Tangle User Manual

February 9, 2014

Contents

1 Quick start instructions

1. Server

i. Verify that all the desired instruments are connected correctly.

ii. Start up the server. Once it has booted, log in either remotely—

```
%> ssh sonics@192.168.33.1
```

or locally using the login screen. If you log in locally to the server, open a terminal window.

iii. Check `Instruments` directory—

```
cd ~/Network/Tangle
ls Instruments/
```

make sure files are present for each of the instruments you intend to use.

iv. Run `Master.ck` – in the shell (this is assuming we are still in the root project directory)

```
%> chuck src/Master
```

v. It will produce a lot of text: double check to make sure no errors were emitted. You could run the server from the miniAudicle, but this is often less stable over long periods of time.

2. Client: On your own machine

i. If you want to send your own OSC to the client

a. Double check the address patterns the server printed at the end of its initialisation, you should be good to go.

ii. If you want to use MIDI run `Client.ck` – it needs some information to function correctly. The minimum is:

a. An address for the server; hopefully 192.168.33.1 or `leoadmins-Mac-mini.local` will work[1]. These will vary from server to server; if on a different server, change accordingly.

b. An IP address for the client machine[2]

c. Using this information, run `Client.ck`. The information above must be passed as arguments. For a server at the address 192.168.33.1, a client IP address of 192.168.33.3 and MIDI on the port "IAC Driver 1 Bus 1" the command would be as follows:

```
%> chuck Client.ck:server=192.168.33.1:self=192.168.33.3:\
        midi=IAC Driver 1 Bus 1
```

A full list of options an be found in section 3.

[1] To find the IP address the server is using for the ethernet either look in the settings or use `ifconfig` to find the IPv4 address.

[2] If this isn't working and you are connecting via ethernet, make sure your computer's wifi is switched off.

2 Start Up procedure

What actually happens when the above instructions are followed:

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.[3] 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.
- **f.** Client checks if any latency calibration has been specified; if so, it sends it to the server. Blocks until server indicates it is complete.
- **g.** Client checks if any test patterns have been asked for, if so sends to server, blocking until notified of completion.
- **h.** Client listens for MIDI input on specified port and translates appropriately.

[3] Confusion between `/<name>/note,ii` and `/system/instrument/note,ss` should be avoidable given the different typetags and the `/system` prefix, although it is an unfortunate homonym.

3 Client.ck options

Options for `Client.ck` are specified via a colon separated list of arguments. All arguments are specifed in the form `<key>=<value>`. A full list of options is in table 1.

Table 1: `Client.ck` options

key	values	Description
server	url or numerical IP in the format AAA.BBB.CCC.DDD	Tells the client the IP address of the server.
self	url or numerical IP in the format AAA.BBB.CCC.DDD	Gives the server a return address to send information about connected instruments.
in	an integer	Port on which Client listens for communication from server (default 50001).
out	an integer	Port on which server is listening (default 50000).
midi	an integer or the name of a MIDI port	MIDI port on which the client listens, defaults to 0. It is better to use a name as the order of these can shift. Available MIDI ports can be found by running `chuck -probe`.
test	a comma separated list of names of instruments (ignores case)	Tells the client whether or not to ask the server to run a test pattern, and if so on which instruments. An empty string or `none` will not cause any tests, `all` will ask the server to test all instruments connected.
delay	a comma separated list of names of instruments.	Tells the client whether or not to ask the server to begin the delay calibration process. Special values `on` or `off` tell the server to calibrate all or no instruments respectively.

An example of a likely command to run the client might then be:

```
%> chuck Client.ck:self=192.168.33.3:server=192.168.33.1:\
midi=IAC Driver 1 Bus 1:test=all:delay=on
```

4 Adding New Instruments

The server discovers instruments by searching the `Tangle/Instruments` directory. Each instrument requires a file which tells the server how to talk to it. This can be a configuration file to be read in or a ChucK source file which contains a sub-class of Instrument. All files that do not end in .ck must begin with `type=`<*something*>, all .ck files must begin `//type=`<*something*> (commented so that ChucK will still compile them). <*something*> must be a key defined in `Sever.ck` which the server uses to decide what type of instrument to instantiate.

4.1 MIDI Instrument Config Files

Files with no extension and a first line of `type=MIDI` are used to generate an instance of MidiInstrument. In general, details are specified by `<key>=<value>` The possible lines in the file are described in table 2. Note that all that is required is a name and a port. The file parser treats each individual line as a potential setting so only one can be specified per line. Comments can be added with the # symbol, which causes the remainder of the line to be ignored.

Table 2: MIDI Instrument Configuration Options

Configuration	specified as	description	required
Name	`name=<string>`	Name of the instrument.	yes
MIDI Port	`port=<string> | <number>`	Which MIDI port to output received commands to.	yes
Translation	`[<number>=]<osc message>=<midi message>`	how to transform incoming MIDI. The first (optional) number is what status byte the client should listen for, the OSC message is what it gets turned into (needs a type tag of 2 numbers) and the midi message is the result that gets sent to the instrument.	no
Note	`note=<string>`	A note about the instrument that gets sent to clients. Everything until the end of the line (except comments) is used, nothing more.	no

Translation lines consist of:

1. A MIDI status byte (the first byte of a MIDI message with the channel bits cleared). This defines the message the client receives to trigger the next events. It is optional, a translation without it will still set up the OSC listeners on the server.

2. An OSC message. If specified without the name of the instrument at the root level, the name will be prepended. The OSC message must have a type tag of two numbers (either integer or floating-point).

3. A MIDI message. This is specified as three bytes separated by commas. The first gives the type of message and the channel necessary to communicate with the instrument. The next two are the data bytes which can be either constants or variables taken from the OSC message. Variables are specified as `$1` or `$2` which refer to the first or second arguments respectively. These can be in any order or repeated.

The default translations for note on and control change are created automatically and passed through to channel one unless redefined. Examples can be found in the `Instruments` directory.

4.2 Subclassing Instrument

To add more complicated logic a new class can be made and stored in `Instruments`. The first line needs to be `//type=<TYPE>` and some edits need to be made to `Server.ck` so that it instantiates the new class when the file is found.

Things to note when creating new instruments:

1. Override `init( OscRecv, FileIO )` – you can more or less ignore the arguments but this is called by the server and is where any initialisation should take place. This function returns true or false depending on whether initialisation was successful.

2. Call `_init( OscRecv, string[] )` at the end (probably return the value it returns). The string array is the list of OSC messages you want to listen for, creation of the listeners is handled by Instrument.

3. Override `handleMessage( OscEvent, string )` to handle incoming messages. The OscEvent is the event that just fired, the string contains its address pattern as there is no way in ChucK to retrieve this from the OscEvent.

4. If you are sub-classing MidiInstrument:

   a. if you want to tell the client to listen for a midi messages, add the number to the field `nonstandard_statusbytes`. This is an associative array, it uses a string as the key. Use the address pattern of the OSC message you want the MIDI to be transformed into.

   b. remember to open a midi port.

`Kritaanjli.ck` provides a simple example of creating a sub-class of MidiInstrument to add some additional logic.