uses parts already written in Pure Data. The tpf-server keeps track of the connected clients and coordinates a few common parameters that the endpoints must agree on before they are able to establish an audio transmission. It manages a few data containers and notifies clients about updates when data is changed. The tpf-server sends current data to the clients upon their request, while it is the duty of the clients to request data if they receive an update notification from the server. The data containers include:

4.3.1. Client ID And Name

When a client connects, the tpf-server assigns it a unique client ID (unique in the scope of the session). This ID, usually a small integer number, is used to identify each client. The same ID is also used to send an OSC message to a specific client by putting it into the first field of the OSC path. After establishing the connection to the server, the client registers a name (e.g. given name or location). It allows clients to display the list of connected peers in a more human-friendly way (see Client List).

4.3.2. Parameter List

The client with the smallest ID, usually the one that connects first to the server, is given a special role: it has the authority to set or change a set of parameters that all clients are mandated to use for the current session – samplerate, blocksize, bit resolution. Those parameters are distributed to all clients and the clients either adjust their settings or report an error when a mismatch occurs. The parameter list is not a hard-coded set. Instead, it is fully defined by the clients.

4.3.3. Client List

The tpf-server keeps a list of all connected clients with their ID, name, IP address and role. Whenever a client connects or disconnects, the tpf-server broadcasts an update of this list to all clients. It is therefore crucial that clients terminate their connection properly, otherwise they keep appearing in the client list until the connection is considered terminated. This period depends on the operating system.

4.3.4. Link List

In a full mesh network, each node is linked to every other node. If n is the number of nodes, the number of links (l) is:

$$l = \frac{n(n-1)}{2} \tag{1}$$

The tpf-server assigns each pair of clients a link ID, so each link ID associates two clients. The tpf-server sends each client its own list of their peer's client IDs along with the corresponding link ID. Clients use the link ID to establish the audio transmission to a specific peer. Early versions used one server port per transmission and tpf-client used the link ID as the port offset parameter for running JackTrip. In the current version, the link ID is used to generate a token string. Two clients using the same ID and thus the same token string are linked by the subscription-based UDP proxy.

[9] The netpd-server is part of the netpd framework developed by Roman Haefeli. The code is hosted at https://github.com/reduzent/netpd-server

When every transmission was using its dedicated UDP port, it seemed appropriate to let the server, as a central authority, assign link IDs to avoid collisions, but also to ensure that only IDs corresponding to an active UDP proxy would be assigned. With the subscription-based UDP proxy, this coordination task became moot, as clients could also negotiate a token by peer-to-peer communication without involving the server. Future versions of the tpf-server might remove the link list.

5. THE TPF-CLIENT

5.1. Written in Pure Data

The tpf-client is implemented in Pure Data, so it can be built on top of an already existing framework. For communication to the server, parts from the netpd client have been reused. Designed as a real-time audio programming language, Pure Data has already covered many aspects of dealing with low-latency audio. Furthermore, part of the Pd "eco system" is a vivid community that has been contributing many libraries extending the functionality of the software. Namely, there are so-called externals for parsing and formatting OSC messages (*osc*) and for accessing network sockets (*iemnet*). Pure Data has native JACK support built-in and runs on a variety of platforms.

5.2. Implementation

The purpose of the tpf-client is to manage audio transmissions to one or many peers joining the same session. It is the implementation of the client side of the tpf protocol. First drafts only implemented the management aspects in order to get the necessary information for starting the original JackTrip utility with the appropriate command-line arguments, so the audio transmission part was left completely to JackTrip. It was later decided to also re-implement the JackTrip utility as an abstraction.

5.2.1. Rewrite of JackTrip as Pd abstraction

Implementing the audio transmission part in Pure Data has some advantages:

- The lack of a stable and feature-complete Pd external for running system commands makes it hard to consistently control many JackTrip instances from Pd. JackTrip reimplemented as a Pd abstraction is easier to control and interface with.
- An implementation of the JackTrip protocol in Pd allows to extend it, if necessary. A small addition – the subscription by sending a token message – to the JackTrip functionality was necessary to support the subscription-based UDP proxy.
- Although able to create many JackTrip connections, the tpf-client appears as one JACK client, which somewhat simplifies the process of drawing connections in the connections dialog of QjackCtl.
- Since the audio signals travel through Pd, some signal processing could be applied. The current implementation doesn't apply any processing, though.
- Since the audio signals travel through Pd, signal level monitoring can be used and graphically represented in the client user interface.
- Signal path can be used to measure round-trip time of the audio signal with built-in latency meter.

5.3. User interface

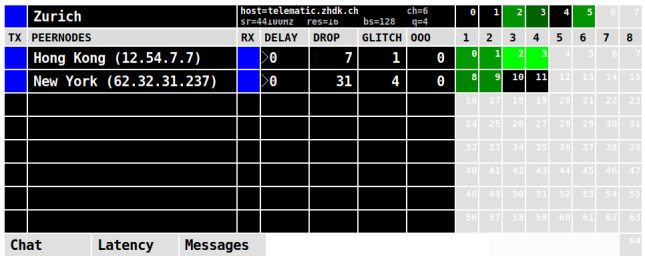

Figure 2: *The tpf-client user interface.*

The user interface displays a few configuration parameters that are settable before the connection to the server is initiated:

- name
- hostname (or IP address) of the tpf-server
- blocksize (of the JackTrip packets)
- number of channels (outgoing)
- queue buffer size

The *samplerate* and *bit resolution* cannot be changed in the client. The bit resolution is hard-coded to 16 bit. The samplerate is mandated by the JACK server and is inherited by Pd. After the connection is established, those configuration parameters become locked and cannot be changed until the session ends.

The client registers its name and either uploads the audio parameters such as *samplerate*, *blocksize*, *bit resolution* to the server or matches them against the mandated parameters, if another client already has configured those parameters. If there is mismatch between configured and mandated parameters, the client either reports an error (mismatch with *samplerate*, *bit resolution*) or silently adjusts the parameter (mismatch with *blocksize*). It is worth noting that blocksize configured in the tpf-client is decoupled from the blocksize used by the JACK server. This allows clients to run JACK with deviant blocksizes. After successfully having registered the name and matched audio parameters, the connection button (top left) turns blue to indicate that the client is ready for audio transmissions.

5.4. Managing transmissions

Peer clients are each listed in a separate row in the client interface. Audio transmissions are not started automatically, but are initiated by a user on either side by clicking the left-most button in the row. The button on the respective row on the peer's client starts flashing. Only when confirmed by the other end by clicking on the flashing button is the audio transmission started. The number of received channels is represented by the number of squares turning from grey to black in the respective row. Depending on the signal level of each channel, the square changes color from black (silence) to bright green (full amplitude). The number in each square corresponds with port number of the tpf-client in the QJackCtl connection dialog.

5.5. Transmission monitoring

During an audio transmission, three types of glitches are counted and displayed in the respective row:

DROP number of dropped packets. Late packets that miss their time frame to be played back are also considered dropped.

GLITCH number of audible audio glitches. Often, many packets are dropped in a row, resulting in one audible glitch. Thus, the number of audible artefacts is always smaller than the number of dropped packets.

OOO number of packets received out of order. If an out-of-order packet misses its time frame, it is dropped. Otherwise it is played back in correct order.

All counters are reset to zero when the audio transmission is restarted. Although those counters are not of much use during a real concert (not much can be done about bad statistics), they might help compare the quality of different network links, when testing setups or internet providers.

5.6. Message and chat window

Beside the main window, tpf-client's interface has a message window, where info, warning and error messages are displayed. There is also a built-in chat in the chat window. A channel of communication not involving audio is often desired.

5.7. Built-in latency measurement tool

To measure the overall round-trip time of the audio signal, both endpoints need to configure the audio path accordingly. The method is robust enough to allow the signal to be played back by a speaker and recorded with a microphone, even in a mildly noisy environment. The signal path of a full round-trip measurement is shown in Figure 3 .

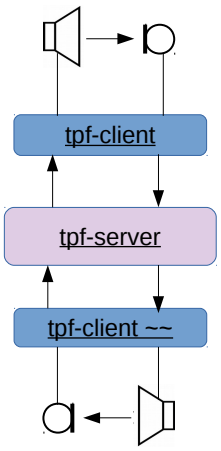

Figure 3: *Signal path of latency measurement.*

5.8. Adding artificial latency

The tpf-client allows each audio transmission to add an artificial audio delay. By adjusting the delay, it is possible to target a specific total round-trip time. Reasons for latency adjustment include:

- The performance of a certain musical piece requires the perceived latency to be aligned to the given tempo of the work.
- In a three-node setup, where one peer location is far more remote than the other, the un-adjusted latencies differ significantly, so it might be desired to "harmonize" the perceived latencies by artificially increasing the "distance" of the closer peer.

5.9. Considerations

Certain aspects of writing software in Pd are difficult. Designing a graphical user interface is relatively hard and the graphical representation is bound to pixel sizes and cannot be scaled dynamically (i.e. by resizing the window). Also, it is not possible to create dynamic interfaces that display different content depending on context. Due to those limitations, it was decided to restrict some capabilities of the client in order to provide a simple and consistent interface. The number of channels per audio transmission is limited to 8. Also, the maximum number of displayed peers and thus the number of concurrent audio transmissions is limited to 8. This limits the overall number of connected client being able to interact with each other to 9. Those limitations are not imposed by the tpf-server or the protocol, and the client could be adapted if need be. They are abitrary choices and during the past year of using the tpf-tools, those limits never have been reached in real life.

Unlike the original JackTrip implementation, each party in a setup using the tpf-tools can choose the number of channels to be sent individually. This saves bandwidth and might improve quality. Also, the configured blocksize is not dependent on the blocksize mandated by the JACK server. This can be an advantage, since the value for the most optimal JACK configuration might differ between clients.

6. EXPERIENCES AND DISCUSSIONS

We were interested to know how the usage of the tpf tools impacts audio quality and overall latency. We performed tests to compare the usage of the UDP proxy with a traditional JackTrip client-server connection. We wanted to know whether the usage of the UDP proxy has an influence on the number of dropped packets. In another test, we examined the latency differences between using a UDP proxy and a direct JackTrip connection. We also examined, whether the tpf-client imposes a penalty to the quality of the audio transmission compared to the original JackTrip.

6.1. Dropped packets imposed by UDP proxy

For counting glitches (which are a result of dropped packets), we sent a 1kHz-sine-tone through JackTrip to a remote JackTrip instance, that looped back the signal, and recorded the result for a predeterimined period of time. We used the `-z` commandline option of JackTrip, so that glitches were visually more easy to spot in the waveform. Then we counted the glitches by loading the recorded sound file into a sound editor and examining the discontinuities in the waveform. We were not able to determine a significant difference between a direct link and a link using the UDP proxy. At another instance, that was totally unrelated to the test series, we experienced many dropped packets. We later found out that the reason was a bug in the driver of the virtual network interface of the virtual machine the UDP proxy is running on. While the UDP proxy usually does not impact the number of dropped packets negatively, there is a plethora of possibilities as to why the UDP proxy might behave badly, because it

depends on hardware, on the operating system and of the software itself. These sources of error do not apply to a direct JackTrip link.

6.2. Latency imposed by UDP proxy

At the time of comparing the latency of a direct link to the UDP proxy, the tpf-client had not been written. So a simple tool in Pd was built to send a single UDP packet to a remote location, that sends back the packet immediately. The tool measures the delay betwen sending and receiving the packet. The average travel times turned out to be identical for both, a direct link a link using the UDP proxy. This behavior was consistent with different remote locations. This can be eppxlained by the fact that both, the computer taking the samples and the server running the UDP proxy, were located at the same campus. By using tools like mtr or traceroute, we found out that the number of hops between the computer taking samples and the remote computer was the same for both link types. In a scenario where both endpoints are located outside the campus hosting the UDP proxy, using the UDP proxy adds additional latency. The amount depends on how far the UDP proxy is away from the direct network path between both endpoints.

6.3. Performance of the tpf-client

We also tried to examine the impact of using tpf-client compared to the original JackTrip. It turns out that Pure Data adds one block of additional latency, because the way it communicates with JACK decouples its audio processing from the strict graph of the JACK server. Many other JACK clients like JackTrip are tightly coupled and do not add additional latency. When using a blocksize of 128 at a samplerate of 48kHz, the penalty of using tpf-client is 2.6666 ms. It increases with larger blocksizes or lower samplerates.

Because Pd interfaces the JACK server differently, it is possible that Pure Data's audio processing experiences audio drop-outs while the JACK server does not. This means that the tpf-client introduces a new source of possible buffer underruns. From our experience, this theoretical penalty has not become manifest in more glitches when using tpf-client, at least not when running tpf-client on a macOS system. On Linux, Pure Data was found, in some situations, to be the source of glitches when not running with realtime privileges. It was usually simple to remedy the situation.

6.4. Shortcomings of the JackTrip protocol

While measuring the number of glitches with different combinations of blocksize and number of channels, we found there was a sudden increase in glitch rate when the number of channels exceeded a certain value. When running two parallel transmissions with each only carrying half the channels, we experienced a low rate of glitches. By running other tests with the tool iperf, which allowed us to set the rate and size of UDP packets, we found that link capacity was only one limiting factor. Not less important was the so-called Path MTU [10] . UDP packets larger than the Path MTU are fragmented during transport. The loss of a single fragment results in the loss of the whole UDP packet. The likeliness of a UDP packet being dropped increases with the amount of fragmentation it experiences. For best performance, the UDP packet size should not exceed the Path MTU.

[10]Maximum Transmission Unit, is the maximum packet size that is a transmitted in a single network layer transaction, while Path MTU refers to the maximum packet size that is transmitted through all intermediate hops without fragmentation.

By running tests with iperf, we were not able to saturate a network link with a UDP stream, when choosing a relatively large packet size (e.g. 16000 bytes). By selecting a smaller packet size (e.g. 1400 bytes), we were able to achieve a data transfer rate close to the theorethical maximum while still keeping the number of dropped packets low. This finding shows that the JackTrip protocol is not suitable for all kinds of payloads, since the UDP packet size depends on bit resolution, number of channels and blocksize:

$$packetsize = H_{\text{UDP}} + H_{\text{jacktrip}} + N_{\text{channel}} \times \frac{b_{\text{res}}}{8} \times B_{\text{buffer}} \quad (2)$$

where H_{UDP} = Header size of UDP datagram,
H_{jacktrip} = Header size of JackTrip frame,
N_{channel} = Number of channels,
b_{res} = bit resolution,
B_{buffer} = buffer size

Larger numbers of channels or blocksize result in UDP packet sizes bigger than the optimal size. With a typical Path MTU of 1500, and a given blocksize of 128, the largerst number of channels still fitting into the Path MTU is 5 (1296 bytes). A single audio transmission with a high number of channels could be split into two or more parallel transmissions with a lower number of channels in order to reduce the resulting packet size. However, synchronization between the transmissions is not guarantueed and therefore this is not a suitable solution. The ability to detect the Path MTU and to optimize UDP packet size by splitting a transmission into many, while keeping synchronicity, are features that still need to be researched.

6.5. UDP hole punching

While there is none or only a negligible penalty for using the UDP proxy when it is located close to one participating party, it might add significantly to unacceptable latency, when the participating parties are all located geographically distant from it. In terms of network latency, using a direct link is sometimes as good, and in many cases clearly superior to using a proxy. A technique called UDP hole punching allows us to establish a direct UDP conncetion between two end-points, both acting as client, that is able to traverse many types of NAT-firewalls. NAT-firewalls usually let an incoming UDP packet pass, when its receiver address (IP and port) matches the sender address of a previously outgoing UDP packet. That is because UDP is a stateless protocol and has no notion about connection. That is how NAT-firewalls discern outbound connections (that are usually allowed) from inbound connections (that are usually blocked). Before establishing the connection, both endpoints contact a central server to learn about their peer's public IP address and port number. Then they start sending packets to the address they learned. Because this happens on both sides, the firewall on either side "thinks" the connection was initiated from a local client and it will pass incoming packets. The technique is already used in webRTC and IP telephony applications. The tpf-client supports UDP hole punching as an experimental feature. By double-clicking (instead of single-clicking) the left button in the peer row an audio transmission using a direct link is requested. There are still many scenarios where establishing a such link fails. Supporting more cases and making UDP hole punching a viable option is certainly a field worthy of further exploration.

7. ACKNOWLEDGEMENTS

Our research was funded by the *Swiss National Science Foundation* [11] and took place at the *Institute For Computer Music And Sound Technology* [12] (ICST), *Zurich University of the Arts*, Switzerland. Over the course of two years, we have been working within a team that explored many aspects of telematic performances, including scenography, audio engineering, audio and video streaming, network technology and considerations in the field of media theory. We are grateful for the collaboration with those very interesting people and feel there is still a lot waiting to be explored and further researched. We have been working with Matthias Ziegler, flutist and head of the research group, Benjamin Burger and Joel De Giovanni, video artists and scenographers, Bojan Milosevic, composer and researcher, Gina Keller and Ernesto Coba, audio engineers. We also thank all collaborating parties spread around the world for having showed the willingness to organize and perform telematic concerts with us and to use and test our tools. We appreciate being a part of this vivid community.

8. REFERENCES

[1] Juan-Pablo Caceres and Chris Chafe, "JackTrip: Under the Hood of an Engine for Network Audio," *Journal of New Music Research*, 2010.

[2] Robert W. McChesney, *Digital Disconnect: How Capitalism is turning the Internet against Democracy*, The New Press, New York, 2013.

[3] "tpf: Protocol Specification," `https://gitlab.zhdk.ch/TPF/tpf-server/blob/master/protocol_specification.txt`.

[4] Roman Haefeli, "netpd - a Collaborative Realtime Networked Music Making Environment written in Pure Data," in *Linux Audio Conference*, 2013.

[5] Miller Puckette, "Pure Data," `http://puredata.info`, 1996, Software.

[6] "Open Sound Control," `http://opensoundcontrol.org/`, Protocol.

[7] J. Romkey, "A Nonstandard For Transmission Of IP Datagrams Over Serial Lines: SLIP," Tech. Rep. RFC 1055, IETF, Network Working Group, 1988.

[8] Adrian Freed and Andy Schmeder, "Features and Future of Open Sound Control version 1.1 for NIME," in *NIME*, 2009.

[9] Miller Puckette, "Pd Documentation," `https://puredata.info/docs/manuals/pd/x2.htm#s7.1`, 2.7.1. abstractions.

[11] SNF: http://www.snf.ch/

[12] ICST: https://www.zhdk.ch/en/research/icst