**Multimedia Tools and Applications manuscript No.**
(will be inserted by the editor)

# Distributed Tools for Interactive Design of Heterogeneous Signal Networks

**Joseph Malloch · Stephen Sinclair · Marcelo M. Wanderley**

Received: date / Accepted: date

**Abstract** We introduce libmapper, an open source, cross-platform software library for flexibly connecting disparate interactive media control systems at run-time. This library implements a minimal, openly-documented protocol meant to replace and improve on existing schemes for connecting digital musical instruments and other interactive systems, bringing clarified, strong semantics to system messaging and description. We use automated discovery and message translation instead of imposed system-representation standards to approach "plug-and-play" usability without sacrificing design flexibility. System modularity is encouraged, and data are transported between peers without centralized servers.

**Keywords** Media Mapping · Networking

## 1 Introduction

One focus of research in our lab is the development and evaluation of novel "Digital Musical Instruments" (DMI)—using a variety of sensing technologies to provide real-time control over sound synthesis for the purposes of live music performance. In this case, a device may have hundreds of different parameters

J. Malloch
IDMIL, CIRMMT, McGill University
527 Sherbrooke St. West, Montreal, Canada
E-mail: joseph.malloch@mail.mcgill.ca

S. Sinclair
IDMIL, CIRMMT, McGill University
527 Sherbrooke St. West, Montreal, Canada
E-mail: stephen.sinclair@mail.mcgill.ca

M. M. Wanderley
IDMIL, CIRMMT, McGill University
527 Sherbrooke St. West, Montreal, Canada
E-mail: marcelo.wanderley@mcgill.ca

available for input or output, and the difficult questions arise: which of these parameters should control which? How should they be connected? How can technology and aesthetics help us decide? Especially in collaborative undertakings, who makes these decisions?

These questions are just as applicable to the design of any interactive system for which the precise use-cases are not well defined (here, "... for making music" is not considered a complete specification!). In addition to professional artists and researchers exploring this space, there is also a large and growing community of individuals creating interactive systems with readily-available, low-cost sensors and microcontroller platforms such as Arduino [1]. Mobile telephones now commonly contain a substantial collection of sensors for which real-time data are available: accelerometers, gyroscopes, magnetometers, satellite navigation system receivers, microphones, cameras, and ambient light sensors. The "App" ecosystems surrounding various mobile phone platforms (e.g., iOS, Android) also provide exciting opportunities for developers to experiment with creative, real-time control of media synthesis.

In the field of interactive music, a connection or collection of connections between real-time sensor or gesture data and the inputs of media control is commonly referred to by the noun *mapping* [11]. We also use the term as a verb to refer to the activity or process of designing these relationships, as in "... a tool for mapping of digital musical instruments."

A mapping may be as simple as a single connection, or it may consist of an arbitrary number of complex connections limited only by the constraints of computation, communications bandwidth, or the designer's imagination; it may be explicitly designed or implicitly learned by a machine learning algorithm, which might be guided (supervised) by the preferences of a human designer or might represent structure found in the data alone. In typologies of mapping we also commonly distinguish between *convergent* mapping, in which multiple source parameters are combined to control a single destination parameter, and *divergent* mapping, in which a single source is mapped to control multiple destination parameters. The system designer may wish for an instrument mapping to be different for different pieces of music, different performances, or indeed within a single piece. Studies have provided evidence that complex mappings may be preferred by performers [10]—this seems to indicate that simple one-to-one mappings, such as the assignment of a single knob to control a particular sound parameter, can be perceived as less interesting to play as compared to mappings which include mixing of controls signals. It follows that a certain amount of iterative experimentation during interaction design is necessary to achieve a balance that is sensible but does not quickly become boring to play.

[1] http://www.arduino.cc/

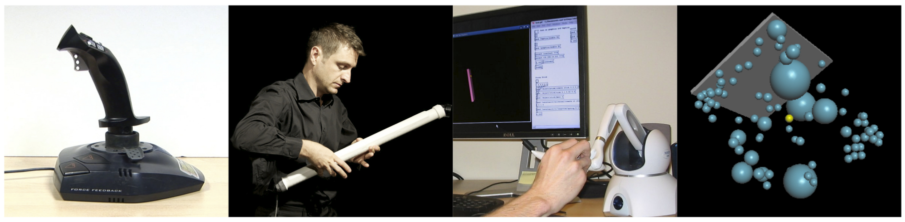


**Fig. 1** Examples of some of the types of devices we wish to flexibly connect to media synthesizers. Clockwise from top–left: commercial computer input devices such as joysticks, "novel" musical instruments such as the T-Stick [18] (photo: Vanessa Yaremchuk), force-feedback/haptic devices, and systems for modelling virtual physical dynamics such as DIMPLE [28].

## 1.1 The Problem

The design of the mapping layer profoundly affects the behavior and perception of the instrument and the experience of both performer and audience; the creation of mappings is an essential part of the design process for a new digital instrument, and is often revisited when composing a new piece of music for a DMI. Crucially, a mapping designer needs to be able to quickly experiment with different mapping configurations while either playing the interface themselves or working closely with collaborating performers.

There is, however, an a priori lack of compatibility between systems: there is no "natural" mapping between a sensor voltage level and a sound parameter. After digitization, one must deal with different data types and rates, different units and ranges, and different approaches to system representation. In practice this usually means that some fairly extensive programming is necessary in order to make different systems and environments intercommunicate. The mapping designer must consider the control space of both source and destination devices, and explicitly devise a scheme for bridging the two. This takes valuable time that could be spent more creatively on the mapping design itself, and restricts the activity of mapping to the relatively smaller group of artist/programmers.

There are a number of established systems and approaches for representing interactive systems, and for communicating control data; some examples are described below in section 1.3. All of these approaches are valid for some systems or scenarios, useful for some users, but does their incompatibility demand further standardization? We argue instead that adding another standard is not helpful, since even if it is deemed successful, not *everyone* will accept it—thus, further fragmentation will result. Worse, there is artistic interest and novelty in building systems that cannot be easily represented within existing standards and established schemas. Since our goal is to support intercommunication of *experimental* interactive systems, imposing further standardization will likely not be helpful.

We argue that what is needed in the research and artistic communities are tools that,

1. allow free reconfiguration and experimentation with the mapping layer during development;
2. provide compatibility between the differing standards (and systems that eschew any standard);
3. and allow the free use of interesting mapping layers between controller and synthesizer (e.g., machine learning, high-dimensional transformations, implicit mapping, dynamic systems, etc.).

Mapping interactive systems is often a time-consuming and difficult part of the design process, and it is appropriate to demand tools and approaches that focus squarely on the mapping task.

### 1.2 Terminology

Here we define some terminology that will aid in describing existing solutions as well as discussing our proposed solution.

Signal
: Data organized into a time series. Conceptually a signal is continuous, however our use of the term *signal* will refer to discretized signals, without assumptions regarding sampling intervals.

Data representation
: How a signal's discretized samples are serialized for the purpose of storage or transmission. This could include the data *type* (e.g., floating point, integer), its *bit depth*, *endianess*, and *vector length* if applicable. It might also include an *identifier* or a *name* used to refer to the data, whether it is included with the stored/transmitted data or defined elsewhere in a specification document.

System representation
: Further information needed to interpret the data, such as its *range*, *unit*, *coordinate system* (e.g., Cartesian, polar, spherical, etc.), and any compression applied to the data values (e.g., logarithmic scaling). At a still higher level, the system representation also includes any assumptions or abstractions applied to the control space for the given system. This might include the level of control a parameter addresses, e.g. directly controlling output media vs. controlling a property of a higher-level model, or even whether a given parameter is exposed at all.
As an example, the MIDI standard includes specification of low-level data transport (7– and 14–bit little-endian integers), and a mid-level stream interpretation (e.g., pitch values represent tempered tuning semitone increments with 69 = A4 = 440Hz). It also includes a high-level control abstraction using the concept of a "note" to represent sounds as temporal objects, each with an explicitly defined pitch, beginning and end—something that is often more ambiguous in acoustic systems.
Another familiar example in audio synthesis is the "ADSR" (attack–decay–sustain–release) control model for the evolution of temporal "envelopes" on

analog synthesizers. This is not the only way to control envelopes, or even the “correct” way, but it has proven useful enough that it is still used on many hardware and software synths. The very use of envelopes is an abstraction of low-level control to a higher-level (i.e. lower-dimensioned or temporally-compressed) control space, and an important detail to consider when describing the system.

## 1.3 Existing Solutions

Since the early 1980s, the Musical Instrument Digital Interface (MIDI) protocol [20] has been the de facto standard for connecting commercial digital musical instruments. Although it is extremely widespread, it has been argued that MIDI has many failings from the perspective of designers of “alternate” music controllers. To cite some examples, MIDI’s bandwidth and data resolution are insufficient [21], messaging semantics are confused [33], its bias to percussive instruments is constraining [19], and it is unsuited for representation of complex control spaces. Although the bandwidth concerns are not strictly tied to the protocol, and extensions such as SKINI have dealt with the limited data types and resolution [5], the protocol’s lack of metadata, reliance on normalization, and inability to well-represent control systems substantially different from piano keyboards make it a bad fit for more generalized mapping needs.

In the academic community, Open Sound Control (OSC) [34] has largely supplanted MIDI as the protocol of choice. Unlike MIDI, however, the OSC specification describes only how to properly format and package bytes to form a named OSC message; although this formatting is transport-agnostic, OSC is almost exclusively transported over packet-switching networks as UDP datagrams. OSC is vastly more flexible than MIDI: it includes support for single- and double-precision floating-point numbers, integers, and 64-bit NTP-formatted absolute timestamps. Most importantly, OSC messages are tagged with a user-specifiable, human-readable string instead of a predetermined controller ID number; thus, simultaneously an advantage and disadvantage, OSC messages are not forced into any higher-level hierarchy or schema.

This lack of standardization for specific OSC message names, the so-called OSC *namespace*, means that while OSC-capable hardware or software can decipher the contents of a message originating elsewhere, there is no guarantee that it will be able to make use of it on a semantic level. The result is that most OSC-capable devices use their own custom protocol running on top of OSC designed by individual device developers. Intercommunication between two OSC-capable devices must typically be provided by consulting the documentation for the receiving device, and specifying the IP address, receiving port, OSC path string, and argument format to the sending device.

Although this can be seen as a disadvantage for reasons of inter-device compatibility, it has nonetheless become evident in our experience, (and to the credit of the designers of the OSC protocol) that device-custom naming schemas allow a level of expression that is far easier to understand for humans. To resolve compatibility problems, two approaches are possible: firstly, *normalization* of namespaces and control-space representation to allow automatic interpretation of a set of "known messages"; or secondly, *translation* of messages from one representation to another.

We argue in this work that the latter is a better choice, and translation of representations is the approach adopted by libmapper. Numerous systems for OSC namespace standardization have been proposed [34, 24, 13, 4, 26, 2, 14], but none has yet been widely adopted. We believe that this is symptomatic of the idea that a one-size-fits-all semantic layer is not the right solution. Not only do representation standards risk leaving semantic gaps that force designers to "shoehorn" their data, as is common with MIDI, but we also argue that normalization, while convenient, discards valuable information about the signal being represented. In fact, we believe that this lack of imposed representation standards is the greatest strength of Open Sound Control over other solutions, and that translation is ultimately a better path to improving compatibility.

### 1.4 Solutions from Other Domains

The (slow) transition from MIDI to OSC mentioned above can be seen as part of a trend in many domains to move legacy dedicated-wire protocols to IP-based systems. In non-musical media control, the legacy DMX512 system for lighting control [7] is gradually being replaced by systems such as the Architecture for Control Networks (ACN) [8]. Meta-data standards for describing the capabilities of devices are often paired with the communication protocol specifications: *Device Description Language* for ACN; *Transducer Electronic Datasheet* for IEEE 1451 [12] and *Transducer Markup Language* [23] for sensor and actuator description. The *Virtual Reality Peripheral Network* (VRPN) [32] functions to allow network-transparent access to control data from various input devices, and also requires a standardized (albeit extensible) representation of devices.

## 2 Translating representations with libmapper

Our approach to solving the problems of standardization and intercommunication is different than previously proposed standardization/normalization-based approaches. Rather than enforcing conventions in the representation of signals (names, ranges, vector lengths, etc.) we simply provide a minimal layer to help devices describe themselves and their capabilities. One reason we prefer description over standardized representation is that our goal is not precisely *automatic* connectivity but rather *flexible* connectivity. Although libmapper can certainly be used to load previously-designed connections in a production-oriented scenario, we wish to emphasize that libmapper is designed for use in the space of mapping design and exploration, not simply connectivity.

While OSC is currently used by libmapper for transporting data, automatic translation is provided to the namespace expected by the destination. Crucially, this means that the sending application need not know the destination's expected OSC namespace—the destination is responsible for announcing this information, and libmapper takes care of translating the sender's messages into a form expected at the destination. This includes both translation of the OSC *path*[2] as well as transformation of the data according to some mathematical expression; typically, this is simply a linear scaling, but can be much more complex if desired.

We do not enforce a particular range convention, and in fact explicitly encourage users to avoid arbitrary normalization. This is further encouraged by providing automatic scaling and type coercion between sources and destinations of data. Thus, the endpoints can be represented in the most logical or intuitive way without worrying about compatibility when it comes time to make mapping connections.

Most importantly, we do not presume to tell the user what this "most logical or intuitive way" might be; instead, we simply try to make it easy for the user to represent their systems modularly and with strong semantics. Further, our position is that redundant representations of the signals from different perspectives are often useful (though perhaps for different people or at different times) [13]; by managing connections, libmapper makes it easy to expose large numbers of parameters for mapping without flooding the network with unused data.

Our approach aims to make the mapping task work transparently across programming languages, operating systems, and computers—the user can choose the language or programming environment best suited for the task at hand. From the instrument designer or programmer's point of view, libmapper provides the following services:

– Decentralized resource allocation and discovery
– Description of entities, including extensible metadata
– Flexible, run-time connectivity between nodes
– Interaction with a semantic abstraction of the network (e.g. connecting devices by name rather than setting network parameters)

The first iterations of these tools were developed to meet the needs of a collaborative instrument development project [16]; since then, we have refined the concepts and functionality, reimplemented parts of the system that were written in other languages in transportable C, added bindings for other popular languages, and added utilities for session management, data visualization, recording and playback. We have offered demonstrations and workshops to test our documentation and solicit feedback (e.g. [17]), and used the system for further projects with other universities and industry.

[2] The OSC path refers to the text string identifying the semantics of an OSC message.

## 3 libmapper concepts

In this section we describe the main features of libmapper, and give our reasoning behind choices made during its conception. The libmapper library itself is used by disparate programs running on a local network, but the collection of these devices can be characterized as a distributed, peer-to-peer communications system based on a representation oriented around named signal streams. It is effectively a metadata, routing and translation protocol built on top of OSC that includes extensive means for describing signals and specifying connections between them over a network.

While the distributed aspect adds complexity as compared to centralized models, this is well-mitigated by a common communications bus and extensive description of signal properties. These make it possible for libmapper programs to find each other automatically, resolve naming conflicts, and make intelligent default decisions during mapping.

### 3.1 Peer to peer communication

As mentioned, for libmapper we have chosen a distributed approach instead of a more centralized network topology. This is not the only choice, and therefore it is necessary to explain our decision.

One possibility for distributing sensor data on a network is the "blackboard" approach: a central server receives data from client nodes, and republishes it to receiving clients that request particular signals. This approach is often taken by online media services such as multiplayer online video games, in which a large number of clients must stay synchronized to a common world state. A multi-layer database back-end used to track and synchronize real-time state updates with an "eventual consistency" approach is often employed today for such applications, with a mirrored-server architecture to distribute the load [6]. This has proven to work well, but is a complex solution made necessary by a strong synchronization requirement. Often, ensuring consistency relies on optimistic updates with roll-back to provide perceived fast response times.

Such a centralized method lends itself well to shared environments, however a real-time musical network more typically requires fast, unsynchronized flow of independent signals from sensors to musical parameters, for which several instances may be present at once—multiple controllers, multiple synthesizers, and multiple performers may share a common studio environment with a single network. MIDI handles this scenario by supporting multiple channels multiplexed over a single serial bus, and, more and more, by providing several MIDI buses and devices in a studio setting, unaware of each other so that they do not interfere. Today, MIDI devices which connect over USB appear to the operating system as their own MIDI device, or even as multiple devices, rather than taking advantage of MIDI's daisy-chaining ability. This means that the user must keep track of which combination of device and channel number represents which physical controller, relying on device drivers to provide meaningful identifiers which are often composed of a concatenation of the product vendor and number. If data from a MIDI device connected to one computer is needed on another, some 3rd-party transport is required, usually specific to the software in use, if such a service is provided at all.

However, an IP-based network actually lends itself well to a peer-to-peer approach, which we have extensively leveraged in the design of libmapper. That is to say, each node on the network can be instructed to send messages to any other node, and any node can issue such an instruction. Additionally, establishment of connections is performed in a stateless manner, and when data transformation is needed, the system is agnostic to where the computation actually takes place.

In our system, data transformation is currently handled by the source node, but since the details of the connection are worked out between the nodes the potential for an alternative agreement is left as a possibility. For example, the receiver could perform all or part of the calculations, or a third-party node could be instructed to process the signal. Computational cost may be considered against hardware capabilities in order to make this decision. This also lends the possibility of agreeing on alternative data transports, such as TCP/IP, or shared memory if the source and destination are on the same host; this latter scenario would be useful for controllers that embed their own audio synthesizer.

## 3.2 A communication bus for "administrative" messages

To enable discovery, it is of course necessary to have a means of communication between all nodes. This is accomplished by having the peer-to-peer data communications run in parallel with a separate bus-oriented architecture for the low-traffic control protocol. Since our target scenario is several computers cooperating on a single subnet, we have found that multicast UDP/IP technology is ideally suited for this purpose. Multicast is used for example by Apple's Bonjour protocol for tasks such as locating printers on the network [1].

It works by network nodes registering themselves as listeners of a special IP address called the *multicast group*. IP packets sent to this group address are reflected by the Rendezvous Point via the Designated Routers to all interested parties [29]. A small multicast network therefore takes the form of a star configuration, however multicast also allows for more complex multi-hop delivery by means of a Time-To-Live (TTL) value attached to each packet. In the case of libmapper, all nodes listen on a standard multicast group and port, and the TTL is set to 1 by default in order to keep traffic local.

We refer to this multicast port as the "admin bus", since it is used as a channel for "administrative communication": publishing metadata, and sending and acknowledging connection requests. The admin bus is used for resolving name collisions, discovering devices and their namespaces, specifying the initial properties of connections, and making modifications to connection properties. It is important to note that no signal data is communicated on the admin bus; signals consist of OSC messages transmitted (most commonly) by unicast UDP/IP directly between device hosts.

#### *3.2.1 Comparison between multicast and other options*

The choice of multicast UDP/IP was not the only possible carrier for this information. Other possibilities include:

1. broadcast UDP messages;
2. a centralized message rebroadcaster;
3. or rebroadcasting of instructions within a mesh network, where each node communicates with a subset of other nodes, and messages are propagated throughout the mesh after several hops.

We chose multicast since, as mentioned, we targeted the use case of a LAN subnet where support for multicast could be guaranteed. Broadcast would also have worked, but we consider multicast to be more "friendly" on a large network, since only interested nodes will receive administrative packets.

The idea of using a mesh is tempting for networks where multicast or broadcast is not available, but the complexity of such an approach was prohibitive and not needed for our applications. However, it is reserved as a future possibility if the need arises. Mesh networks would require the address of a pre-existing node at initialization time, whereas standardizing a multicast port makes it "just work" from the user's point of view. One advantage of mesh networking is to allow the ad-hoc creation of disjoint networks without requiring any special tracking of which multicast ports are available, but use cases for this scenario are, we believe, quite rare.

Another possibility lies between these two extremes: using multicast or broadcast solely for discovery, as in the case of OSCBonjour [22], while sending commands and metadata through a mesh network of reliable TCP connections. Indeed, recent work inclines us towards this solution since we have observed problems of dropped packets when large numbers of signals and devices are present.

The use of multicast has brought to light a distinct lack of support for this useful protocol in OSC-compatible tools. Before developing libmapper as a C library, we added multicast ability to PureData's OSC objects, found bugs in Max/MSP's `net.multi.send` object, and also added multicast to the liblo OSC library, which is used internally by libmapper. With libmapper these improvements were not needed, since libmapper uses liblo directly, and as we provide bindings to libmapper, multicast OSC is handled by the library for all supported bindings. It was nonetheless a useful exercise to provide multicast support in these various environments, and we would like to encourage developers of future audio applications to support the use of multicast.

Finally, since the information carried by the admin bus is essentially a distributed database, we considered the use of decentralized database technologies such as a distributed hash table (DHT) as used by the BitTorrent protocol [31]. This is certainly an interesting option, but a DHT is best suited for information that is evenly and redundantly distributed throughout a set of nodes. In the case of libmapper, each node maintains its own information, and ostensibly this information is useless if the device is unavailable—the expiry of device information is conveniently automatic when a device goes offline. Conversely, deleting expired data from a DHT is a less common operation, and, due to their traditionally public nature, is complicated by trust and security implications which do not apply to local media control networks with which we are concerned here.

## 3.3 Comparison with centralized topology

Of course, a centralized organization facilitates or enables certain actions. For example, recording several data sources to a centralized database requires access to all signals at a single point. Likewise, drawing realtime correlations or other statistical analyses on the set or a subset of signals, useful for gesture recognition or trend identification, similarly requires a global view on the data and therefore a centralized approach is best-suited.[3] In our experience, most systems designed for mapping use this client-server approach (e.g., [2]).

However, the use of a central hub is a subset of the possible connection topologies using libmapper's peer-to-peer approach—it is trivial to model such a topology by simply routing all signals through a central libmapper device. This can be done automatically by a program which monitors the network for new connections and re-routes them through itself. It can then record, analyze, or modify any signals before passing them on to the intended destination. As a proof of concept, we make available a small C program called *greedyMapper* that "steals" existing and future network connections and routes them through itself. When the program exits, it returns the mapping network to its previous peer-to-peer topology. This program is not intended for real use, since we find it preferable to simply create a duplicate of each interesting connection, so that the source device sends the signal once to its mapped destination and once to the recording or data-processing device (Figure 2).

Extending in this manner by doubling the connections, instead of modifying the network to provide a hub, is a less intrusive means to the same end, and results in the same amount of overall network traffic. In particular, when the central device disconnects, it is not necessary to re-route signals back to their original configuration, creating a potential drop-out period, but is enough to simply disconnect the recording pathways. For the same reason, the sudden disappearance of the central device has far less drastic consequences, in the case of a computer crash or power disruption.

[3] Note that there do exist many decentralized statistical analysis approaches, such as graph-based techniques that can distribute successive reduction steps throughout several computational nodes [15]. Application of such techniques to libmapper may be possible, and is the subject of future work.

This connection-doubling technique is used by *mapperRec*, a program that automatically duplicates any connections from a device matching a given specification. Ignoring the connection properties, it maps the data untransformed, and records them directly either to a text file, binary file, or to a PostgreSQL database via *OSCStreamDB* [27]. This can be used for later playback, or to convert to formats appropriate to analysis tools.

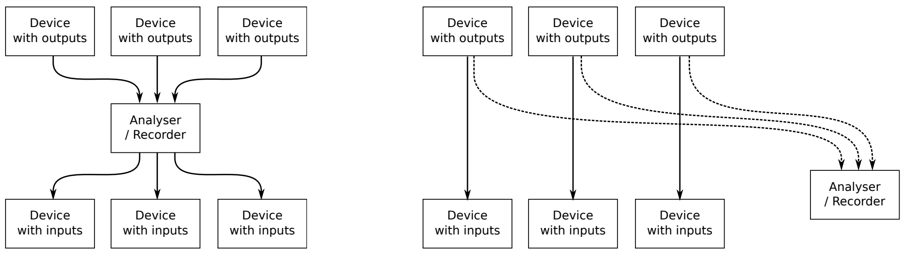


**Fig. 2** A comparison of network topologies: on the left, data recording or analysis is performed at a central location; on the right, most such scenarios can be handled with lower latency by duplicating the existing connections instead.

## 4 Implementation

Conceptually, libmapper is organized into several components and subcomponents that function together to represent network nodes and their functionality. Most of these components are exposed as user-facing interfaces, however not all components are required by all libmapper programs.

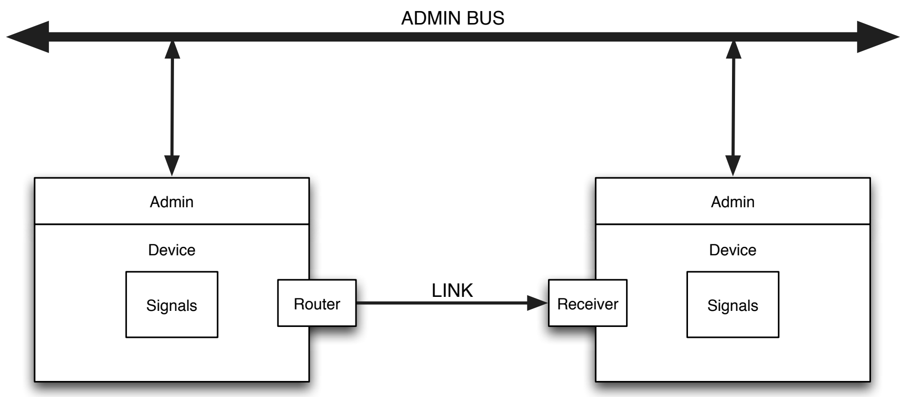


**Fig. 3** libmapper system components: *devices*, *signals*, *routers*, and *links*. *Connections* are contained within links

### *4.0.1 Devices, signals, routers*

Firstly, a network node that can send or receive data is called a *device*. This terminology stems from the original usage scenario where a hardware device is controlled by a single libmapper application, however a *device* may just as easily be a software synthesizer, a data transformation service, a bridge to another bus system, or anything else that may send or receive realtime data. Every device has a name as well as an ordinal which is automatically determined to uniquely identify it if other devices with the same name are found on the network.

A libmapper device must declare its *signals*, which are named value streams available for connecting. Signals have a distinct data direction, i.e. they may either be an input or output, but not both. It is possible in a given device to have one input and one output signal with the same name, however, so this is not a limiting factor. Properties of signals include its name, data type, vector length (see section 4.1), optional data range, and optional units.

We also conceive of an entity known as a *router*. Conceptually, a router is an object to which the device sends all of its signal messages, and it handles transforming these messages into the form specified by existing connections, and sends them as required to their destinations. This decouples the device from connection management, with the router maintaining a list of destination addresses and transformation rules for each signal. The reason for this distinction is to allow data processing and translation to potentially take place on other network nodes, either alone or shared with the sending node. A device contains a list of routers, one for each destination device it is linked to, and this is purely an internal concept—application code never needs to interact with a router directly.

### *4.0.2 Links and Connections*

In practise, a router corresponds to a *link*—a network connection created between two devices. Each router contains the address information (IP address and port) required to send data to its destination device, as negotiated by the link-creation protocol.

The router also keeps a list of *connections* associated with each of the device's output signals mapped to the link's destination device. There is no limit to the number of times a signal may be connected. Connections are specified with information about the source and destination signal names, and any desired data transformation behavior. In addition to linear scaling, data transformation may include user-defined mathematical expressions, automatic calibration, muting, and clipping (See section 5).

*Default connection properties:* The following steps are used to determine initial connection properties, which have been designed to allow fast experimentation provided the data ranges are well-specified:

1. Any properties that are specified as part of the connection will take precedence.

2. Otherwise, any connection preferences specified by the signals involved will be added. Some signals may have default clipping behavior to prevent damage to equipment, for example.
3. If the connection processing is still unspecified and the ranges of the source and destination have been provided, processing will default to linear scaling between the input and output ranges.
4. Otherwise no data transformation is provided; data is "passed through" unaffected. (So-called "bypass" mode.)

In all cases, if the types do not match and type coercion is possible, then data types will be automatically transformed. Type coercion follows the rules of the C language: integers are automatically promoted to floating-point numbers, and conversely floating-point numbers are truncated to integers if necessary.

#### *4.0.3 Monitors*

We also support network nodes called *monitors* which can send and receive administrative messages, but do not have any signals of their own. These nodes are used for observing and managing the network of libmapper-enabled devices, typically in the form of a graphical user interface (see section 6). Due to the fact that libmapper administration is performed on a shared bus, an arbitrary number of monitors can be used simultaneously on a given mapping network, and they can be used from any computer in the local network.

A libmapper *device* can also use the monitor functionality, for example responding to remote activity on the network by dynamically adding or removing from its collection of signals, or by automatically creating links and connections to remote devices when they appear on the network.

### 4.1 Data types

Signals are associated with a particular data type, which must be a homogeneous vector of one or more values. Values may be 32- or 64-bit integers, or 32- or 64-bit floating-point numbers, for example.

The choice to support only homogeneous vector types may be seen as inflexible. Indeed, many aspects of libmapper could be adapted to support heterogeneous vectors, however we chose to support only homogeneous vectors because signals are not intended to represent *data structures*, but rather values associated with properties of a system. The reason for introducing vectors is that certain values *are* vectors from a semantic point of view, but heterogeneous types imply something that can be broken up into pieces. We wanted to encourage as much as possible the use of *short* vectors, limited to, for example, 2- or 3D position data, but not used to organize an entire system state vector. Rather, such a structure representing a system state should be split up into its components, each as a separate signal (Figure 4).

```
/tuio/2Dobj set s i x y a X Y A m r

/session/object/id/position x y
/session/object/id/angle a
/session/object/id/velocity X Y
/session/object/id/angular_velocity A
/session/object/id/acceleration m
/session/object/id/angular_acceleration r
```

**Fig. 4** Top: example data encoding specification from the TUIO protocol [14] using a large heterogeneous vector to carry the entire state of an object. Below: the same data exposed as separate "signals" with semantically strong labels and only short, homogenous vectors where appropriate.

To enable flexible mapping design, it is necessary to access as much as possible individual pieces of information that can be mixed and matched by the user on the fly. Specifying large amounts of data at particular indexes of a large state vector is comparable to "hiding" the natural-language semantic specification enabled by Open Sound Control, and we feel such a choice sacrifices one of the main advantages of OSC over MIDI. Being able to assume homogeneous vector types also allows more succinct expression of element-wise mathematic functions.

Even 3D position data may be usefully represented as separate $(x, y, z)$ components, but some values such as quaternions have terms that are rarely referred to individually. At this point the reader may be asking, why not also support a matrix type? Indeed, why not types for multidimensional tensors? There is certainly a case to be made for higher-dimensioned arrays, such as transmission of pixel data for example, or handling of rotation matrices. However, from a practical standpoint we felt that it was necessary to draw the line somewhere, as libmapper—intended as a lightweight library—cannot provide a full scientific programming language for data processing. Moreover, it is often inefficient to transmit large bundles of data in real-time, and it is preferable to extract properties of this data and expose these as signals.

As an example, a video feed could be transmitted as a 2D matrix signal, but a bare video feed has little use for audio control. It is far more efficient and useful to extract video features such as body or face position using computer vision techniques, and to transmit signals such as "/eye/left/position" as 2-valued vectors.

We believe these arguments are sound, but support for high-dimensioned data is not entirely out of the question, and could perhaps be added in the future if the need arises. In the meantime, matrices can of course be transmitted as flattened vectors, which scalar support alone would not have allowed.

## 4.2 Metadata

Several properties of connections were mentioned in section 4.0.2. These included the data transformation expression, the connection mode, and the boundary behavior. Devices and signals also have properties, such as their name, type, length, and range.

In libmapper, this metadata can be extended by the user to include any extra information that may be useful for visualization or analysis of the network. Devices and signals may be extended with named values that can be of any data type supported by OSC.

For example, the user may wish to assign position information to each device in order to identify its physical location in the room. This may aid in the development of a visualization tool for the design of an art installation. Key-value pairs `x=`$x$ and `y=`$y$ may be used for this purpose. In other cases perhaps location is more meaningfully communicated by indicating the name of the room in which a device is situated, or perhaps the name of the person to which a wearable device is attached.

Another example might be to mark certain connections as special, for example because they are related to a recording device. When displaying the connected network topology, it may be desirable to avoid including such connections in order to simplify the visual display. Alternatively names could be used to semantically group certain collections of devices as belonging to a particular user, or being components in a particular subsystem. Since the admin bus is shared by all devices, identification of who made which connection could be important for collaborative scenarios.

## 4.3 Queries

In addition to explicit connections between source and destination signals, an entirely different mapping context is possible: statistical learning of desired mappings based on examples (supervised learning), or based on data analysis (unsupervised learning). Although implementation details and usage of this "implicit" approach to mapping are outside the scope of this article, its basic requirements should be supported by a mapping system.

In particular, for the supervised approach, it is necessary to learn the value of the destination signals in order to train the system—information that is not available through unidirectional mapping connections. Often, the value of the destination would have been *set* by the connection in question, but the value could also have been changed by another libmapper peer or locally by a user or by some automated process (as in the case of *play-along learning* [9]).

Two facilities are provided by libmapper for this purpose. Firstly, it is possible to query the value of remote signals. A *monitor* can query remote signals directly, since it has access to information about the mapping network (the existence and network location of the remote signal, for example), but a *device* has knowledge only of its own signals. In the libmapper application programming interface (API), remote values can still be retrieved by calling a remote query function on a local output signal; libmapper will query the remote ends of any mapping connections originating from the specified local output signal and return the number of queries sent out on the network.

In order to process the responses to value queries, the local device must register a query response handler function for the local output signal. The query protocol can also handle the case in which a queried signal does not yet have a value.

Secondly, libmapper also provides the ability to reverse the flow of data on a mapped connection. When this behaviour is enabled, every update of the destination signal, whether updated locally or remotely by another connection, causes the updated value to be sent "upstream" to the connection source. In this case no signal processing is performed other than coercing the data type if necessary. The sampled destination values can then be used to establish interpolation schemes or to calculate errors for supervised training (Figure 5).

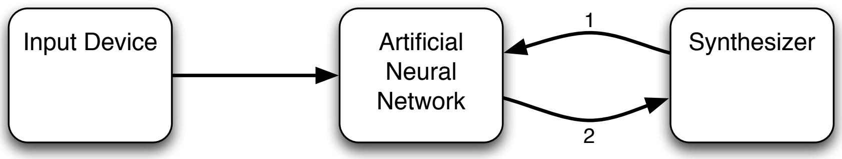


**Fig. 5** Example "supervised" implicit mapping scenario: connections from an input device are routed through an intermediate device rather than directly to the synthesizer. During training, the values of connected destination input signals are sent upstream to the intermediate device (1) using either individual value queries or "reverse"-mode connections. After training, the connections from intermediate to destination devices are reset to "bypass" mode (2). Note that the arrows marked (1) and (2) actually represent the same data structures; only the dataflow direction changes. A typical implicit mapping scenario might use many such connections rather than the simplification shown here.

## 5 Signal Processing

The first two connection modes ("linear" and "bypass") have already been described in the context of determining default behaviors for new connections. For simple use, the "linear" mode may suffice if the signal ranges have been well defined; for more advanced usage two other options are provided.

The third connection mode is called *calibrate*, and it can only be enabled if the destination range has been defined. While a connection is in this mode, libmapper will keep track of the source signal extrema (minimum and maximum values) and use them to dynamically adjust the linear scaling between source and destination. Re-entering *linear* mode has the effect of ceasing calibration while keeping the recorded extrema as the source range. Ranges are stored independently for each connection, and can be added or edited as part of the connection metadata; different connections can thus map to different sub-ranges of the destination, for example.

The final connection mode is *expression*, in which the updated source values are evaluated using a user-defined mathematical expression in the form "y=x". This expression can contain arithmetic, comparison, logical, or bitwise

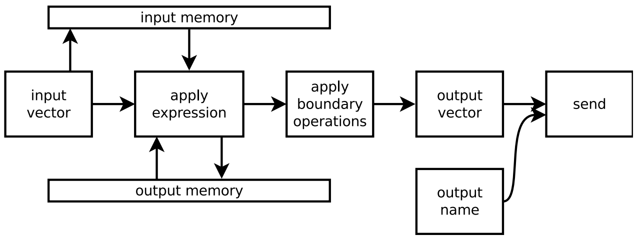


**Fig. 6** Processing pipeline used by libmapper.

operators. The linear and calibrate modes automatically populate the expression metadata, so that when switching into expression mode the user can start by editing the expression defining the previous mapping.

In addition to the active modes mentioned, each connection can be independently muted, allowing users to temporarily prevent data transmission without losing the connection state.

## 5.1 Indexing delayed samples

Past samples of both input and output are also available for use in expressions, allowing the construction of FIR and IIR digital filters (Figure 6). These values are accessed using a special syntax; some examples are shown in Table 5.1. We currently limit addressing to a maximum of 100 samples in the past.

| Function | Expression Syntax |
|---|---|
| Differentiator | `y = x - x{-1}` |
| Integrator | `y = x + y{-1}` |
| Exponential moving average | `y = x * 0.01 + y{-1} * 0.99` |
| Counter | `y = y{-1} + 1` |

**Table 1** Some example expressions using indexing of delayed samples.

## 5.2 Boundary Actions

A separate "boundary" stage is provided for constraining the output range after the expressions are evaluated. Actions can be controlled separately at the minimum and maximum boundaries provided, and can take one of five different modes:

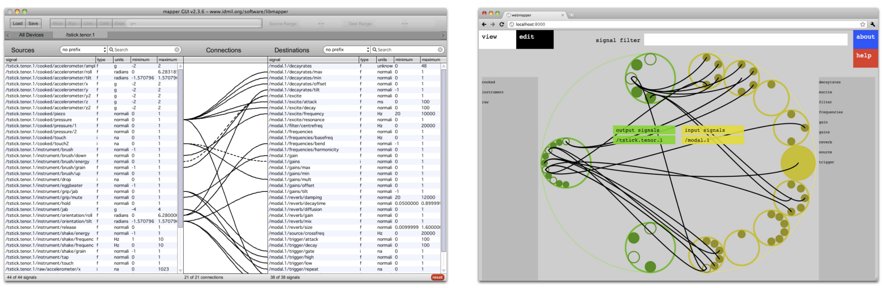
none – values outside of the bound are passed through unchanged.
clamp – values outside the bound are constrained to the bound.
mute – values outside the bound are not passed to the output.
wrap – values outside the bound are "wrapped" around to the other bound.
fold – values outside the bound are reflected back towards the other bound

## 6 Mapping Session Management

When designing mappings between libmapper devices it is necessary to interact somehow with the network (via the admin bus); typically this is done using one of the graphical user interfaces (GUI) we have developed in parallel with the library itself. These interfaces are also used to save and load pre-prepared mappings, for example when performing a piece in concert.

### 6.1 Graphical User Interfaces

Currently our working GUIs use a *bipartite graph* representation of the connections, in which sources of data appear on the left-hand side of the visualization and destinations or sinks for data appear on the right (Figure 7). Lines representing inter-device *links* and inter-signal *connections* may be drawn between the entities on each side, and properties are set by first selecting the connection(s) to work on and then setting properties in a separate "edit bar". They use a multi-tab interface in which the leftmost tab always displays the network overview (links between devices) and subsequent tabs provide sub-graph representations of the connections belonging to a specific linked device.



| signal | type | units | minimum | maximum |
|---|---|---|---|---|
| /tstick.tenor.1/cooked/accelerometer/ampl | f | g | -2 | 2 |
| /tstick.tenor.1/cooked/accelerometer/roll | f | radians | 0 | 6.28318 |
| /tstick.tenor.1/cooked/accelerometer/tilt | f | radians | -1.570796 | 1.57079 |
| /tstick.tenor.1/cooked/accelerometer/x | f | g | -2 | 2 |
| /tstick.tenor.1/cooked/accelerometer/y | f | g | -2 | 2 |
| /tstick.tenor.1/cooked/accelerometer/y2 | f | g | -2 | 2 |
| /tstick.tenor.1/cooked/accelerometer/z | f | g | -2 | 2 |
| /tstick.tenor.1/cooked/accelerometer/z2 | f | g | -2 | 2 |
| /tstick.tenor.1/cooked/piezo | f | normali | 0 | 1 |
| /tstick.tenor.1/cooked/pressure | f | normali | 0 | 1 |
| /tstick.tenor.1/cooked/pressure/1 | f | normali | 0 | 1 |
| /tstick.tenor.1/cooked/pressure/2 | f | normali | 0 | 1 |
| /tstick.tenor.1/cooked/touch | i | na | 0 | 1 |
| /tstick.tenor.1/cooked/touch2 | i | na | 0 | 1 |
| /tstick.tenor.1/instrument/brush | f | normali | -1 | 1 |
| /tstick.tenor.1/instrument/brush/down | f | normali | 0 | 1 |
| /tstick.tenor.1/instrument/brush/energy | f | normali | 0 | 1 |
| /tstick.tenor.1/instrument/brush/grain | f | normali | -1 | 1 |
| /tstick.tenor.1/instrument/brush/up | f | normali | 0 | 1 |
| /tstick.tenor.1/instrument/drop | i | na | 0 | 1 |
| /tstick.tenor.1/instrument/eggbeater | f | normali | -1 | 1 |
| /tstick.tenor.1/instrument/grip/jab | f | normali | 0 | 1 |
| /tstick.tenor.1/instrument/grip/mute | f | normali | 0 | 1 |
| /tstick.tenor.1/instrument/hold | f | normali | 0 | 1 |
| /tstick.tenor.1/instrument/jab | f | g | -4 | 4 |
| /tstick.tenor.1/instrument/orientation/roll | f | radians | 0 | 6.28000 |
| /tstick.tenor.1/instrument/orientation/tilt | f | radians | -1.570796 | 1.57079 |
| /tstick.tenor.1/instrument/release | f | normali | 0 | 1 |
| /tstick.tenor.1/instrument/shake/energy | f | normali | -1 | 1 |
| /tstick.tenor.1/instrument/shake/frequenc | f | Hz | 1 | 10 |
| /tstick.tenor.1/instrument/shake/frequenc | f | Hz | 0 | 10 |
| /tstick.tenor.1/instrument/shake/grain | f | normali | -1 | 1 |
| /tstick.tenor.1/instrument/tap | f | normali | 0 | 1 |
| /tstick.tenor.1/instrument/touch | f | normali | 0 | 1 |
| /tstick.tenor.1/raw/accelerometer/x | i | na | 0 | 1023 |

| signal | type | units | minimum | maximum |
|---|---|---|---|---|
| /modal.1/decayrates | f | unknow | 0 | 48 |
| /modal.1/decayrates/max | f | normali | 0 | 1 |
| /modal.1/decayrates/min | f | normali | 0 | 1 |
| /modal.1/decayrates/offset | f | normali | 0 | 1 |
| /modal.1/decayrates/tilt | f | normali | -1 | 1 |
| /modal.1/excite | f | normali | 0 | 1 |
| /modal.1/excite/attack | f | ms | 0 | 100 |
| /modal.1/excite/decay | f | normali | 0 | 100 |
| /modal.1/excite/frequency | f | Hz | 20 | 10000 |
| /modal.1/excite/resonance | f | normali | 0 | 1 |
| /modal.1/filter/centrefreq | f | Hz | 0 | 20000 |
| /modal.1/frequencies | f | normali | 0 | 1 |
| /modal.1/frequencies/basefreq | f | Hz | 0 | 1 |
| /modal.1/frequencies/bend | f | normali | -1 | 1 |
| /modal.1/frequencies/harmonicity | f | normali | 0 | 1 |
| /modal.1/gain | f | normali | 0 | 1 |
| /modal.1/gains | i | normali | 0 | 1 |
| /modal.1/gains/max | f | normali | 0 | 1 |
| /modal.1/gains/min | f | normali | 0 | 1 |
| /modal.1/gains/mult | f | normali | 0 | 1 |
| /modal.1/gains/offset | f | normali | 0 | 1 |
| /modal.1/gains/tilt | f | normali | -1 | 1 |
| /modal.1/reverb/damping | f | normali | 20 | 12000 |
| /modal.1/reverb/decaytime | f | normali | 0.0500000 | 0.89999 |
| /modal.1/reverb/diffusion | f | normali | 0 | 1 |
| /modal.1/reverb/gain | f | normali | 0 | 1 |
| /modal.1/reverb/mix | f | normali | 0 | 1 |
| /modal.1/reverb/size | f | normali | 0.0099999 | 1.60000 |
| /modal.1/source/crossfreq | f | Hz | 0 | 20000 |
| /modal.1/trigger/attack | f | normali | 0 | 100 |
| /modal.1/trigger/decay | f | normali | 0 | 100 |
| /modal.1/trigger/gate | i | na | 0 | 1 |
| /modal.1/trigger/high | f | normali | 0 | 1 |
| /modal.1/trigger/low | f | normali | 0 | 1 |
| /modal.1/trigger/repeat | i | na | 0 | 1 |

**Fig. 7** Two different graphical user interfaces for managing the mapping network.

We have also explored alternative visualization and interaction techniques, which allow more informed and flexible interaction with the mapping network. Crucially, we believe that there is no need for a single "correct" user interface; rather, different network representations and interaction approaches may be useful to different users, for different mapping tasks, or at different times.

All libmapper GUIs function as "dumb terminals"—no handling of mapping connection commands takes place in the GUI, but rather they are only responsible for representing the current state of the network links and connections, and issuing commands on behalf of the user. This means that an arbitrary number of GUIs can be open simultaneously supporting both remote network management and collaborative creation and editing during the mapping task. This approach has brought our attention to interesting research into other collaboratively-edited systems (e.g. [25]); our approach to undo/redo functionality is based on this work.

### 6.2 Automatic Session Recovery

Apart from explicit saving and loading, for specific projects we commonly implement small session-recovery programs to save time during working sessions. Sometimes while programming a particular device it is necessary to frequently recompile and relaunch the device, and recreating the mapping links and connections would be tedious. Using the *monitor* API, it only takes a few minutes to write a small program that watches for specific relaunched devices and recreates the desired mapping.

Eventually, we plan to create a general-purpose session logging and recovery system based on libmapper, backed by a version control database allowing the recovery of any previous configuration of the mapping network.

## 7 Mapping scenarios—libmapper use cases

In this section we briefly describe some examples of use cases for which libmapper has been designed.

### 7.1 Explicit Mapping Scenario

Imagine that Bob has an interesting synthesizer, and that Sally has been working on a novel physical input device. Since they both used libmapper bindings, collaboration is simple—they use a GUI to view the mapping network and create a link between the two devices. Over the course of their session, they experiment with different mappings by creating and editing connections between the outputs of Sally's controller and the inputs of Bob's synthesizer.

### 7.2 Implicit Mapping Scenario

Sally and Bob decide to try a different approach: they are happy with some of the results from the explicit mapping approach but they want to jump quickly to controlling more parameters. They remove the direct link between their devices in the mapper network and instead link them through an intermediate machine-learning module/device and connect the signals they want to include in the mapping. Sally clicks on the button labeled "snapshot" on the intermediate device each time she wants to indicate that the current combination of gestural data and synthesizer configuration should be associated. Lastly, she clicks on a button labeled "process", and the device begins performing N-to-M-dimensional mapping from the controller signals to the synth signals.

Finally, Sally and Bob might decide to use a combination of the approaches outlined above, in which many "integral" parameters of the synth affecting the timbre of the resulting sound are mapped implicitly using machine learning, but the articulation of sounds (attacks, releases) and their overall volume are controlled using explicitly chosen mapping connections. They decide that for this particular project this approach provides the best balance between complex control and deterministic playability.

## 8 Support for Programming Languages and Environments

libmapper is written in the programming language C, making it usable as-is in C-like languages such as C++ and Objective-C, and by users of media programming platforms such as openFrameworks[4] and cinder[5]. Bindings for the Python and Java programming languages are also provided, the latter intended principally for compatibility with the Processing programming language which is popular in the visual digital arts community.

We have made every effort to ensure that the libmapper API is simple and easy to integrate into existing software. In most programs, the user-code must make only four calls to the libmapper API:

1. Declare a device (optionally with some metadata)
2. Add one or more signals (inputs and/or outputs, again with some optional metadata)
3. Update the outputs with new values
4. Call a polling function, which processes inputs and calls handlers when updates are received.

Figure 8 shows a simple (but complete) program using the libmapper C API. For simplicity here we will not show the monitor functionality, since for the most part only GUIs and other session managers need to instantiate monitors. Full API documentation is available online.

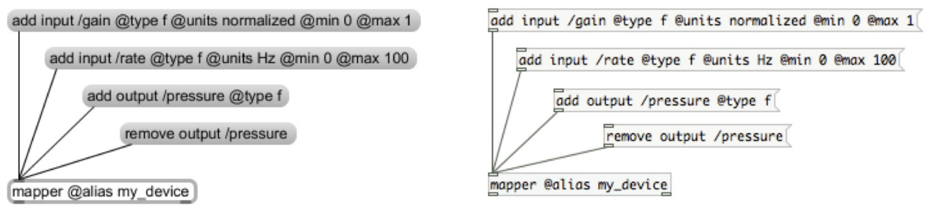
### 8.1 Max/MSP and Pure Data

Max/MSP and PureData—two graphical patching environments for music programming—are supported via a `mapper` external object. This object instantiates a libmapper device with the name of the object's argument, and allows input and output signals to be added or removed using messages sent to the object. Output signals are updated by simply routing the new value into the object's inlet, and received inputs emerge from the object's left outlet. The object's metadata (IP, port, unique name, etc) are reported from the object's right outlet (Figure 9).

[4] http://www.openframeworks.cc/
[5] http://libcinder.org/

```
#include "mapper/mapper.h"

void handler (mapper_signal msig, mapper_db_signal props, int instance_id,
              void *value, int count, mapper_timetag_t *tt ) {
  // do something
}

void main() {
  mapper_device dev = mdev_new("my_device", 0, 0);
  mapper_signal in = mdev_add_input(dev, "/out", 1, 'i', 0, 0, 0, handler);
  mapper_signal out = mdev_add_output(dev, "/out", 1, 'i', 0, 0, 0);

  int my_value = 0;
  while(my_value < 1000) {
    msig_update_int(out, my_value++);
    mdev_poll(dev, 100);
  }

  mdev_free(dev);
}
```

**Fig. 8** Simple program using the libmapper C API to declare one input and one output signal.



```
add input /gain @type f @units normalized @min 0 @max 1
add input /rate @type f @units Hz @min 0 @max 100
add output /pressure @type f
remove output /pressure
mapper @alias my_device
```

```
add input /gain @type f @units normalized @min 0 @max 1
add input /rate @type f @units Hz @min 0 @max 100
add output /pressure @type f
remove output /pressure
mapper @alias my_device
```

**Fig. 9** Screenshots of the `mapper` object for Max/MSP (left) and PureData (right).

The `mapper` external object also provides an opportunity to provide parameter bindings for the popular music sequencing and production software Ableton Live[6], since it is possible to load Max/MSP patches as plugins in Ableton. Using the provided support for parameter discovery, one can quickly scan the open project and declare all the Ableton parameters as mappable signals on the network.

We also provide the `implicitmap` external object as a reference implementation for support of implicit mapping techniques with libmapper, and was built specifically for bridging libmapper and the MnM mapping tools from IRCAM [3]. This object instantiates both a libmapper *device* and a *monitor*, which it uses to observe its own connections and dynamically adjust its number of inputs and outputs as necessary.

[6] https://www.ableton.com/

## 9 Device and Software Support for libmapper

The most complex devices we use for mapping are invariably the ones developed in our lab—complex in terms of numbers of signals available for mapping, and also in terms of their departure from standard input devices such as piano keyboards. There are relatively few commercial offerings in the "alternate music controller" space. In addition to the lab-developed prototypes and research instruments, we also use a number of commercial input devices such as joysticks, depth-sensing camera systems, and various optical, magnetic, and inertial motion capture systems. We maintain a public collection of device drivers and utilities for working with libmapper that we hope will grow alongside the community making use of the library.

### 9.1 Protocol Bridges

In the interests of compatibility with other communication standards and with legacy hardware, we are developing a series of software daemons that function as protocol bridges to the libmapper mapping network. The MIDI protocol in particular is of interest to us, since almost all music hardware built since the 1980s is compatible with MIDI, and it remains the standard for most commercial music software today. Our MIDI bridge makes use of libmapper and the open-source cross-platform MIDI library RtMidi[7] to expose MIDI signals to the mapping network. If the software is running as a daemon, any MIDI devices recognized by the operating system will be dynamically added to the available pool of mappable devices without any user intervention.

Popular computer peripheral input devices such as gaming joysticks and graphics tablets are also commonly used for multidimensional control of music software. For this reason, we are also developing a software daemon for automatically exposing the parameters of peripherals using the Human Interface Device (HID) standard for mapping.

Finally, the Arduino microcontroller platform[8] is immensely popular for creating DIY electronics in general, and new control interfaces for music in particular. Students in our lab frequently use Arduino circuit boards for sampling sensors and communicating with a computer running audio synthesis in PureData or Max/MSP; usually custom firmware is written to enable the microcontroller to efficiently perform specific tasks, however a generic firmware — "Firmata" — is sometimes used instead to allow dynamic reconfiguration of the Arduino at run-time [30]. For users of Firmata, we make available an adaptation of the application "firmata_test" [9] that exposes the configured pins for mapping [10].

[7] http://www.music.mcgill.ca/~gary/rtmidi/
[8] http://www.arduino.cc/
[9] http://firmata.org/wiki/Main_Page
[10] https://github.com/IDMIL/firmata-mapper

### *9.1.1 Sending and Receiving Open Sound Control from libmapper*

Although we argue that calling libmapper from your application is the best route towards intercommunication, libmapper-enabled devices/applications can be used to send and receive Open Sound Control messages for compatibility with tools lacking libmapper bindings but including OSC support. Sending OSC messages from a libmapper-enabled application to an OSC-only application simply involves manually providing enough routing information (receiving IP and port and the expected OSC message path) to create a dummy libmapper connection. Sending messages from an OSC-only application to libmapper is also simple; since libmapper uses OSC internally for message passing one can simply send properly-formatted OSC messages to the destination device. The routing information, OSC paths, data types and vector lengths for the destination signals can be easily inspected using any mapper GUI on the network.

Naturally, these tricks only work for fairly simple scenarios, and will not support device discovery, value queries, or much of the session management features of libmapper. However, in the libmapper-to-OSC scenario the spoofed connection will support signal processing and session management.

## 10 Conclusions and Future Work

In conclusion, we believe that designers of interactive systems should use the most suitable, interesting representations of their systems, and that translation should be used for providing compatibility rather than standardization and normalization of signals. We offer libmapper to the community as tools for accomplishing this goal: a cross-platform, open-source software library with bindings for an increasing number of popular programming languages, platforms, and environments. The future of libmapper includes lots of exciting work for anybody interested in aiding or guiding development—a few of the items on our roadmap are described in the next section.

Our own experience using libmapper for system interconnections on a variety of large and small projects has been largely positive. While libmapper does not miraculously make mapping easy (of course!), it greatly streamlines our work. Systems are now much less tedious to set up and interconnect, even when working with student projects that were not conceived to be compatible in any way. The use of libmapper and surrounding tools also acts to democratize collaborative mapping sessions, since participants can experiment with mappings between systems written in unfamiliar programming languages, and often even while the devices themselves are being modified. Most importantly, we find that the time we save means that we can try more ideas in the available workshop time, easily compare and contrast them, iterate and refine them. We use libmapper in workshops, demos, concert performances, and daily in our lab.

### 10.1 Future Work

Current work on libmapper focuses on finalizing the function and programming interfaces for dealing with *instances* of signals, for the handling of polyphony in synthesizers as well as “blob tracking” and other methods that involve virtual entities that pop in and out of existence. This work will be treated in a future publication.

In tandem with the *instances* functionality, we are progressing on the full integration of absolute time-tagging of mapped data. Once the groundwork is finished, this functionality will enable automatic jitter-mitigation in data streams since libmapper will be able to dynamically determine the amount of latency in a given link or connection. Perhaps more interestingly, connection processing will be extended to allow arbitrary processing of the message time-tags, enabling flexible delays, debouncing, resampling, or ramping to be part of the designed mapping connections.

Finally, we are continuously working on adding support for new programming languages and input devices, including progress on interfacing libmapper with embedded systems.

## 11 More Information

More information on libmapper and related projects can be found on the project website `libmapper.org` or by subscribing to one of the mailing lists (developer or user). The website includes online documentation of the libmapper application programming interface, pre-built binary versions of the library for various platforms, and links to projects and utilities using libmapper. All libmapper development is performed in open consultation with the community mailing list, and anyone interested can join to participate in defining and implementing the future of the library and its surrounding ecosystem.

**Acknowledgements** Development of *libmapper* has been supported by funds from the Fonds de recherche sur la société et la culture (FQRSC) of the Quebec government, the Natural Sciences and Engineering Research Council of Canada (NSERC), and the NSERC/Canada Council for the Arts New Media Initiative.

## References

1. Apple: Bonjour printing specifications (2005). Available: `https://developer.apple.com/bonjour/printing-specification/bonjourprinting-1.0.2.pdf`. Accessed March 2013.
2. Baalman, M.A., Smoak, H.C., Salter, C.L., Malloch, J., Wanderley, M.M.: Sharing data in collaborative, interactive performances: the SenseWorld DataNetwork. In: Proc. New Interfaces for Musical Expression, pp. 131–134. Pittsburgh, USA (2009)
3. Bevilacqua, F., Müller, R., Schnell, N.: MnM: a Max/MSP mapping toolbox. In: Proc. New Interfaces for Musical Expression, pp. 85–88. University of British Columbia, Vancouver, Canada (2005)

4. Bullock, J., Frisk, H.: The Integra framework for rapid modular audio application development. In: Proc. International Computer Music Conference. University of Huddersfield, Huddersfield, UK (2011)
5. Cook, P.R., Scavone, G.: The synthesis toolkit (STK). In: Proc. International Computer Music Conference, pp. 164–166 (1999)
6. E. Cronin A. R. Kurc, B.F.: An efficient synchronization mechanism for mirrored game architectures. Multimedia tools and applications **23**(1), 7—30 (2004)
7. ESTA: ANSI E1.11-2004 Entertainment technology USITT DMX512-A asynchronous serial digital data transmission standard for controlling lighting equipment and accessories. Online (2004)
8. ESTA: ANSI E1.17-2010 Architecture for control networks device description language (DDL). Online (2011). Available: http://tsp.plasa.org/tsp/documents/docs/ACN-ddl_2009-1024r1_free.pdf. Accessed June 10, 2012.
9. Fiebrink, R.: Real-time human interaction with supervised learning algorithms for music composition and performance. Ph.D. thesis, Princeton University, Princeton, NJ, USA (2011)
10. Hunt, A.: Radical user interfaces for real-time musical control. Ph.D. thesis, University of York, UK (1999)
11. Hunt, A., Wanderley, M.M.: Mapping performance parameters to synthesis engines. Organised Sound **7**(2), 97–108 (2002)
12. IEEE: IEEE Standard for a smart transducer interface for sensors and actuators wireless communication protocols and transducer electronic data sheet (TEDS) formats. IEEE Std 1451.5-2007 (2007). DOI 10.1109/IEEESTD.2007.4346346
13. Jensenius, A.R., Kvifte, T., Godøy, R.I.: Towards a gesture description interchange format. In: Proc. New Interfaces for Musical Expression, pp. 176–179. IRCAM—Centre Pompidou (2006)
14. Kaltenbrunner, M., Bovermann, T., Bencina, R., Costanza, E.: TUIO - a protocol for table-top tangible user interfaces. In: Proc. of the 6th International Workshop on Gesture in Human-Computer Interaction and Simulation. Vannes, France (2005)
15. Karl, H., Willig, A.: Protocols and architectures for wireless sensor networks. Wiley-Interscience (2007)
16. Malloch, J., Sinclair, S., Wanderley, M.M.: A network-based framework for collaborative development and performance of digital musical instruments. In: R. Kronland-Martinet, S. Ystad, K. Jensen (eds.) Computer Music Modeling and Retrieval - Sense of Sounds, *LNCS*, vol. 4969, pp. 401–425. Springer-Verlag, Berlin / Heidelberg (2008)
17. Malloch, J., Sinclair, S., Wanderley, M.M.: Libmapper (a library for connecting things). In: Proc. of the International Conference on Human Factors in Computing Systems (CHI2013), pp. 3087–3090. ACM, New York, NY, USA (2013). Extended abstract
18. Malloch, J., Wanderley, M.M.: The T-Stick: From musical interface to musical instrument. In: Proc. New Interfaces for Musical Expression. New York City, USA (2007)
19. McMillen, K.: ZIPI: Origins and motivations. Computer Music Journal **18**(4), 47–51 (1994)
20. MMA: The complete MIDI 1.0 detailed specification: Incorporating all recommended practices (1996)
21. Moore, F.R.: The dysfunctions of MIDI. Computer Music Journal **12**(1), 19–28 (1988)
22. Muller, R.: OSCBonjour (2006). Available: http://recherche.ircam.fr/equipes/temps-reel/movement/muller/index.php?entry=entry060616-173626. Accessed March 2013.
23. Open Geospatial Consortium: Transducer markup language. Online (2006). Available: http://www.ogcnetwork.net/infomodels/tml. Accessed September 12, 2009.
24. Place, T., Lossius, T.: Jamoma: A modular standard for structuring patches in Max. In: Proc. International Computer Music Conference. Tulane University, New Orleans, LA (2006)
25. Ressel, M., Gunzenhäuser, R.: Reducing the problems of group undo. In: Proc. of the international ACM SIGGROUP conference on Supporting group work, GROUP '99, pp. 131–139. ACM, New York, NY, USA (1999). DOI 10.1145/320297.320312. URL http://doi.acm.org/10.1145/320297.320312
26. Schiesser, S.: midOSC: a Gumstix-based MIDI-to-OSC converter. In: Proc. New Interfaces for Musical Expression, pp. 165–168 (2009)

27. Schmeder, A.: Efficient gesture storage and retrieval for multiple applications using a relational data model of open sound control. In: Proc. International Computer Music Conference (2009)
28. Sinclair, S., Wanderley, M.M.: A run-time programmable simulator to enable multi-modal interaction with rigid body systems. Interacting with Computers **21**(1–2), 54–63 (2009)
29. Soucy, R.P.: IP multicast explained (2012). Available: http://www.soucy.org/network/multicast.php. Accessed March 2013.
30. Steiner, H.C.: Firmata: Towards making microcontrollers act like extensions of the computer. In: Proc. New Interfaces for Musical Expression, pp. 125–130. Carnegie Mellon University, Pittsburgh, PA (2009)
31. Steinmetz, R., Wehrle, K.: Peer-to-peer systems and applications, vol. 3485. Springer (2005)
32. Taylor II, R.M., Hudson, T.C., Seeger, A., Weber, H., Juliano, J., Helser, A.T.: VRPN: a device-independent, network-transparent vr peripheral system. In: Proc. of the ACM symposium on Virtual reality software and technology, pp. 55–61. ACM (2001)
33. Wright, M.: A comparison of MIDI and ZIPI. Computer Music Journal **18**(4), 86–91 (1994)
34. Wright, M., Freed, A., Momeni, A.: OpenSound Control: State of the art 2003. In: Proc. New Interfaces for Musical Expression, pp. 153–160. McGill University, Montreal, Canada (2003)