netpd - a Collaborative Realtime Networked Music Making Environment written in Pure Data

Roman HAEFELI
Media Artist
Zürich,
Switzerland
roman.haefeli@gmail.com
http://www.netpd.org

Abstract

This paper presents *netpd*, a framework intended for making music collaboratively and in real-time written in Pure Data (Pd)[1]. Users join by connecting to a central server in order to have a session together (not much unlike a jam session in Jazz music) and load self-written or pre-existing instruments (Pd patches) to play with. The framework maintains state synchronicity between clients at any given time by exchanging control messages over the server. The protocol in use is fully based on OSC[2].

netpd does not address the transmission of audio data, thus it is primarily used for synthesized / generated sound, but might be useful in other areas where state synchronicity is a goal (networked games, graphics, etc.).

Keywords

Pure Data, Network, Music, Real-time, OSC

1 Introduction

Early experiments with transmitting control messages over a network for making music started in 2004 when an early version of *netpd* was developped. While first drafts of instruments were an interleaved blend of DSP parts, message control and state synchronization parts, it became clear soon that a design which clearly separates those parts would allow the creation of a framework that gives the designer of an instrument a high degree of freedom while keeping the complexity of state synchronization under hood. Crucial to the *netpd* experience are two distinct layers:

1. The *netpd* core framework (in this paper simply called framework), which consists of a server, a client application, and a set of abstractions[1] (*netpd-abstractions*) which are used to create *netpd-instruments*.
2. The *netpd-instruments*: Pd-patches created by *netpd* users which are loaded in the client application and played during a session.

This paper primarly addresses the design of the framework, which aims to provide the tools to enable skilled[2] users to design their own instruments to be used and shared with *netpd*. It is important to understand that the presented framework has no notion of music and is only the basis for user-designed instruments and that those instruments make up the *netpd* experience. It becomes apparent that collaboration happens on the level of playing together, but also on the level of designing and sharing patches.

2 Basic design

Users load netpd's client application chat.pd in Pd (or Pd-extended, for that matter) which establishes a connection to a central server. The server acts as a message relay between clients: it forwards incoming messages from clients to any or all other clients.

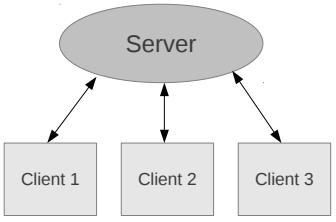

Figure 1: Client-server model

A session starts as soon as a user loads an instrument - which itself is a Pd patch with certain *netpd* specific properties - into the client. The clients keep the list of of loaded instruments synchronized among each other at any time. As necessary, clients even transfer the instruments (the .pd-files) to their peers in order

[1] abstractions are modules written in Pure Data that can be instantiated in a patch.

[2] knowing how to create Pd patches is sufficient to be able to create *netpd-instruments*

to ensure synchronicity . Any user may load more instruments into their client and these appear immediately (or after the time of transfer) in all clients.

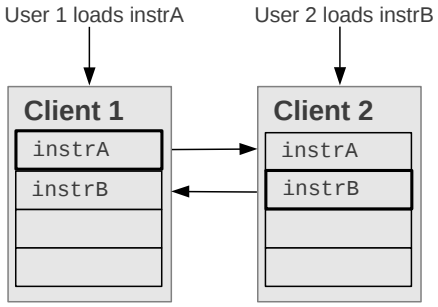

Figure 2: Instrument synchronization

Users play the instruments by manipulating their GUIs. Also the instruments keep their state synchronized among clients. Any change is immediately reflected on all clients. All users can play on all loaded instruments. Also, every user immediately experiences the manipulations of its peers. Although the generated sound is rendered on every client separately, the result is the same everywhere.

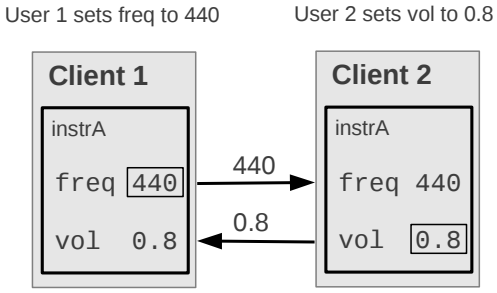

Figure 3: State synchronization

3 The framework

Let me divide the requirements of above scenario into three main tasks which will be covered separately in this document.

Obviously, the clients need a way to communicate with each other. A message protocol is defined, which the whole communication between clients (and between client and server) is based on.

Another task of the framework is to share instruments between clients and to make sure that at any given time the set of loaded instrument is synchronized among connected clients. In that respect, *netpd* acts as a peer-to-peer file sharing tool for Pd patches.

State synchronization among instances of *netpd-instruments* is a further goal of the framework. Similar to the sharing of instruments described above, state synchronicity must be maintained at any given time. Unfortunately, state synchronization does not automatically work for arbitrary patches. The use of *netpd-abstractions* facilitates the creation of state-synchronized instruments.

3.1 The message protocol

It was decided to make the communication of the framework fully based on the OSC protocol, mainly because of its flexibility and its wide acceptance in music and related fields. OSC is agnostic of the underlying transport layer. *netpd*'s requirements for reliability left TCP as the preferred transport protocol. *netpd* adheres to the version 1.1 of the OSC specification[3] which specifies SLIP[4] as a framing mechanism for stream-oriented protocols (such as TCP). Figure 4 shows how the protocols are stacked.

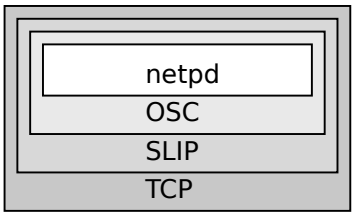

Figure 4: protocol stack

3.1.1 Receiver ID

The sole purpose of the server is to relay messages between clients. Clients may send messages either to all or to specific clients. This is achieved by defining the first field of the OSC address as the receiver ID. Table 1 shows the complete list of valid receiver IDs:

ID	Receiver
/b	broadcast (all clients)
/s	server (not forwarded)
/l	local (not sent to server)
/[ID]	client with given ID

Table 1: List of valid receivers

The /l address is used in a similar way to *localhost* in networking: /l-messages are looped back by the client. All other messages are sent to the server. The server checks the first field of

an incoming messages and forwards it accordingly. It disregards any message with an invalid receiver ID.

/b-messages are forwarded to all connected clients, even to the one the message originated from.

/s-messages are read by the server and allow the client to request certain data such as its client ID.

Any message whose first field is an integer number between 0 and 999'999 is forwarded to client with the given ID if it exists. Otherwise the message is disregarded. The server does not strip the receiver ID field when forwarding a message, instead it replaces it by the ID of the sender. This allows the receiving client to know the origin of the message.

For instance, the server receives the following message from the client 3:

```
/6/megasynth/voice1/freq 8000
```

The server replaces the receiver ID by the sender ID and forwards the message to the client 6:

```
/3/megasynth/voice1/freq 8000
```

3.1.2 Server methods

Only the first field of the incoming message is relevant to the server, except if the first field is /s. The second field of a /s-message is the namespace for optional server modules. At the time of writing none exist, so only /server is currently used. The third field specifies the type of data the client requests. Only a minimal set of client requests is supported:

/s/server	Data
/socket	socket number (client ID)
/ip	IP address of the client
/num_of_clients	number of clients
/protocol_version	protocol version

Table 2: List of supported server methods

The server responds to such messages only to the requesting client. Client requests may consist only of the OSC address, whereas the server appends the requested data to the message. A typical client request and the according server response looks like:

```
/s/server/protocol_version
/s/server/protocol_version 2 0
```

3.1.3 Interoperability

Since OSC is a standardized protocol, other applications (Pd based or not) may be used to participate in a *netpd* session as long as the messsages adhere to above definitions. Similarly, it is thinkable to intercept *netpd* traffic in order to control hardware, for instance. Altough the protocol as well as the server were designed with the presented framework in mind, those might be used solely for the purpose of sending messages between an arbitrary number of nodes in applications totally different from *netpd*. Modularity was one of the key aspects when rewriting the framwork, which should make it easy to take out and use only those parts of interest.

NOTE: so called OSC-Bundles are not supported by *netpd* and yet it is not clear if and how support for them can be added.

3.2 The client

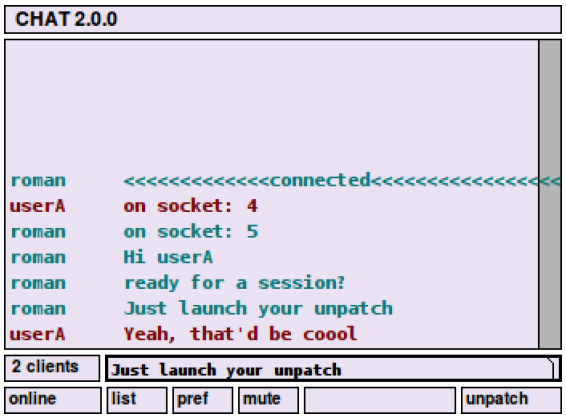

Figure 5: chat user interface

A user participates by opening the *netpd*-client's main patch called chat.pd in Pure Data. This patch immediately establishes a connection to the server and presents a user interface for chatting with other users. Being able to communicate (with words) is crucial for establishing a session. Before sending anything else though, chat checks the protocol version of the server and requests its own client ID. It also shows who is participating in the current session.

At this point the user does not yet take part in the ongoing session, but they might do so by launching unpatch (by clicking the unpatch button in chat).

Unpatch is the management interface for loading and closing *netpd-instruments*. As soon as it is launched, it automatically synchronizes with its peers and loads all *netpd-instruments*

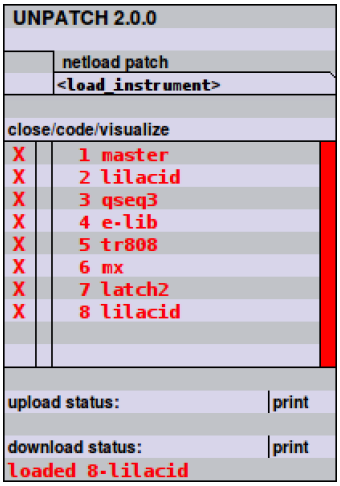

Figure 6: unpatch user interface

that are part of the running session. *netpd-instruments* that are not yet present locally are requested and transferred from the peers before loading. Once all instruments are loaded and synced, the user is able to control all instruments used in the session.

3.2.1 netpd meta tags

How does unpatch know which patches (instruments) need to be transferred? What happens when two users have different versions of the same instrument? What if a patch has dependencies, because it uses abstractions? *netpd* uses meta tags to define some properties of an instrument. A valid meta tag section in an instrument is mandatory, otherwise unpatch refuses to load the instrument. If present, those tags are read and parsed before the instrument is actually loaded. The *netpd* meta tags are organized hierarchically in subpatches and message boxes. The subpatches act as namespaces, whereas message boxes contain properties and optionally one or more values for those properties. A subpatch [pd abslist] that contains a messagebox [synthvoice(is equivalent to a messagebox containing [abslist synthvoice(. In terms of implementation, subpatches may use any depth of nesting. However, *netpd* uses one at most.

`pd NETPD 2 0` declares the section of the *netpd* meta tags. This subpatch may be placed anywhere in the instrument patch, however it is advised to put it into the main (topmost) canvas for readability. 'NETPD' is a reserved name and must not be used for any other subpatches in an instrument. The '2 0' part is optional and specifies the version of the meta tag definition. unpatch assumes the most current version if not specified.

`version 0 3 1(` is mandatory and specifies the version of an instrument. The version must consist of three integer numbers. *netpd* does not define the way they are used. When comparing two differing versions, it is only relevant for unpatch which is higher, whereas the first number is the most significant.

`pd abslist` is optional and defines the dependencies. It contains references to abstractions used by the instrument.

`synthvoice(` is the name of the abstraction and refers to a file netpd/abs/synthvoice.pd.

`singleton(` is optional and defines a singleton instrument. Such an instrument may be only loaded once. Loading further instances of such an instrument is denied by unpatch.

Both instruments (patches) and their depencencies (abstractions) must contain a meta tags section. The meta tag 'singleton' only applies to instruments, since abstractions are never loaded directly by unpatch (they are instantiated within instruments). Dependencies are resolved recursively which allows to group several abstraction into a meta abstraction that holds nothing more but a meta tag section that references many abstractions (like meta packages in Debian).

Before unpatch loads an instrument, all its dependencies and child dependencies must be resolved. Instruments can only depend on abstractions, but not on other instruments, since instruments are only loaded by user interaction and never automatically. Because instruments and their abstractions are treated in distinct ways, they are saved in separate directories: netpd/patches for instruments and netpd/abs for abstractions. Only instruments that reside in netpd/patches can be loaded with unpatch.

3.2.2 Instrument synchronization

When a client loads an instrument, it notifies all other clients about the name of the instrument, its version, its dependencies and their version. Its peers check the list and issue a request to the initiating client for any item they do not have at all or whose version number is smaller.

If they find their local version of an item to be higher, they notify the initiating client about it and the user who loaded the instrument will be presented an according message ("found version 1.3.5 of abstraction 'synthvoice' on client 7"). Such a version mismatch is not resolved automatically in order to protect the user from loading a version different from the one they had in mind. The user may still decide to resolve the situation by reloading unpatch. In this case his client requests the intrument and its dependencies from a peer and the more up-to-date remote version will overwrite the local one.

When a client joins an already running session, it tries to find a peer in a 'synced' state. If there is any, it requests the list of currently loaded instruments and dependencies (with corresponding versions) from it. As soon as all instruments are loaded successfully, it marks itself as 'synced'. From this moment it will also advertise itself as 'synced' to new clients and answer instrument list requests. In case a client does not get a response to the initial request - because it is the first in the session - it sets itself to 'synced' after a timeout.

3.3 State synchronization

netpd thinks of instruments as containers of a variety of different data sets and data types that define the state of the instrument and may be modified at any time. Those data sets may be a number (changed by slider movements, for instance), a table of numbers, a list of strings, a string, a multi-dimensional number array, etc.

A user plays an instrument by modifying those data sets which in turn control the parameters of the instrument. *netpd* provides a handful of abstractions (a.k.a *netpd-abstractions*) that each cover a specific data set. Such an abstraction automatically keeps the content of a data set of a certain instrument synchronized among all clients, no matter which user is manipulating it. Depending on the network latency and the amount of data the synchronization might happen in near real-time.

All *netpd-abstractions* provide an input for manipulating data and an output for passing those modifications to the instrument. In the simplest case the modification and the data is equivalent. For instance, a user interaction changes a number that is sent to the abstraction, the abstraction stores the number and broadcasts it to all clients and finally outputs it to the instrument. Another example of a data set is a table of numbers that may be used for a table-lookup oscillator. A modification of such table can be the change of a value at a certain position, but also a change of a whole section of the table. Changing the size of the table may represent an other valid form of manipulation. The *netpd-abstraction* responsible for synchronizing tables broadcasts those modifications to all clients. When received, the modifications are applied to the table and output to the instrument. In the case of table-lookup oscillator the instrument may not need the output as it is reading the table constantly. In other cases it might be crucial to know what exactly has been changed.

If all variable parameters of an instrument are synchronized with above techniques, the generated sound (or whatever the instrument outputs) is identical for every client.

3.3.1 Namespaces

In order to allow many instances of an instrument simultaneously, each instrument is assigned a unique ID (an integer number) at loading time. Unpatch does not load instruments as stand-alone patches, but instantiates them as abstractions and gives the ID as argument. This allows an instrument to operate in its own namespace when exchanging messages with other clients. Those namespaces are used in OSC message by putting the instrument ID into the second field of the OSC address and the instrument name into the third field. Technically the instrument name is not necessary, but it makes OSC messages more human-readable. *netpd-abstractions* used within instruments operate in a child namespace of the instrument namespace.

```
  receiver ID
  | instrument ID
  | |     instrument name
  | |     |    netpd abstraction name
  | |     |        |
 /b/7/megasynth/lookup 12 0.7 0.8
```

Figure 7: OSC namespaces in *netpd*

Namespaces with more depth might be used if appropriate. A typical use case is to group many *netpd-abstractions* into another abstraction. This way a *netpd*-ified (state-synchronized) module is created that can be instantiated many times in an instrument, with different arguments for different namespaces.

3.3.2 *netpd-abstractions*

As explained in the previous section, several kinds of *netpd-abstractions* manage different kind of data sets. In order to ensure state synchronicity of instruments among clients - even if new clients join a session - each instrument must contain a special *netpd-abstraction* [netpd_head] that manages state initialization and state transfers between clients. It is kind of the master of all other *netpd-abstractions* in the instrument. Those do not send their data to the network directly, but to [netpd_head] which prepends the appropriate namespace of the instrument before forwarding it to the network. [netpd_head] also may request all *netpd-abstractions* to dump their current state when necessary.

At instrument loading time it initializes the instrument by requesting a dump which it forwards to /l (the local client [itself]). It does so in order to transfer the internal default values of all *netpd-abstractions* to the instrument. Then it tries to find a remote peer in 'synced' state. If such a peer is around, it prompts it to send back the current state of the instrument. This sets all *netpd-abstractions* of the local instrument to the current state. The same mechanism as the instrument list synchronization of unpatch is used here.

To sum it up, an instrument needs one [netpd_head] and any number of other *netpd-abstractions*:

`netpd_head $1 megasynth` is responsible for the state management of the instrument. The variable $1 is replaced by the instrument ID given by unpatch. The second argument represent the instrument name.

`netpd_f $1 volume 0.7` synchronizes a single number. Additionally, it reads the value from a GUI object (slider, number box, radio button, etc.) whose receive and send names are set to '$1-volume' and automatically updates the GUI object on state changes. The third argument '0.7' is optional and sets the init value.

`netpd_t $1 waveform 256` synchronizes a table named '$1-waveform'. The third argument sets the table size. Although it could have been designed to hold the table internally, an external table allows the use of a graphical array in the GUI of the instrument.

`netpd_r $1 something` is a simple receiver for content in the given namespace. This comes in handy when a certain data set is needed in different locations of the instrument.

`netpd_s $1 something` is the counterpart of [netpd_r]: It sends any kind of data under given namespace. It is only used in special cases, as it doesn't have any state and thus can't be used for state synchronization.

`netpd_a $1 anything` a is container for an anything message (a message with an arbitrary number of elements).

Currently there aren't more *netpd-abstractions*, though one could easily think of more data types to be synchronized. Pure Data provides only very few data types natively, so covering those in *netpd* would require additional external libraries. More *netpd-abstractions* might be added in future versions of *netpd* (for instance, for the 'matrix' type as defined by the 'iemmatrix' library).

4 Conclusions

Although all aspects *netpd* addresses seem to work flawlessly, the overall experience is not free of issues. A major culprit are audio drop outs caused by certain tasks like loading new instruments. Some causes for audio drop outs cannot be addressed by *netpd* as they are intrinsic to Pd's design. Others have been addressed by employing threaded externals. Also there are ways to work around certain causes, for instance by loading all instruments beforehand.

In past years, *netpd* enabled the creation of a community of ever varying members from around the globe. Some sessions were spanning three continents. Quite a few users wrote instruments and some more used to play with it. A huge pile of music[3] from recorded sessions grew over the years. While technically not matured, there used to be a lot of activity. After it got more quiet around *netpd*, I decided to rewrite the framework from scratch, since some design flaws became more apparent. In the meantime, Pure Data and many libraries evolved to a higer degree of maturity, which made the rewrite presented in this paper possible at all. Many instruments have been ported from the old framework and some new ones have been written. *netpd* has been "tinkered in the

[3] http://www.netpd.org/Listen

quiet" since and no community has been grown again. A few sessions with media art students revealed that there aren't any show stoppers with the framework and some people may be intrigued by playing with it. It's now time to spread the word again, which is one reason for the desire to present it at LAC 2013.

5 Acknowledgements

The companions from the early days of *netpd* - Enrique Erne, Moritz Wettstein, Syntax the Nerd - deserve credit for contributing their thoughts about design in uncountable discussions and for writing many instruments. Also, I would like to thank the authors of the external libraries that are the most crucial for *netpd* and without them the realization wouldn't have been possible: the 'osc' and 'slip' libraries written by Martin Peach and the 'iemnet' networking library written by Martin Peach and IOhannes Zmölnig. Both authors showed a great willingness to add features useful for *netpd* and to fix issues in their libraries. Collectively, I thank all people who helped realizing sessions on radio broadcasts, at concerts or other special occasions.

References

[1] Miller Puckette. Pure Data. `http://puredata.info`.

[2] Open Sound Control. `http://opensoundcontrol.org/`.

[3] Adrian Freed and Andy Schmeder. Features and future of open sound control version 1.1 for nime. In *NIME*, 2009.

[4] J. Romkey. A Nonstandard For Transmission Of IP Datagrams Over Serial Lines: SLIP. Technical Report RFC 1055, IETF, Network Working Group, 1988.