Control: a Performance and Composition System for Robotic Instruments

Paul Mathews

February 4, 2014

1 Background

Using networks of computers for musical performance is a concept that has been developed since the late 1970s when The Hub [**bischoff_music_1978**] began performing, controlling parameters of each other's synthesis algorithms via a local network. Since then several groups have explored similar ideas

Control builds on the work undertaken by The Machine Orchestra [**vallis_building_2012**, **kapur_machine_2011**]

2 Overview

The system is divided into two parts, the *client* and the *server*. The server runs on a single computer, connected to the various devices and listens to incoming Open Sound Control (OSC) messages and translates them to the appropriate control for the target device.

The client runs on any number of computers networked to the server and serves to translate MIDI into OSC which it then relays to the server. This is optional, users can interface with the server directly through OSC (via UDP) if desired.

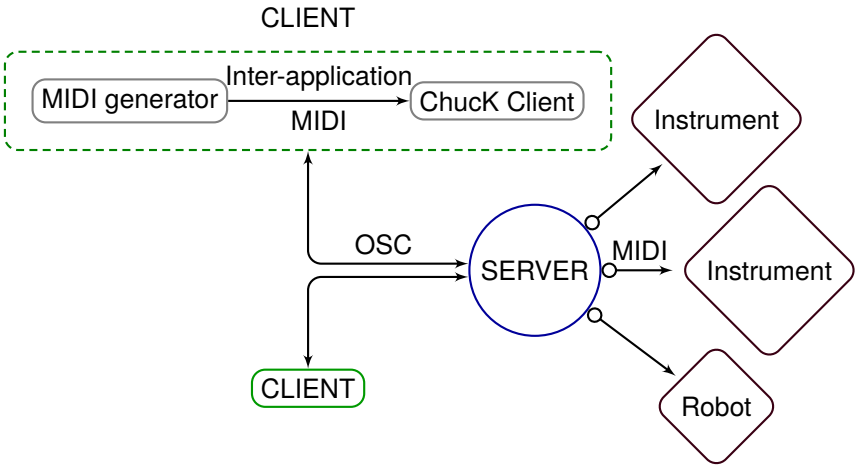

Figure 1: Overview of the system.

Figure 1 provides an overview of the system. The server interfaces directly with an arbitrary number of instruments, translating messages received from some number of clients.

2.1 Server

The server determines what instruments are available by reading files in the `Instruments` directory of the project folder; it assumes every file present represents an instrument currently connected and sets itself up to communicate to them based on the information in the file.

2.2 Client

Client machines communicate to the server using OSC over UDP. A translation layer is provided which accepts MIDI messages and turns them into OSC which it then sends to the server. Some form of inter-application MIDI routing is required for this to succeed, but allows the use of standard musical software to interface with the server.

When the client translation layer starts up it notifies the server of its presence on the network. The server then enumerates the instruments it is currently connected to and any methods they offer outside the standard interface. This step means the client layer is agnostic to changes on the server – instruments can come and go and controls can change without any updates needing to happen on the client end.

Currently the client automatically allocates MIDI channels, starting from 1, in the order they are presented by the server. This order is not necessariy guaranteed, but at the end of initialisation the client will print a list of all the instruments, their channel and what they expect.

2.3 Standard Interface

All instruments in the system should respond to to at least two types of message: note and control. The OSC address patterns they expect these on is `/<name>/note,ii` and `/<name>/control,ii` where `<name>` is the name of the instrument.

By default the client will produce these from incoming MIDI note on and control change messages respectively, choosing the instrument from the channel and sending each of the data bytes as the two OSC integer arguments. The server will listen for these messages even if not defined in the file for the instrument, and the MIDI instrument class produces default messages of a note on or control change on channel 1, with the two OSC arguments making up the two data bytes.

The `/note` message is used by the server for startup tests and latency calibration, so it would be ideal if it elicited a sonic reaction.

3 Client (Client.ck)

This file is ideally all that needs to be run on a client machine to enable easy communication. Note that it is not necessary to communicate over OSC, but translates MIDI into OSC to be sent over the network. As such there is some comunication required. The necessary elements are as follows.

a) The host, consisting of

  i) 4-byte IP address (eg. `192.168.0.1`) or hostname (eg. `localhost`)

  ii) a port number, which is just an unsigned integer.

b) The port on which you would like the server to send communications back (if different from above). This can be specified by prefixing it with `in=`

c) The clients IP address, formatted as above. Unfortunately the client has to actually know this to enable two-way communication, in ChucK there is no way of getting the address of a received message, so we have to specifically tell the server where we are. To avoid confusion this is specified as `self=`*<ip address>* and `in=`*<port>*.

d) The MIDI port to listen on – this must be prefixed by `midi=` can be a number or a name, if it is a name containing spaces it might be wise to surround it in quotes (eg. `midi="IAC Driver 1 Bus 1"`)

If these need to be set, they can be provided as arguments to `Client.ck`. In ChucK arguments are expected to be colon delimited and appended to the name of the file, so the command to run `Client.ck` would become something like

```
%> chuck Client.ck:192.168.33.1:8000:in=8001:midi="IAC Driver Bus 1":\
   self=192.168.33.1
```

These can be specified in any order. If you are running the file from the midiAudicle, you can add the argument list in the text field above the main editor.

The essential values are the IP address of the server and the clients own IP address, without these no communication is possible. The MIDI input port is also likely to be necessary to specify, with no input it will default to 0, which mauy or may not be useful.

3.1 Standard Interface

The standard interface for which all instruments must do something consists only of two standard messages: note and control. Refer to per instrument specifications for detail, they may interpret these slightly differently.

Note This message is intended to initiate a note event. The MIDI representation is a standard Note On, a 3 byte message in which the first four bits of the first byte are always binary 1001 and the second four bits are the channel (note on channel 1 is therefore 144 and channel 16 159). The channel determines the instrument and this should be enumerated shortly after startup. The next two bytes represent the data, typically the first byte gives the note and the second some other information, for example the desired volume. This MIDI mesage is translated into an OSC message with the address pattern `/instrument/note,ii` where `instrument` is the name of the instrument and the two integers specified in the type tag are the two midi data bytes, in the same order.

Control This message is intended to supply some kind of control data. The MIDI representation is a Control Change message, again this is 3 bytes, with the first four bits of the first channel in this case 1011 (giving the whole byte a range from 176–191). The next byte typically specifies which control and the third the value, which are defined per-instrument. The OSC representation is `/instrument/control,ii`, exactly as above.

Instruments will define exactly what these do and may define more complicated systems for control if required.

3.2 Full list of Arguments

In addition to those above there are a number of optional arguments available to access more extended functionality. All of these are specifed as *<key>=<value>* with exceptions noted.

self IP address or hostname of the client machine. Either a string such as `Pauls-MacBook-Pro.local` or a numerical IP adress such as `192.168.33.3`. Defaults to `localhost`.

in Port for return communication from the server, defaults to 50001.

server IP address or hostname for server, specifed as for **self**. Defaults to `localhost`, if an IP address is passed as an argument without a key, it is assumed to the server IP.

out Port to send to the server on, defaults to 50000 (this has to match what is defined in `Server.ck`. A number without a key is assumed to be this.

midi The MIDI port on which to listen, this can be specifed as an index into the list of available ports (beginning at 0) or an explicit name, with or without quotations.

test Whether or not to test and if so which instruments. Possible values are `all` to test every instrument on the server or a comma separated list of names for the server to test. Misspelled instruments will trigger a warning, any present instruments in the list will be tested.

delay Sets the latency compensation mode. Possible values are: `off` for no compensation (default), `on` attempt to run latency compensation for all instruments or a comma separated list of names in which case the system will run delay compensation on the named instruments and assume a latency of 0 for any that are not named. For a more detailed discussion of the latency compensation mechanism and when it might be desirable, see the documentation for `Server.ck`.

4 Server (Server.tex)

This is the main file for the Server. It initialises the attached instruments and triggers them to send information about themselves to new clients. The server discovers which instruments are available by looking in the `Instruments` directory, ignoring all subdirectories. When it discovers a file it attempts to load the appropriate instrument based on the first line of the file. The Server listens for UDP traffic on port 50000 by default, this can be changed by altering `Server.ck`. The server initialises instruments to share a reference to a send `OscRecv` object, all instruments listen on the same port. The handling of instrument input is handled entirely within `Instrument` and its subclasses.

When initialising instruments, the server reads the first line of the file to determine what sort of file it is. It then constructs and instance of the appropriate subclass and calls its `init()` method, passing the the `OscRecv` and the file object. The instrument then handles configuring itself based on the file, with the return value of `init()` indicating whether or not it was successful.

Once all the instruments have been loaded, the server shred begins listening for new clients. A new client announces itself by sending a `/system/addme,si` OSC message to the server, where the two arguments are the hostname or IP address and the port for return communication. Using this information, the client iterates through the list of instruments and calls each ones `sendMethods()` and `sendNotes()` methods. These messages all take an `OscSend` object as their sole argument which they use to send relevant data describing the instrument back to the client.

4.1 Master.ck

5 Master.ck

`Master.ck` is the file which starts the server running. In order to do this it adds all the required files to the virtual machine. In order for the server to work properly with any possible number of new types of instruments it also adds any `.ck` files in the `instruments` directory. To make it easy to move instruments to and from the `Unconnected` directory, `Master.ck` searches recursively through all subdirectories of `Instruments` adding everything it finds. The order in which the files are added to the virtual machine is below.

1. `Util.ck`
2. `Parser.ck`
3. `MIDIDataByte.ck`
4. `MIDIMessageContainer.ck`
5. `Instrument.ck`
6. `MIDIIInstrument.ck`
7. `MultiStringInstrument.ck`
8. Instruments found in `Instruments` and subdirectories. This is done with a pre-order

depth first search.[1]

6 Instrument Data Files

This section defines the data files which specify what instruments are available. It must

1. begin with a "type" field consisting of `type=` followed by a description of a type which the system knows. This can be a constant, shorthand for one of the default generic types (at the moment: `MIDI`) or a file name (ending in `.ck`). In the case of a file name the system will check the first line for `// type=`. It will then attempt to match the remainder of the line to a value defined in the `Server.ck` file, at which point it should add the file to the virtual machine and instantiate an object. Note that this code will have to be added to `Server.ck` when a new instrument is added by a `.ck`.
2. follow this with a name, specified by `name=`*name*
3. the following in no particular order
   a. A port, specified as `port=`*number or name* (if the name of the port is specified, it can be in quotes, but it is not required). Specification of multiple ports may not issue an error, but only a single one will be used and at least one must be given.
   b. Zero or more translation lines `input=message=output` where input is the input midi status byte (disregarding channel), message is the OSC message that it becomes and output is the MIDI message sent to the instrument. The osc message will always have the typetag `ii` with the remaining two input midi bytes (or 0 if it is a smaller message) sent as arguments. In specifying the output message the special terms `$1` and `$2` can be used to refer to the first and second arguments of the osc message. The input status byte (and following =) is optional, omitting it will cause the server to still listen for the specified OSC message and perform the appropriate transform, but will not set up the client to generate the OSC.
   c. Zero or more notes, specified as `note=`<*text*> where <*text*> can be anything except a newline. These notes are displayed by the client on initialisation, so might include information about the ranges and limits of the instrument etc.

Note that messages that do not begin with the name of the instrument will have it prepended. Also if the default messages are not specified, they will be added. A file with 0 translation lines would cause the server to instantiate the object specified with just the default messages, which might be useful.

The files may also contain comments. These are indicated by a #, anything between a # and a linebreak is ignored.

6.1 Example

6.1.1 Basic MIDI File

```
type=MIDI                          #this must come first
name=Example                       #and this second
# and now we are free of order
port="IAC Driver 1 Bus 1"          #could be a number
# this would be enough for only the standard interface
128=/noteoff,ii=129,$1,$2          #pass a note off to channel 2
```

[1] All files with a `.ck` extension are added in the current directory (beginning in `Instruments`) and the process is repeated for each subdirectory in turn. Note that if you want to add a new type of abstract instrument or extend a custom instrument, you may need to pay attention to this order and probably edit `Master.ck` to ensure everything is added in the correct order.

```
224=/pitchbend,ii=225,$2,$2      #only send one byte
144=/note,ii=145,$2,$1   #redefine default note, change order
/block,f=160,64,$1       #message without MIDI input
note=An example file for a MIDI instrument.
```

6.1.2 Example Addition to Class Hierarchy

For a more useful example, see `Kritaanjli.ck`.

```
//type=EXTENDED_TYPE
/* ^^ has to be first line
 * EXTENDED_TYPE must be defined in Server.ck
 * and this file must have been added to the VM by
 * Master.ck
 *
 * a MIDI instrument that sends each
 * message on a different channel */
public class ExtendedType extends MidiInstrument
{
    int chan;
    // need to do some init
    fun int init( OscRecv input, FileIO file )
    {
        // file contains this file
        // we will just hardcode the default msgs
        "ext" => name; // name is a field in Instrument
        string patterns[0];
        MidiMsgContainer noteMsg;
        MidiDataByte d1, d2;
        d1.set( MidiDataByte.INT_VAL );
        d2.set( MidiDataByte.INT_VAL );
        noteMsg.set( 144, d1, d2 );
        noteMsg @=> transform_table["/ext/note,ii"];

        MidiMsgContainer cntMsg;
        MidiDataByte d3, d4;
        d3.set( MidiDataByte.INT_VAL );
        d4.set( MidiDataByte.INT_VAL );
        cntMsg.set( 176, d3, d4 );
        cntMsg @=> transform_table["/ext/control,ii"];

        // choose a MIDI port for output
        if ( !setMidiPort( 0 ) )
        {
            cherr <=
               "Extended Type could not open MIDI port"
                  <= IO.nl();
            return false;
        }

        // let Instrument set up OSC listeners
        patterns<<"/ext/note,ii";
```

```
44          patterns<<"/ext/control,ii";
            return __init( input, patterns );
46      }

48      fun void handleMessage( OscEvent evt, string addrpat )
        {
50          if ( transform_table.find( addrpat ) )
            {
52              // get message define
                transform_table[addrpat].getMsg( event )
54                  @=> MidiMsg msg;
                if ( msg != null )
56              {
                    chan +=> msg.data1;
58                  16 %=> ++chan;
                    mout.send( msg );
60              }
            }
62      }
    }
```

7 Summary of Process

Summary

1. Server starts running
   - **a.** Searches `Instruments` directory, attempts to load files (ignores directories)
   - **b.** Constructs list of instruments from files
     i. MidiInstrument class sets up MIDI output and translation from OSC
     ii. Base class Instrument sets up OSC listeners according to what is defined in file.
     iii. All instruments have two default messages they expect if nothing is specified – `/<name>/note,ii` and `/<name>/control,ii`
   - **c.** Starts listening for new clients
2. Client starts
   - **a.** Sends `/system/addme,si` to Server (with own hostname and port)
   - **b.** Server responds with a series of `/system/instruments/add,s` messages which list the instruments constructed earlier by name.
   - **c.** Any instruments with possible messages beyond the basic two send `/system/instruments/extend,ssi` to the client which contain the name of the instrument, the pattern for the message and the MIDI status byte to transform.
   - **d.** All instruments send any information they know about themselves in `/system/instruments/note,ss` messages where the first string is the name of the instrument and the second is a note about the instrument, probably defined in the data file. [2] The notes get displayed by the client to give the user any information the instrument's designer feels useful.
   - **e.** Client uses this data to construct a table of MIDI input to OSC output and prints details about the instruments connected to the server to the console.

[2] Confusion between `/<name>/note,ii` and `/system/instrument/note,ss` should be avoidable given

f. Client listens for MIDI input on specified port and translates appropriately.

the different typetags and the `/system` prefix, although it is an unfortunate homonym.