A Network-Based Framework for Collaborative Development and Performance of Digital Musical Instruments

Joseph Malloch, Stephen Sinclair, and Marcelo M. Wanderley

Input Devices and Music Interaction Laboratory
Centre for Interdisciplinary Research in Music Media and Technology
McGill University – Montreal, QC, Canada
joseph.malloch@mcgill.ca, sinclair@music.mcgill.ca,
marcelo.wanderley@mcgill.ca

Abstract. This paper describes the design and implementation of a framework designed to aid collaborative development of a digital musical instrument mapping layer[1]. The goal was to create a system that allows mapping between controller and sound parameters without requiring a high level of technical knowledge, and which needs minimal manual intervention for tasks such as configuring the network and assigning identifiers to devices. Ease of implementation was also considered, to encourage future developers of devices to adopt a compatible protocol.

System development included the design of a decentralized network for the management of peer-to-peer data connections using OpenSound Control. Example implementations were constructed using several different programming languages and environments. A graphical user interface for dynamically creating, modifying, and destroying mappings between control data streams and synthesis parameters is also presented.

Keywords: Mapping, Digital Musical Instrument, DMI, OpenSound Control, Network.

1 Introduction

Although designers of Digital Musical Instruments (DMI) are interested in creating useful, flexible, and inspiring interfaces and sounds, this process often depends on the vision and insight of a single individual. The McGill Digital Orchestra project instead brings together research-creators and researchers in performance, composition and music technology to work collaboratively in creating tools for live performance with digital technology [1]. A large part of this research focuses on developing new musical interfaces.[2]

[1] This paper is a revised and substantially expanded version of a preliminary report on this project presented at ICMC 2007[16].

[2] The McGill Digital Orchestra is a research/creation project supported by the *Appui à la recherche-création* program of the *Fonds de recherche sur la société et la culture* (FQRSC) of the Quebec government, and will culminate with concert performances of new works during the 2008 MusiMars/MusiMarch Festival in Montréal.

In the process of creating instruments for this project, we have found ourselves faced with the unique challenge of mapping new instruments in collaboration with experienced performers, as well as with composers tasked with writing pieces for these instruments. Because this ambitious project has taken on these three main challenges of the digital performance medium simultaneously, we have found ourselves in need of tools to help optimize the process. Specifically, mapping the various streams of controller output to the input parameters of synthesis engines has presented us with situations where both ease of use and flexibility were both of the utmost importance. We needed to be able to modify connections between data streams during precious engineer-composer-performer meeting time, while minimizing wasted minutes "reprogramming" our signal processing routines. Although arguably both powerful and intuitive, even graphical environments like Cycling 74's Max/MSP did not seem appropriate for these purposes, because non-programmers who had limited familiarity with such tools were expected to help in experimentation and design.

In consideration of several ongoing projects, including GDIF [13], Jamoma [18], Integra [2], and OpenSound Control (OSC) [24], we have created a "plug and play" network-based protocol for designing and using digital musical instruments. Controllers and synthesizers are able to announce their presence and make their input and output parameters available for arbitrary connections. Any controller is able to connect to any synthesizer "on the fly," while performing data scaling, clipping, and other operations.

In the course of developing an adequate working environment for this project, we have made developments in three main areas: the design of a network architecture which lends itself to a distributed "orchestral neighbourhood", in which controllers and synthesizers can interface with each other over a UDP/IP bus by means of an OSC-controlled arbitrator; the creation of a "toolbox" containing many useful functions which we found ourselves using repeatedly, coded as Max/MSP abstractions; and lastly a graphical mapping tool with which gestural data streams can be dynamically connected and modified.

We have tried to create a GUI that is intuitive and transparent: relationships between parameters are visible at a glance, and changing mappings and scaling requires only a few mouse clicks. We have used all of these tools in a real collaborative context, allowing us to present not only implementations, but also observations of their effect on our group dynamic and workflow. We have tried to create an interface that is useful not only for technical users, but also as a creative tool for composers and performers.

The remainder of this paper is organized as follows: section 2 provides background information and explains the motivations behind our approach to mapping for this project. Section 3 describes the design and implementation of the networked mapping system. Section 4 gives some details of how users might experience the mapping system through the provided graphical interface, and also provides information for developers on how to make compatible software. Section 5 briefly outlines some of the functions and tools we have created to help speed such development, available as a software package entitled the *Digital*

Orchestra Toolbox. Finally, sections 6 and 7 provide discussion, insights, and plans for future development.

2 Gesture Mapping

The digital instrument builder is faced with several tasks: after considering what sensors should be used, how the musician will likely interface with them, and what sounds the instrument will make, there is still the decision of which sensors should control which aspects of the sound. This task, known as *mapping*, is an integral part of the process of creating a new musical instrument [10].

2.1 Mapping Methods

Several past projects have developed tools for mapping between sound and control. However, "mapping" is a term with a wide scope, and these projects do not necessarily agree on methods or terminology. One way to categorize mapping methods is by whether the connections are known explicitly, or are the result of some process which builds implicit relationships [8].

An example of the latter is [14], in which it is seen that neural networks can be used to adapt mapping to a performer's gestures rather than the inverse. In contrast, toolboxes such as LoM [22] or MnM [6] are in the former category. They are intended to aid in developing strategies for using low-dimensional control spaces to control higher-dimensional timbral spaces through the use of several interpolation techniques. [21] also created a tool box of mapping functions for PureData (Pd) [19]. This work is similar to the set of Max/MSP abstractions that we present in Section 5, and, as we'll see in the next section, can be a useful resource for performing signal conditioning in the context of our system.

In the current work, we focus on easing the design of mapping by representing the individual connections between parameters in a very direct and explicit way, rather than worrying about signal conditioning or transformation strategies. Additionally, we feel that the choice of algorithm, or even programming language, used to communicate with physical devices and perform signal processing should have limited impact on inter-device interaction if they are connected on the same network. Thus, while tools such as LoM and MnM are useful for Max/MSP programmers, or the Pd mapping toolbox can be a useful resource for PureData programmers, we decided that approaching the problem of interoperation from a networking and protocol point of view would yield greater long-term results, and provide useful functionality for development of mapping techniques in a collaborative environment, while not necessarily precluding the use of other mapping approaches within its framework.

2.2 The Semantic Layer

An important result of previous discussions on mapping has been the acknowledgement of the need for a multi-layered topology. Specifically, Hunt and Wanderley [9] suggested the need for 3 layers of mapping, in which the first and last

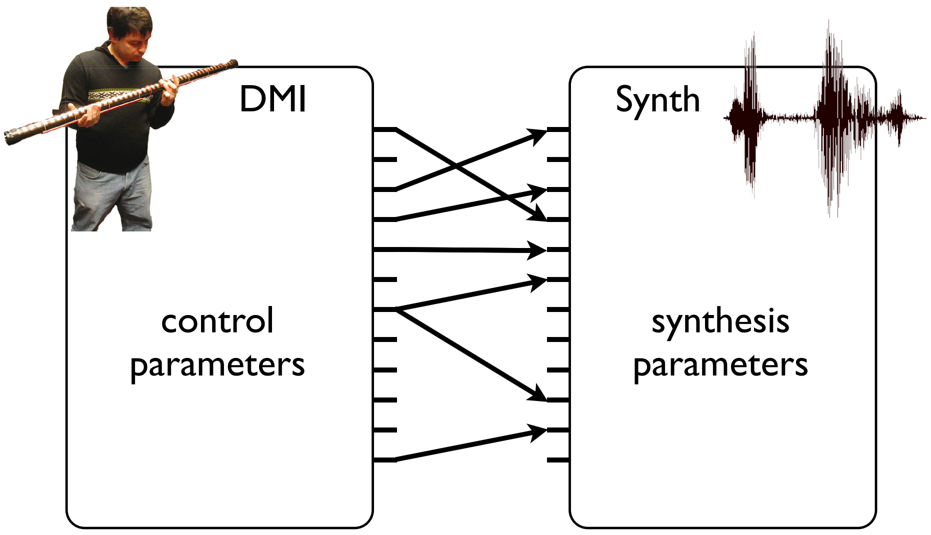


Fig. 1. An example single-layer mapping. One-to-many and many-to-one mappings are defined explicitly.

layers are device-specific mappings between technical control parameters and gestures (in the case of the first) or aesthetically meaningful "sound parameters", such as *brightness* or *position* (in the case of the third). This leaves the middle layer for mapping between parameter names that carry proper gesture and sound semantics. We shall refer to this layer as the "semantic layer", as described in Figure 2.

The tools presented here adhere to this idea. However, since the first and last mapping layers are device-specific, the mapping between technical and semantic parameters (layers 1 and 3) are considered to be part of the controller and synthesizer interfaces. Using an appropriate OSC addressing namespace, controllers present all available parameters (gestural and technical) to the mapping tool. The tool is used to create and modify the semantic layer, with the option of using technical parameters if needed.

As a simple example, the T-Stick interface [15] presents the controller's accelerometer data for mapping, but also offers an event-based "jabbing" gesture which is extracted from the accelerometers. The former is an example of layer 1 data which can be mapped directly to a synthesizer parameter. The latter is a gestural parameter presented by layer 2, which can be mapped, for example, to a sound envelope trigger. The mapping between layers 1 and 2 for the "jabbing" gesture, (what we call *gesture extraction*), occurs in the T-Stick's interface patch (see Figure 3).

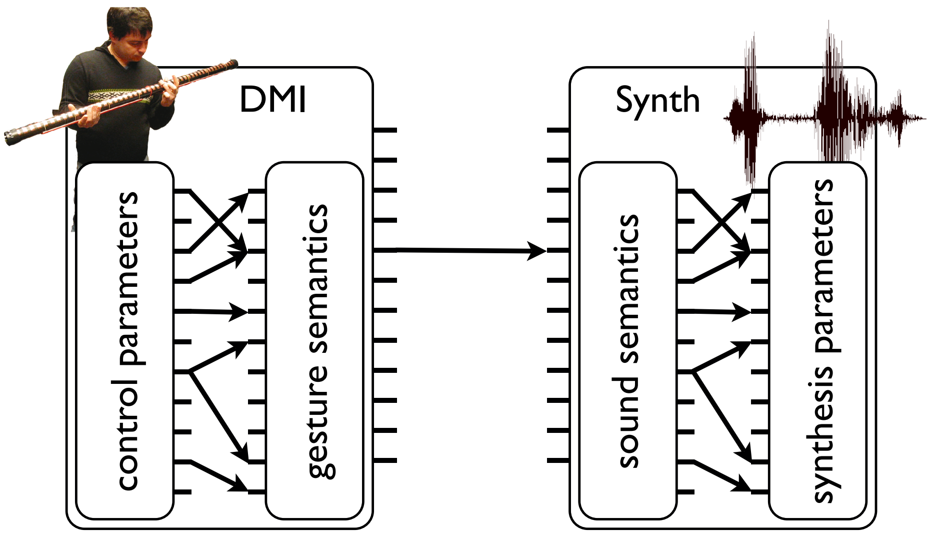


Fig. 2. A diagram of the 3-layer framework used for Digital Orchestra development, adapted from [9]. Note that the simple "one-to-one" connection shown in the center semantic mapping layer may in fact describe a much more complex relationship between technical parameters.

We have also used this system in another project[3] for mapping gesture control to sound spatialization parameters [17]. In this case a technical mapping layer exposes abstract spatialization parameters (such as sound source trajectories) to the semantic layer, rather than synthesis parameters.

2.3 Connection Processing

Gestural data and sound parameters will necessarily carry different units of measurement. On the gestural side, we have tried, whenever possible, to use units related to physical measurements: distance in meters, angles in degrees. In sound synthesis, units can sometimes be more arbitrary, but some standard ones such as Hertz and MIDI note number are obvious. In any case, data ranges will differ significantly between controller outputs and synthesis inputs. The mapping tool attempts to handle this by providing several features for scaling and clipping data streams.

One useful data processing tool that is available is a filter system for performing integration and differentiation. We have often found during sessions that a particular gesture might be more interesting if we could map its energy or its rate of change instead of the value directly [7]. Currently the data processing is

[3] Compositional Applications of Auditory Scene Synthesis in Concert Spaces via Gestural Control is a project supported by the NSERC/Canada Council for the Arts New Media Initiative.

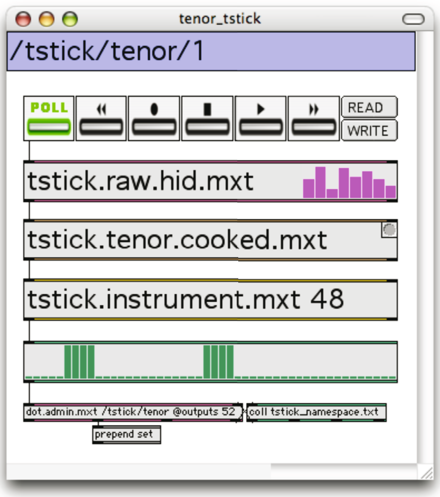


Fig. 3. A screenshot of the Max/MSP patch used for the T-Stick DMI, showing two layers of control data abstraction. The "cooked" sub-patch contains smoothing routines for sensor data, while the "instrument" sub-patch computes instrument-related gesture information such as "jabbing".

limited to first-order FIR and IIR filtering operations, and anything more complex must be added as needed to the "gesture" mapping layer and included in the mappable namespace.

2.4 Divergent and Convergent Mapping

It has been found in previous research that for expert interaction, complex mappings are more satisfying than simple mappings. In other words, connecting a single sensor or gestural parameter to a single sound parameter will result in a less interesting feel for the performer [10, 20].

Of course, since our goal is to use abstracted gesture-level parameters in mapping as much as possible, simple mappings in the semantic layer are in fact already complex and multi-dimensional [11]. Still, we found it would be useful to be able to create one-to-many mappings, and so the mapping tool we present here supports this. Each connection may have different scaling or clipping applied.

We also considered the use of allowing the tool to create many-to-one mappings. The implication is that there must be some *combining* function which is able to arbitrate between the various inputs. Should they be summed, or perhaps multiplied, or should some sort of comparison be made between each of the inputs?

A combining function implies some relationship between gestural parameters; in some cases, the combination of gestural data may itself imply the extraction of a distinct gesture, and should be calculated on the first mapping layer and presented to the mapping tool as a single parameter. In other cases the

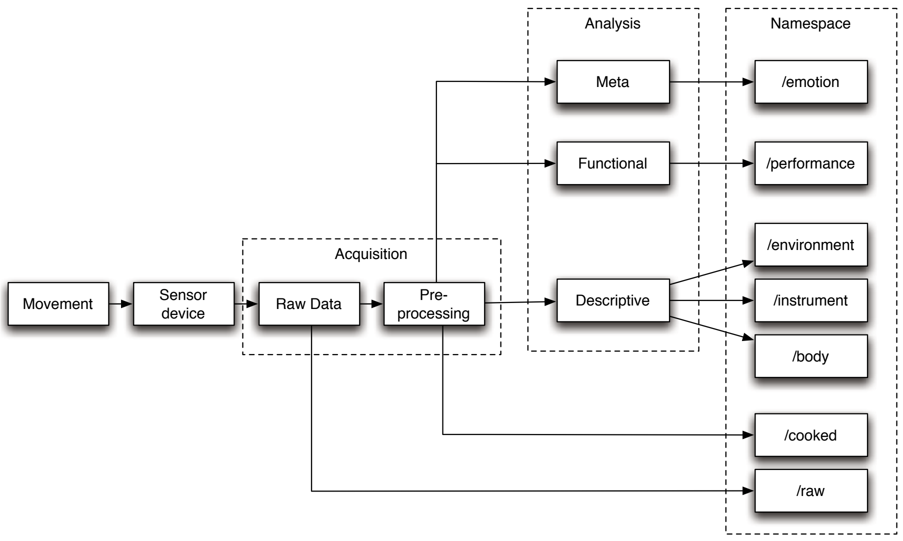


Fig. 4. A diagram from [12] showing the namespace hierarchy proposed for GDIF and used for the mapping system

combination may imply a complex relationship between synthesis parameters that could be better coded as part of the abstracted synthesis layer. In yet other cases the picture is ambiguous, but the prospect of needing to create on-the-fly many-to-one mappings during a working session seems to be unlikely. We did not implement any methods for selecting combining functions, and for the moment we have left many-to-one mappings for future work.

2.5 Portability of Mapping Sets

An important goal of the GDIF project is to pursue portability of mapping sets between similar devices. Control devices and DMIs using OSC namespaces structured using the hierarchy proposed for GDIF (Figure 4) will likely share subsets of their namespaces, especially at the higher levels, which are focused on interaction or environment rather than specific interfaces. Likewise, synthesizers receiving mapped parameters also often have named parameters in common, provided they have been constructed according to GDIF guidelines. The motivations for and structure of this hierarchy is described in detail in [12].

We designed the loading functionality of our system to permit portability of mapping sets between different classes of device, as described in Section 4.4. An important result of this pursuit is that different controllers and synthesizers may be swapped without the bulk of the mapping being redefined. This provides another motivation for encouraging users of a mapping system to make use of high-level abstracted parameters whenever possible.

3 The Orchestral Network Neighbourhood

In our system, there are several entities on the network that must communicate with each other using a common language. These include controllers and synthesizers, as well as the software devices used for address translation and data processing, called *routers*, and finally the GUI used to create and destroy connections. This protocol must allow them to perform some basic administration tasks, including: announcing their presence on the network; deciding what name and port to use; and finally describing what messages they can send and receive.

The system can be thought of as a higher-level protocol running on top of an OSC layer. OSC was chosen to encapsulate message passing because it has several advantages in the domain of audio systems. It was designed to take care of several drawbacks typically associated with MIDI: it is transport-independent, meaning that it defines a sequence of bytes but makes the assumption that the transport layer will take care of accurately carrying these bytes over some transmission medium. This lends itself well to IP networks, but is equally valid over another transport, such as a simple serial transmission line for example. OSC can specify data in several formats such as floating point values, strings, or integers, instead of being restricted to a specific range as in MIDI. Data type is specified in the message header. It is clear that OSC can be a flexible and useful messaging system, but its main advantage for us is that it is already supported by a large number of audio software packages, (although some support it better than others.) This means that while we were able to efficiently design the system described here using Max/MSP, the protocol we describe can be supported by several other audio-oriented programming languages. This topic will be covered more completely in Section 4.7.

In any case, while OSC can be a powerful tool, it suffers the disadvantage in comparison to MIDI in that it dictates nothing about what lower-level transport protocols and ports to use, nor what kinds of messages should be exchanged. We needed to devise a common set of OSC messages to allow the use of a standard interface to control all devices in question. The approach we have taken—that of translating arbitrary messages from a controller into inputs for a synthesizer—was chosen because we did not wish to impose a particular restriction on the device namespaces themselves: we do not assume to be able to enumerate a set of control messages as was done for General MIDI (GM). GM was designed primarily for keyboard controllers, and this is apparent in its semantics, which has been found limiting when exploring the use of alternative controllers for electronic sound. Instead, we propose a set of messages for *discovering* and *describing* controller outputs and synthesizers inputs, as well as messages for describing the connections and signal conditioning that might occur between them.

3.1 Topology and Protocol

Because OSC addressing is designed to uniquely identify any particular value, it is possible to broadcast messages on a common bus and have them be correctly targeted to the intended recipient. This makes it mostly trivial to switch between

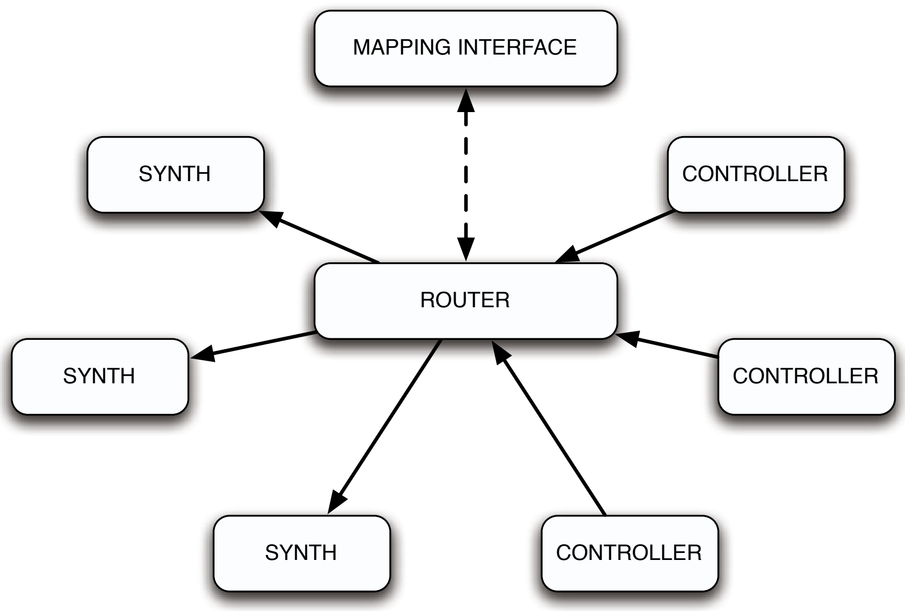


Fig. 5. A centralized topology in which all traffic is routed through a central router service

various network topologies. While a common bus is necessary for locating devices, it is not necessary nor is it optimal to have gestural data streams sharing the same bus.

We have decided to use a multicast UDP/IP port for administrative tasks such as device announcement and resource allocation. This common port is needed to resolve conflicting device identifiers and to allow new devices to negotiate for a unique private port on which to receive messages. We shall refer to it as the "admin bus".

For routing mapped data streams, several topologies can be used. Though it simplifies programming, sharing a bus for high-traffic gestural streams wastes communication as well as processing resources. Messages must be received and addresses must be parsed before being rejected. If several devices are present on the network, a high percentage of traffic may be rejected, making a common bus inefficient. In our system, each device reserves a UDP/IP port for receiving data streams. Thus the OSC traffic is quickly routed and filtered on the transport layer and address parsing is only necessary for properly targeted messages.

Another factor affecting the network topology is the role of the router in mapping. In a previous revision of our system, controllers sent their data streams to a router which performed address mapping and scaling before re-transmitting the transformed messages to a synthesizer. This implies a centralized topology as seen in Figure 5. However, with the protocols described in this section, it is perfectly feasible to have multiple router instances on the network. This can help

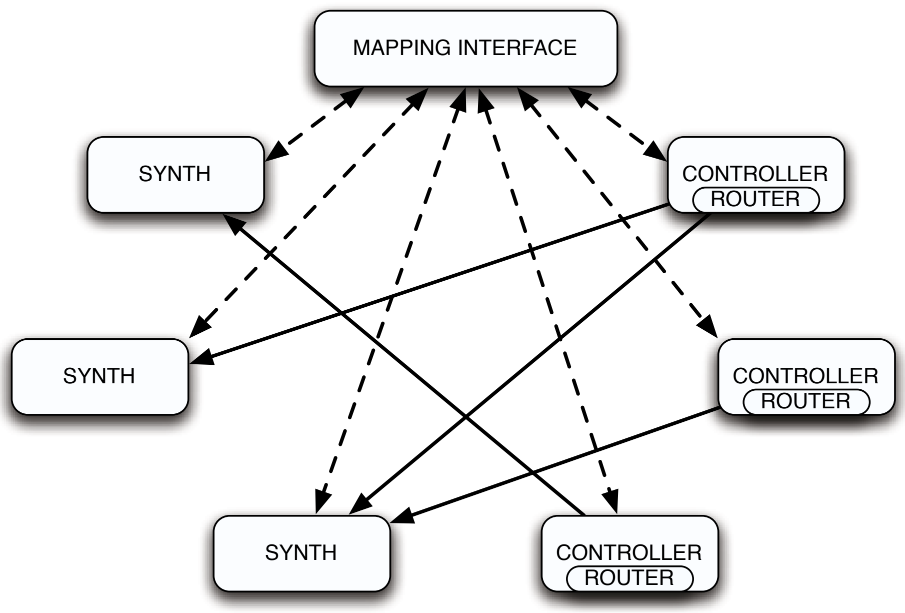


Fig. 6. Equivalently, a router can be embedded in each controller to create a true peer-to-peer network

reduce traffic loads and distribute processing. Extending this idea to the extreme, we embedded the routers inside each controller, in order to create a truly peer-to-peer topology, as described by Figure 6. The signal conditioning and message transformation then takes place on the controller itself, and messages are sent through the network already formatted for the target synthesizer. However, for clarity we consider the embedded router devices to be distinct entities on the network.

3.2 Name and Port Allocation

When an entity first appears on the network, it must choose a port to be used for listening to incoming data streams. It must also give itself a unique name by which it can be addressed. A simple solution would be to assign each device a static name and port. However, we are not interested in maintaining a public database of "claimed" ports, and, (being a digital "orchestra"), we expect multiple instances of a particular device to be available for use.

In an attempt to be more dynamic and decentralized, we have developed a collision handling algorithm for port and name allocation: when a new entity announces itself, it posts to the admin bus a message stating which port it tentatively intends to use. If this port is reserved or is also being asked for by another device, a random number is added and the device tries again. If multiple devices are attempting to reserve the same numbers, several iterations may occur, but

```
**Input**: $p_t$ (tentative port number)
**Output**: $p_f$ (final port number)
$T = 0$;
$c = -1$;
announce($p_t$);
**while** $T < 2000$ **do**
    update time $T$;
    update collision count $c$;
    **if** $T > 500$ *and* $c > 0$ **then**
        $p_t = p_t$ + random(0..$c$);
        $T = 0$;
        $c = -1$;
        announce($p_t$);
    **end**
**end**
$p_f = p_t$;
...
**repeat**
    **if** *collision* **then** announce($p_f$);
**until** *forever* ;
```

Fig. 7. Port allocation scheme, also used for device identifier ordinals

eventually each device ends with a unique port number to use. (Strictly speaking, this is only necessary for devices hosted on the same computer, though we currently run the collision algorithm on the shared admin bus over the network.) The same algorithm is used for determining a device name composed of the device class and a unique ordinal. This unique name is prepended to all messages from that device. Some pseudo-code of this algorithm can be found in Figure 7.

3.3 Discovery

Device discovery is an important step toward a truly “plug and play” environment. Previously, a method has been proposed for device discovery making use of the ZeroConf protocol, which is a decentralized network configuration and announcement protocol available in all major operating systems. A Max/MSP implementation of the idea, called *OSCBonjour*, has been created by Rémy Müller, which we have explored[4]. While the idea is promising, it currently only handles device discovery, leaving us still to deal with port and name allocation for the devices. We decided that, since we are already using a common multicast UDP bus for the allocation scheme, it would be sufficient and more consistent to use it also for device discovery. A pure OSC solution is adequate for our purposes, but this does not preclude the possibility of using *OSCBonjour* in the future, perhaps in parallel with our current scheme.

When a device appears on the network, and after it successfully receives a unique name and port number, it queries the network for other compatible devices by submitting a simple request on the multicast admin bus:

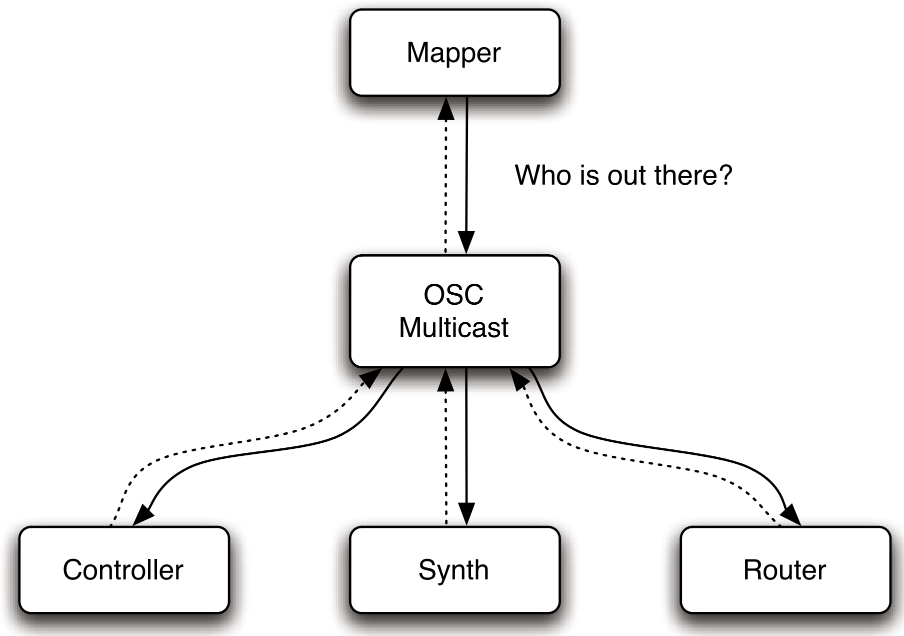


Fig. 8. The mapper GUI requests that devices on the network identify themselves

```
/device/who
```

All compatible devices, including the device just launched, respond to this message with a simple message stating their unique name, device class, I/O, IP address and port:

```
/device/registered /tstick/1 @inputs 1 @outputs 52 @class /tstick
   @IP 192.168.0.3 @port 8001
/device/registered /granul8/1 @inputs 80 @outputs 0 @class
   /granul8 @IP 192.168.0.4 @port 8000
```

3.4 Making Connections

Each device records the names, IP addresses and UDP ports of the other instances on the network. Mapping interfaces (Section 4) also listen on the admin bus and display the available devices as sources (controllers), destinations (synths), or both. In this way, a user can refer to a device by name rather than being required to know the address and port of a particular device. To create a direct network connection between two devices, a message must be sent on the admin bus specifying the devices to connect:

```
/link /tstick/1 /granul8/1
```

The source device dynamically creates a router data structure for each linked destination device. It then responds on the admin bus to acknowledge that it has successfully initialized the connection:

```
/linked /tstick/1 /granul8/1
```

Similarly, devices can be disconnected,

```
/unlink /tstick/1 /granul8/1
```

resulting in the destruction of the corresponding router in the source device, and a response on the admin bus:

```
/unlinked /tstick/1 /granul8/1
```

Once two devices have been connected with the /link message, individual OSC datastreams can be connected to their desired destination:

```
/connect /tstick/1/raw/pressure/1 /granul8/1/gain
/connect /tstick/1/raw/pressure/1 /granul8/1/gain @scaling
   expression @expression x*10 @clipping minimum 0
```

The appropriate router instance records the mapping connection, and sets up address translation, scaling, and clipping. Once complete, the response is sent on the admin bus:

```
/connected /tstick/1/raw/pressure/1 /granul8/1/gain
/properties /tstick/1/raw/pressure/1 /granul8/1/gain @scaling
   expression
/properties /tstick/1/raw/pressure/1 /granul8/1/gain @expression
   x*10
/properties /tstick/1/raw/pressure/1 /granul8/1/gain @clipping
   minimum 0
```

Note the optional connection properties (scaling, clipping) which can be specified as part of the /connect message. If no properties are given, the connection will be created using the defined default properties. (At this time, no scaling or clipping is performed by default.) It is notable that scaling may not be appropriate for certain types of OSC message arguments, such as character strings, and applied to these messages it will cause undefined behaviour. In the case of gain-related destination parameters, use of the "@clipping both" property at connection creation might be advisable to avoid damage to ears and audio equipment. However, this is not done automatically.

Also notable is the two-stage process described for connecting parameters: the first to define a network connection, and the second to define the connection between parameter addresses. Although it is useful to separate device-level and address-level connections, the "admin" Max/MSP abstraction described in Section 5 automatically creates necessary device-level links if a simple "/connect" message is sent.

Similar to the "/unlink" message at the device-connection level, individual addresses can also be disconnected:

```
/disconnect /tstick/1/raw/pressure/1 /granul8/1/gain
/disconnected /tstick/1/raw/pressure/1 /granul8/1/gain
```

To query the properties of a connection,

```
/connection/properties/get /tstick/1/raw/pressure/1
   /granul8/1/gain
```

To modify the properties,

```
/connection/modify /tstick/1/raw/pressure/1 /granul8/1/gain
   <desired properties, @scaling...>
```

Both the "/connection/properties/get" and the "/connection/modify" messages elicit a response specifying the current mapping properties:

```
/connection/properties /tstick/1/raw/pressure/1 /granul8/1/gain
   <@scaling...>
```

3.5 Namespace Queries

Lastly, each orchestra member must be able to tell others what it can do. In other words, it must be able to say what messages it can receive and what messages it can send. Wright et al. [24] proposed the use of the `/namespace` message for causing a device to enumerate its available namespace. We have implemented this for each transmitter and receiver on the network. In addition to listing the namespace itself, each available parameter optionally can include information about data type, data range and units used. These come into play when the mapper is to set up automatic scaling between data streams, as described below.

In order to make this metadata optional, we have used a tagged argument scheme, similar to the syntax used in Jitter for object "attributes." In the example below, the mapper interface communicates with a controller named "/tstick/1" and a granular synthesizer named "/granul8/1". (These ordinals were previously established by the allocation scheme, so as not to be confused with other devices of the same type.) The exchange is described in the sequence diagram seen in Figure 9.

4 The Mapping Interface

A graphical interface has been developed to aid in mapping tasks. It forms a separate program from the other devices, but transmits and receives OSC messages on the same multicast admin bus. In addition to allowing the negotiation of mapping connections from another location on the network, this approach has allowed us to simultaneously use multiple mapping interfaces on the network, with multiple users collaborating to map the parameters of a common set of controllers and synths. The mapping interface has several main functions.

4.1 Browsing the Network Neighbourhood

The first use of the mapping interface is naturally choosing the devices that you wish to work with, both for gesture and for sound synthesis or processing. The interface queries devices on the network, to discover mappable inputs and outputs, and displays this information in an easily understandable format. New devices appearing on the network are automatically added to the display as seen in figure 12.

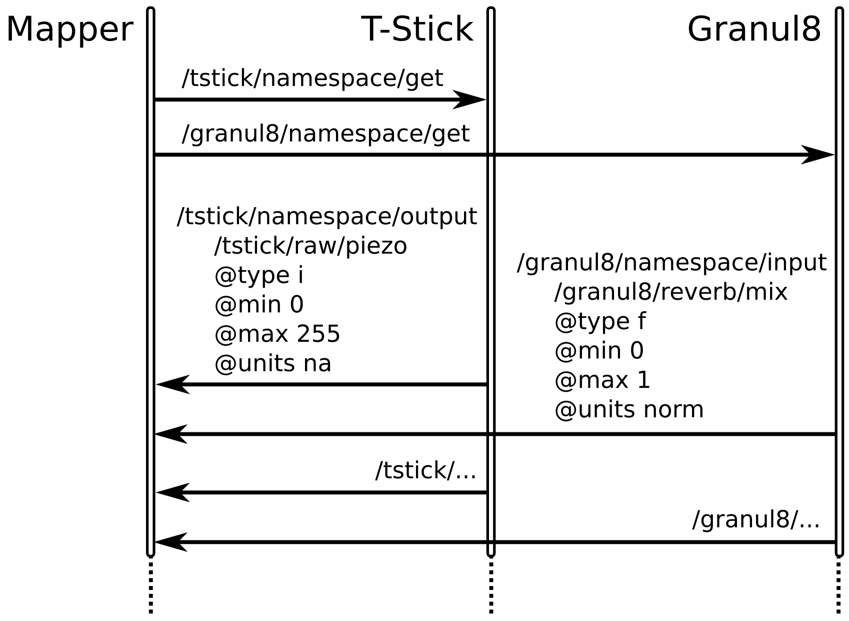


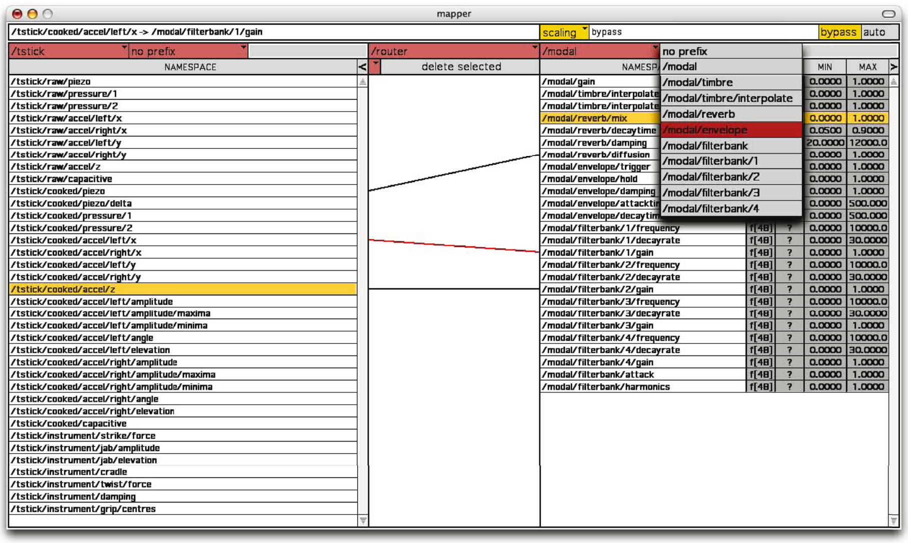
Fig. 9. On receipt of a namespace request, devices return a message for each input or output parameter they support, with information about data range and units

4.2 Browsing and Searching Namespaces

When devices are selected in the mapping interface, a message is sent on the admin bus requesting them to report their full OSC address-space. The interface displays the parameters by OSC address, which is especially informative to the user when strong semantics are used in the namespace. In addition, some other information about each parameter is requested, including whether it is an *input* or an *output*, the data type (*i*, *f*, *s*, etc.), the unit type associated with the parameter (Hz, cm, etc.), and the minimum and maximum possible values. OSC address patterns for controller outputs are displayed on the left side of the mapping interface, and synthesizer inputs are displayed on the right.

In order to manage the browsing and mapping of very large or deep namespaces, the mapping interface also allows for filtering and searching using pattern-matching. Two stages of namespace filtering are available, which may be used together. One stage allows filtering by OSC address-pattern prefix, chosen from an automatically-populated drop-down menu, so that the user may view the set of parameters which are children of a particular node in the address hierarchy. The other stage allows filtering by regular expression, so that only parameters matching a particular pattern are displayed.

On occasions where the namespace can change, such as for entities that have a configurable interface, addition or removal of addresses is announced on the



/tstick/cooked/accel/left/x -> /modal/filterbank/1/gain

scaling: bypass

NAMESPACE
/tstick/raw/piezo
/tstick/raw/pressure/1
/tstick/raw/pressure/2
/tstick/raw/accel/left/x
/tstick/raw/accel/right/x
/tstick/raw/accel/left/y
/tstick/raw/accel/right/y
/tstick/raw/accel/z
/tstick/raw/capacitive
/tstick/cooked/piezo
/tstick/cooked/piezo/delta
/tstick/cooked/pressure/1
/tstick/cooked/pressure/2
/tstick/cooked/accel/left/x
/tstick/cooked/accel/right/x
/tstick/cooked/accel/left/y
/tstick/cooked/accel/right/y
/tstick/cooked/accel/z
/tstick/cooked/accel/left/amplitude
/tstick/cooked/accel/left/amplitude/maxima
/tstick/cooked/accel/left/amplitude/minima
/tstick/cooked/accel/left/angle
/tstick/cooked/accel/left/elevation
/tstick/cooked/accel/right/amplitude
/tstick/cooked/accel/right/amplitude/maxima
/tstick/cooked/accel/right/amplitude/minima
/tstick/cooked/accel/right/angle
/tstick/cooked/accel/right/elevation
/tstick/cooked/capacitive
/tstick/instrument/strike/force
/tstick/instrument/jab/amplitude
/tstick/instrument/jab/elevation
/tstick/instrument/cradle
/tstick/instrument/twist/force
/tstick/instrument/damping
/tstick/instrument/grip/centres

NAMESP			MIN	MAX
/modal/gain			0.0000	1.0000
/modal/timbre/interpolate			0.0000	1.0000
/modal/timbre/interpolate			0.0000	1.0000
/modal/reverb/mix			0.0000	1.0000
/modal/reverb/decaytime			0.0500	0.9000
/modal/reverb/damping			20.0000	12000.0
/modal/reverb/diffusion			0.0000	1.0000
/modal/envelope/trigger			0.0000	1.0000
/modal/envelope/hold			0.0000	1.0000
/modal/envelope/damping			0.0000	1.0000
/modal/envelope/attacktir			0.0000	500.000
/modal/envelope/decaytim			0.0000	500.000
/modal/filterbank/1/frequency	f[48]	?	0.0000	10000.0
/modal/filterbank/1/decayrate	f[48]	?	0.0000	30.0000
/modal/filterbank/1/gain	f[48]	?	0.0000	1.0000
/modal/filterbank/2/frequency	f[48]	?	0.0000	10000.0
/modal/filterbank/2/decayrate	f[48]	?	0.0000	30.0000
/modal/filterbank/2/gain	f[48]	?	0.0000	1.0000
/modal/filterbank/3/frequency	f[48]	?	0.0000	10000.0
/modal/filterbank/3/decayrate	f[48]	?	0.0000	30.0000
/modal/filterbank/3/gain	f[48]	?	0.0000	1.0000
/modal/filterbank/4/frequency	f[48]	?	0.0000	10000.0
/modal/filterbank/4/decayrate	f[48]	?	0.0000	30.0000
/modal/filterbank/4/gain	f[48]	?	0.0000	1.0000
/modal/filterbank/attack	f[48]	?	0.0000	1.0000
/modal/filterbank/harmonics	f[48]	?	0.0000	1.0000

Fig. 10. The mapping graphical user interface can be used to explore the available namespace, make connections, and specify scaling and other data processing

multicast bus so that routers can destroy any connections appropriately, and mappers can add or remove entries.

4.3 Negotiating Mapping Connections and Properties

The mapping interface is essentially memoryless, merely reflecting the state of devices and connections present on the network. Simple methods are provided for creating and destroying mapping connections, and for editing the properties of existing connections (i.e.: scaling, clipping). Connections are created by selecting displayed namespaces on each side of the interface (inputs and outputs), and lines are drawn connecting mapped parameters. Mapping connections can be selected (in which case they are highlighted) for editing or deletion. By selecting multiple namespaces or connections, many mappings can be created, edited, or destroyed together.

When a connection is made, by default the router does not perform any operation on the data ("bypass"). A button is provided to instruct the appropriate router to perform basic linear scaling between the provided data ranges. Another button instructs the router to commence calibration of the scaling using the detected minima and maxima of the input data stream. The user can also manually type "linear" in an expression textbox with arguments defining a specific input and output ranges. Options are also available for defining a clipping range.

The expression box is quite versatile. For more advanced users, it accepts any string which can be understood by Max/MSP's "expr" object, and evaluates the mapped data according to the entered expression. Additionally, expressions can refer to the current value, or a single previous input or output sample. This

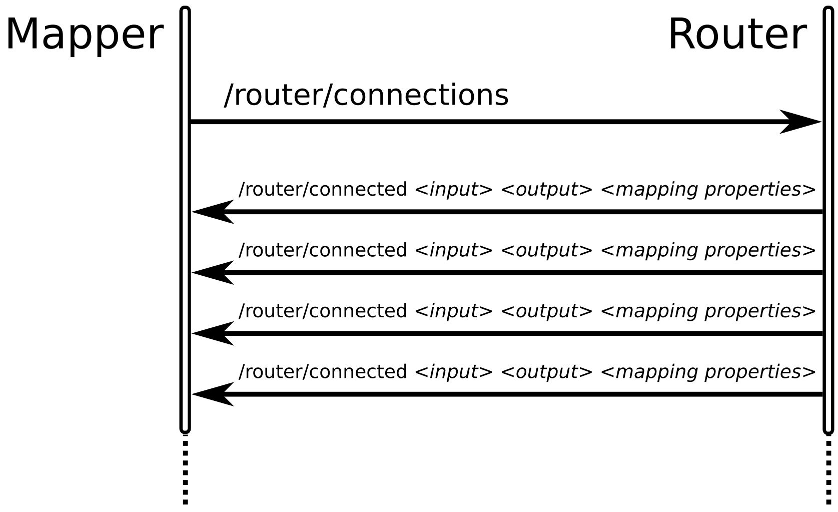

Mapper
Router
/router/connections
/router/connected <input> <output> <mapping properties>
/router/connected <input> <output> <mapping properties>
/router/connected <input> <output> <mapping properties>
/router/connected <input> <output> <mapping properties>

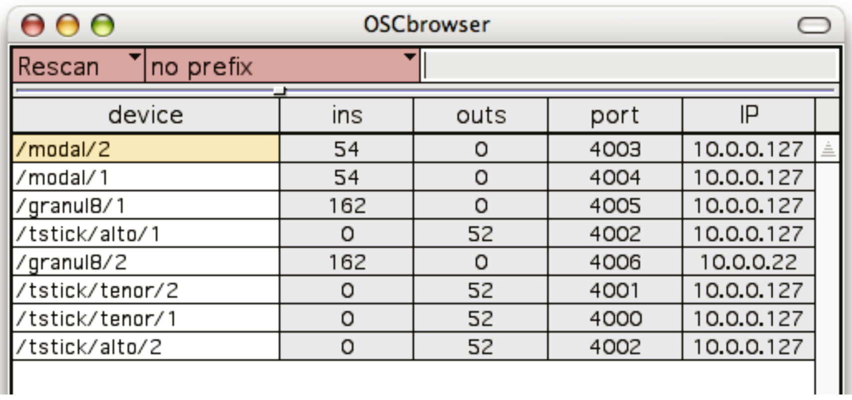
Fig. 11. A router device can report its current set of connections. The mapper GUI requests it when the router is first selected.

control may be used to specify single-order filters, or non-linear, logarithmic mappings, for example. There is currently no support for table-based transfer functions.

4.4 Saving and Loading Mapping Sets

The mapping interface also provides buttons for saving defined mapping sets locally as a text file. This file stores the properties of each connection between the viewed devices, along with the unique names of the devices involved. This information is formatted according to Max/MSP's `coll` object storage, although we have considered that it may be more useful in some XML-based standard as a language-agnostic data format to be easily imported into other implementations. We intend to define such a format in collaboration with the GDIF project at some point in the future.

When using the mapping interface to load a stored mapping file, devices must first be chosen between which the connections will be made, since it is possible that instance numbers and thus unique device names will differ between sessions. The loading function strips each defined connection of its original device identifier and replaces it with the name of the currently selected device. An advantage of loading mapping sets in this way is the possibility of loading the mapping with a different device class, as discussed in Section 2.5. Connections involving parts of the namespace shared by the original and replacement devices will be created normally, otherwise they will simply be discarded.



device	ins	outs	port	IP
/modal/2	54	0	4003	10.0.0.127
/modal/1	54	0	4004	10.0.0.127
/granul8/1	162	0	4005	10.0.0.127
/tstick/alto/1	0	52	4002	10.0.0.127
/granul8/2	162	0	4006	10.0.0.22
/tstick/tenor/2	0	52	4001	10.0.0.127
/tstick/tenor/1	0	52	4000	10.0.0.127
/tstick/alto/2	0	52	4002	10.0.0.127

Fig. 12. The device browser used in the mapping GUI can also be launched as a stand-alone application. It displays all compatible devices along with their inputs, outputs, ports and IP addresses.

4.5 Message Examples

Using the mapping interface, the user selects the controller namespace `/tstick/instrument/damping`, and the synth namespace `/granul8/1/grain/1/filter/frequency`:

Mapper `/connect <controller parameter> <synth parameter>`
Example `/connect /tstick/1/instrument/damping /granul8/1/grain/1/filter/frequency`

The T-Stick receives the message and creates mapping with default parameters:

Device `/connected <controller parameter> <synth parameter> <properties>`
Example `/connected /tstick/1/instrument/damping /granul8/1/grain/1/filter/frequency @scaling bypass @clipping none`

The user begins calibration:

Mapper `/connection/modify <controller parameter> <synth parameter> <properties>`
Example `/connection/modify /tstick/1/instrument/damping /granul8/1/grain/1/filter/frequency @scaling calibrate 20 1000`
T-Stick `/connection/properties <controller parameter> <synth parameter> <properties>`
Example `/connection/properties /tstick/1/instrument/damping /granul8/1/grain/1/filter/frequency @scaling calibrate 20 1000`

Example `/connection/properties /tstick/1/instrument/damping /granul8/1/grain/1/filter/frequency @scaling calibrate 20 1000 @expression (x-32)*0.00345+100`

The user ends calibration:

Mapper `/connection/modify <controller parameter> <synth parameter> @scaling expression`
Example `/connection/modify /tstick/1/instrument/damping /granul8/1/grain/1/filter/frequency @scaling expression`

The user deletes the mapping:

Mapper `/disconnect <controller parameter> <synth parameter>`
Example `/disconnect /tstick/1/instrument/damping /granul8/1/grain/1/filter/frequency`
T-Stick `/disconnected <controller parameter> <synth parameter>`
Example `/disconnected /tstick/1/instrument/damping /granul8/1/grain/1/filter/frequency`

4.6 Adapting Existing Max/MSP Patches for Compatibility

The Max/MSP implementation of the tools presented here has been carefully designed for easily adapting existing device patches (both controllers and synthesizers) for compatibility with the system. If the pre-existing patch already uses OSC for input and/or output, there is in fact very little left to do: the messages simply need to be connected to a copy of the `dot.admin` abstraction described in section 5. The last step is simply to create a text file containing a list of OSC parameters the patch can send and/or receive, and load it into a `coll` object connected to `dot.admin`. More detailed documentation accompanies the distribution when downloaded, but essentially after this the adaptation is functional. Optionally, the properties can be defined for each parameter, specifying its data type, associated unit, and range; these properties, while not required, make scaling and calibration easier when in use.

We recommend that anyone adapting an existing patch—or creating a new patch—for use in this context use strong semantics in their choice of parameter names and avoid obscure abbreviations. This allows users to immediately understand the functions of each parameter without referring to external documentation. While we also advocate the use of hierarchical parameter naming as proposed for GDIF, this is not required;the system itself does not depend on a particular approach.

4.7 Other Implementations

While the main body of work for this project has been developed using Max/MSP, our choice of using OpenSound Control, a well-defined communication protocol

```
#include <mapper.h>

mapper_admin_init();
my_admin = mapper_admin_new("tester", MAPPER_DEVICE_SYNTH, 8000);
mapper_admin_input_add(my_admin, "/test/input","i"))
mapper_admin_input_add(my_admin, "/test/another_input","f"))

// Loop until port and identifier ordinal are allocated.
while (   !my_admin->port.locked
       || !my_admin->ordinal.locked )
{
  usleep(10000); // wait 10 ms
  mapper_admin_poll(my_admin);
}

for (;;)
{
  usleep(10000);
  mapper_admin_poll(my_admin);
}
```

Fig. 13. Framework for a synth-side C program using libmapper. This is the minimal code needed for a synth-side device to announce itself and communicate with other devices on the network.

with growing support, has allowed us to ensure that the system remains independent of specific software and hardware (provided it has IP networking capabilities). To demonstrate this point, and to encourage the use of our protocol, we are developing patches and libraries in several languages that make it easy to create compliant software interfaces. For instance, we have shown that a synthesizer written in PureData can be made to communicate with the system by adding a similar `dot.admin` object to a patch and filling in namespace details. Similarly, we have created a library in C that will enable a wide variety of C and C++ programs to easily support this platform. This has been tested using several synthesizers developed with the help of the Synthesis Toolkit in C++[5] and the LibLo OSC library[3]. An example of the use of this library is given in Figure 13.

5 The Digital Orchestra Toolbox

In the process of creating controller, synthesizer, and mapping patches, we have made an effort to modularize any commonly used subroutines. These have been organized, with help patches, into a toolbox that we find ourselves re-using quite often. Like the rest of the software presented in this paper, this toolbox is freely available on the Input Devices and Music Interaction Laboratory website [4]. It currently contains over 40 abstractions, some of which we will briefly describe.

[4] http://www.idmil.org

5.1 OSC and Mapping Helpers

dot.admin Handles communication on the admin bus. It is used by the synthesizer and controller patches to allow them to communicate with the mapping system. It uses instances of `dot.alloc` to negotiate for a unique port and identifier, and it responds to namespace requests. When required to do so, dot.admin dynamically creates instances of `dot.router` corresponding to each peer-to-peer device link on the network.

dot.alloc The abstracted algorithm used by `dot.admin` for allocating a unique port and device name. On its own, it may be useful for negotiating any unique resource on a shared bus.

dot.prependaddr Prepends the given symbol onto the first symbol of a list, without affecting the remaining list members. This is intended for constructing OSC addresses.

dot.autoexpr Given a destination maximum and minimum, `dot.autoexpr` will automatically adjust the linear scaling coefficients for a stream of incoming data. Calibration can be turned on and off. It can also handle arbitrary mathematical expressions (through the use of an internal `expr` object), and dynamically instantiates objects necessary for performing specified transformations, including first-order FIR and IIR filters.

dot.router The Max/MSP version of our “router” data-structure; performs namespace translation of mapped parameters, and scaling and clipping of data streams.

5.2 Gesture Extraction Helpers

dot.play/dot.record These objects can be used to record up to 254 incoming data channels into a `coll` object using delta timing, and later played back. It is useful for gesture capture and off-line mapping experimentation. The objects `dot.recordabsolute` and `dot.playabsolute` perform the same function with absolute time-stamping.

dot.extrema Automatically outputs local maxima and minima as peaks and troughs are detected in the incoming data. Helps in extraction of gestural events from sensor data.

dot.leakyintegrator A configurable integrator which leaks over time. The integration can either be linear, exponential, or use an arbitrary transfer function specified as a table.

dot.timedsmooth An averaging filter for asynchronous control data that makes use of DSP objects for reliably timed smoothing.

dot.transfer Performs table-based waveshaping on control data streams with customizable transfer function. (This is used in controller patches for signal processing, but not yet accessible through the mapping GUI.)

5.3 Case Example

Sally is a composer of electro-acoustics who has written a piece requiring soundfile triggers. Bob is a percussionist interested in exploring the use of ballistic body movements for electronic performance. Sally creates a Max/MSP patch using `dot.timedsmooth` and `dot.extrema` from the *Digital Orchestra Toolbox* to extract arm movements from 3-axis accelerometers held by Bob in each hand. She exposes the smoothed accelerometer data as well as the trigger velocity information through OSC.

Sally loads the mapping interface. She then loads her accelerometer patch as well as a sampler patch which has been configured for OSC messaging. After loading, these devices are listed in the mapping interface, where she selects them, making the accelerometer and extrema data visible on the left-hand side, and the sample triggers visible on the right-hand side. To begin, she guesses that it would be good to trigger sample 1 using the right-hand forwards movement, scaling the volume according to the movement's speed. She clicks on `/body/hand/right/forward_trigger`, selecting it, and then clicks on `/sample/1/play_at_volume`, connecting them. Since she had originally determined an estimated range of values for the accelerometer data, it automatically scales to the volume information, and the scaling coefficients are visible in the upper right-hand corner of the screen.

Bob tries the configuration for a few minutes, but decides there is not enough control, and it requires too much energy to achieve even modest volume on the sound. They decide to re-calibrate. Sally clicks on "calibrate", and Bob makes several triggering gestures with his right hand, until the volume range seems consistent. He makes some extreme gestures to maximize the range, so that he is able to achieve better control with a moderate amount of effort. Sally then toggles "calibrate" and saves the mapping. Bob plays for a while, and decides some small adjustments to the range are needed, so Sally manually changes the scaling coefficient instead of re-calibrating again.

Next they decide to map a low-pass filter, which is available through the sampler, to the motion of Bob's left hand. Sally chooses `/body/left/hand/accel/x` and then clicks on `/sample/1/filter/frequency`. Instantly the sound drops to a bass tone, much too low. Sally chooses clipping options from the drop-down menu and sets the minimum to 100 Hz, and the maximum to 5000 Hz. They re-calibrate the left-hand accelerometer range while triggering samples with the right hand. Bob begins to understand how to control the sound more accurately as they practice, and eventually they start looking at the score together.

6 Discussion

From their earliest use, the solutions we have developed have allowed us to streamline the process of mapping in collaboration with performers and composers. The ability to quickly experiment with a variety of mapping connections democratizes the mapping process, since it is easier to try everyone's ideas during a mapping session. Showing the performers that the connections are malleable allows them to contribute to the development of a comfortable gestural vocabulary for the instrument, rather than accepting the mappings provided. Composers are able to explore control of sounds that interest them without supervision or assistance of a technical member. Using common tools for the group means that the work of others is easily viewed and understood.

Controllers and synths that are still in development are also easily supported: as the supported parameter-space increases, the device simply presents more namespaces to the GUI.

Naturally this system does not solve all of the problems encountered in a collaborative effort of this type. The technical knowledge of the group members varies widely, and some technical knowledge of the individual controllers and synths is still necessary, not least because they are still in development and may not always respond predictably. As much as possible, however, we have made the connection, processing, and communication of data between devices easy to both comprehend and perform.

One area of frustration in our work has been dealing with devices (specifically commercial software synths) which communicate solely using MIDI. Since the norm in this case is to use MIDI control-change messages, many software environments allow flexible mapping between MIDI input values and their internal semantically labeled synth parameters. This means that although the synth parameters are easily understood from within a sequencing environment for adjustment or automation, external access to these parameters is provided only through an arbitrary set of MIDI control change identifiers. Our solution is to create a static set of MIDI mappings for our use, and provide a translation layer outside the environment to expose semantic parameters identical to those used internally. It is hoped that as users become familiar with see the advantages of semantic mapping, they will move away from a dependence on the traditional MIDI workflow.

In namespace design we have tried throughout to conform to the hierarchy proposed for GDIF [13], since we are also involved in its development, and this also raises some implementation questions. An important part of the GDIF hierarchy concerns representing gesture information in terms of the body of the performer, using the `/body` OSC prefix, and indeed several of our controllers already use this namespace. However, distinguishing performers using OSC address patterns proves much more complex when considering the various possible permutations of multiple performers and controllers.

7 Future Work

In addition to incremental improvements in function and usability, we have planned the addition of several new features:

Many-to-one mapping: As discussed above, we would like to implement the ability to negotiate many-to-one mapping relationships explicitly within the mapping interface, with simple GUI control over the desired combining function.

Vectors: Many OSC devices currently send or receive data in vectors or lists. The ability to split, combine, individually scale, and reorder vector elements will be added.

OSC pattern-matching: Pattern-matching and wild-card functionality is defined in the OSC specification [23] but generally has not been fully implemented in OSC systems. It is easy to imagine scenarios in which using wild-cards in mapped OSC address patterns would be a powerful addition to our system.

Data rates: Rather than sending controller information as quickly as possible, we would like to make the data rate a property of the mapping connection. A data stream might be used to control very slowly-evolving synthesis parameters, in which case very high data rates may be unnecessary and wasteful.

Remote collaboration: The implementations described above currently work over a local area network, however we would like to explore their use between remote locations communicating over the internet. In addition to collaborative mapping sessions between remote locations, this scenario could permit low-bandwidth communication of performance data for remote collaborative performance, in which control data is sent to instances of a software synthesizer at each location.

Acknowledgements

The authors would like to thank Alexander Refsum Jensenius for important discussion related to this effort, as well as the members of the Digital Orchestra Project group, especially Heather Hindman, Xenia Pestova, Chloé Dominguez, Fernando Rocha, D. Andrew Stewart, and Sean Ferguson. This project was 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] The McGill Digital Orchestra (2007), `http://www.music.mcgill.ca/musictech/DigitalOrchestra`
[2] Integra: A composition and performance environment for sharing live music technologies (2007), `http://integralive.org`
[3] liblo: Lightweight OSC implementation (2007), `http://liblo.sourceforge.net`
[4] OSCTools (2006), `http://sourceforge.net/projects/osctools`
[5] The Synthesis ToolKit in C++ (STK) (2007), `http://ccrma.stanford.edu/software/stk`
[6] Bevilacqua, F., Müller, R., Schnell, N.: Mnm: a max/msp mapping toolbox. In: Proceedings of the conference on New Interfaces for Musical Expression, Vancouver, Canada, pp. 85–88. National University of Singapore (2005)
[7] Hunt, A.: Radical User Interfaces for Real-time Musical Control. PhD thesis, University of York, UK (1999)
[8] Hunt, A., Kirk, R.: Mapping strategies for musical performance. In: Wanderley, M., Battier, M. (eds.) Trends in Gestural Control of Music, IRCAM - Centre Pompidou, Paris (2000)
[9] Hunt, A., Wanderley, M.M.: Mapping performance parameters to synthesis engines. Organised Sound 7(2), 97–108 (2002)

[10] Hunt, A., Wanderley, M., Paradis, M.: The importance of parameter mapping in electronic instrument design. In: Proceedings of the 2002 Conference on New Interfaces for Musical Expression, pp. 149–154 (2002)

[11] Hunt, A., Wanderley, M.M., Paradis, M.: The importance of parameter mapping in electronic instrument design. Journal of New Music Research 32(4), 429–440 (2003)

[12] Jensenius, A.R.: Action - Sound: Developing Methods and Tools to Study Music-related Body Movement. PhD thesis, University of Oslo, Norway (submitted, 2007)

[13] Kvifte, T., Jensenius, A.R.: Towards a coherent terminology and model of instrument description and design. In: Proceedings of the conference on New interfaces for musical expression, Paris, France, pp. 220–225. IRCAM – Centre Pompidou (2006)

[14] Lee, M., Wessel, D.: Connectionist models for real-time control of synthesis and compositional algorithms. In: Proceedings of the International Computer Music Conference, pp. 277–280 (1992)

[15] Malloch, J., Wanderley, M.M.: The T-Stick: From musical interface to musical instrument. In: Proceedings of the 2007 International Conference on New Interfaces for Musical Expression (NIME 2007), New York City, USA, pp. 66–69 (2007)

[16] Malloch, J., Sinclair, S., Wanderley, M.M.: From controller to sound: tools for collaborative development of digital musical instruments. In: Proceedings of the International Computer Music Conference, Copenhagen, Denmark, pp. 65–72 (2007)

[17] Marshall, M., Malloch, J., Wanderley, M.M.: A framework for gesture control of spatialization. In: Proceedings of the 2007 International Gesture Workshop, Lisbon, Portugal (2007)

[18] Place, T., Lossius, T.: Jamoma: A modular standard for structuring patches in max. In: Proceedings of the International Computer Music Conference, New Orleans, USA (2006)

[19] Puckette, M.: Pure Data: another integrated computer music environment. In: Proceedings, Second Intercollege Computer Music Concerts, Tachikawa, Japan, pp. 37–41 (1996)

[20] Rovan, J.B., Wanderley, M., Dubnov, S., Depalle, P.: Instrumental gestural mapping strategies as expressivity determinants in computer music performance. In: Proceedings of Kansei- The Technology of Emotion Workshop, Genova (1997)

[21] Steiner, H.-C., Henry, C.: Progress report on the mapping library for pd. In: Proceedings of the PureData Convention, Montreal, Canada (2007)

[22] Van Nort, D., Wanderley, M.M.: The LoM mapping toolbox for Max/MSP/Jitter. In: Proceedings of the International Computer Music Conference, New Orleans, USA (2006)

[23] Wright, M.: OpenSound Control specification (2002), `http://www.cnmat.berkeley.edu/OSSC/OSC-spec.html`

[24] Wright, M., Freed, A., Momeni, A.: OpenSound Control: State of the art. In: Proceedings of the Conference on New Interfaces for Musical Expression (2003)