OSC Communication

OSC network communication

See also: NetAddr, OSCFunc

OSC communication between programs is often done to send messages from one application to another, possibly with the applications running on different computers. In SuperCollider this communication is done by creating a NetAddr of the target application and creating an OSCFunc to listen to another application. The underlying protocol of OSC is either UDP or TCP.

Sending OSC to another application

To establish communication to another application, you need to know on which port that application is listening. For example if an application is listening on port 7771, we can create a NetAddr and send it a message:

```
b = NetAddr.new("127.0.0.1", 7771);    // create the NetAddr
b.sendMsg("/hello", "there");    // send the application the message "hello"
with the parameter "there"
```

Receiving OSC from another application

To listen to another application, that application needs to send a message to the port SuperCollider is listening on. Normally the default port is 57120, but it could be something different if that port was already bound when SC started. The current default port can be retrieved with

```
NetAddr.langPort;    // retrieve the current port SC is listening to
```

Or you can retrieve both the IP and the port with:

```
NetAddr.localAddr    // retrieve the current IP and port
```

You can open additional ports using Main: -openUDPPort. This will return a Boolean indicating whether SC succeeded in opening the new port. Or you can just pass a custom port as the **recvPort** argument to OSCFunc and it will open it automatically if not already open.

```
thisProcess.openUDPPort(1121); // attempt to open 1121
thisProcess.openPorts; // list all open ports
```

To listen to incoming messages, an OSCFunc needs to be created in SuperCollider. If the sending application **has a fixed port it sends message from**, you can set the OSCFunc to listen only to messages coming from that IP and port:

```
n = NetAddr.new("127.0.0.1", 7771);    // create the NetAddr
// create the OSCFunc
o = OSCFunc({ arg msg, time, addr, recvPort; [msg, time, addr, recvPort].postln;
}, '/goodbye', n);
o.free;    // remove the OSCFunc when you are done.
```

NOTE: The port 7771 above is the port the other application is sending **from**, not the port SC is receiving on. See the **recvPort** argument in OSCFunc: *new if you want to receive on another port than NetAddr.langPort.

Receiving from an application that is sending from a variable port

Some applications (notably Pd and Max) do not send messages from a fixed port, but instead use a different port each time they send out a message, or each time a patch starts up it picks a random port. In that case the OSCFunc needs to be set up, so that it listens to messages coming from anywhere. You do this by passing nil as the srcID argument.

```
o.free;    // remove the OSCFunc when you are done
```

Testing incoming traffic

OSCFunc has a convenience method, OSCFunc: *trace which posts all incoming OSC messages:

```
OSCFunc.trace(true); // Turn posting on
OSCFunc.trace(false); // Turn posting off
```

Custom OSC message processing

All incoming OSC messages call the message recvOSCmessage in Main. If needed, one can add a custom Function or other object to Main's recvOSCFunc variable. Although one can do this directly using the corresponding setter, it is better to use the Main: -addOSCRecvFunc and Main: -removeOSCRecvFunc to avoid overwriting any other functions that may have been added by class code.

```
// this example is basically like OSCFunc.trace but filters out
// /status.reply messages
(
f = { |msg, time, addr|
    if(msg[0] != '/status.reply') {
        "time: % sender: %\nmessage: %\n".postf(time, addr, msg);
    }
};
thisProcess.addOSCRecvFunc(f);
);

// stop posting.
thisProcess.removeOSCRecvFunc(f);
```

OSC type tags and sclang

Inbound OSC messages must have type tags, or an error will be thrown.

The following conversions are supported outbound:

- Integers become "i". Both OSC and sclang use 32-bit signed integers.
- Booleans become "i", with true translating to 1 and false translating to 0.
- Nil becomes "i", translating to value 0.
- Floats become "f" (32-bit float). sclang's Floats are 64-bit, so the conversion to OSC is lossy.
- Strings and Symbols become "s" (string).
- Int8Arrays become "b" (blob).
- Chars are added as a raw type tag, appending no data to the OSC packet. This is needed to send `$N` (nil), `$T` (true), `$F` (false), `$I` (infinity/impulse). Be careful with this -- if the destination entity doesn't agree that the tag consumes zero bytes, the remaining part of the OSC message may be corrupted!

If `NetAddr.useDoubles` is set to true, then in outbound messages, sclang will use the "d" (double) type tag instead of "f". OSC doubles are 64-bit. It is up to you to ensure that the receiver understands "d".

The following type tags are supported inbound:

- "i" (32-bit signed integer) becomes Integer
- "f" (32-bit float) becomes Float
- "s" (string) becomes Symbol
- "b" (blob) and "m" (4-byte MIDI message) becomes Int8Array
- "d" (64-bit float) becomes Float

- "I" (infinity/impulse) becomes `+inf`
- "N" (nil) becomes `nil`
- "t" (64-bit big-endian fixed-point time tag) becomes Float
- "c" (character) becomes Char

"r" (RGBA color), "S" (symbol), and "[" and "]" (array start and end) are not supported.

If an unrecognized tag is encountered, sclang will make that unrecognized tag into a Char object and add that to the OSC message. It will then skip ahead in the OSC message as if it were reading a string -- it looks for the next null byte and then skips 0-3 bytes for 4-byte alignment.

helpfile source: /usr/local/share/SuperCollider/HelpSource/Guides/OSC_communication.schelp
link::Guides/OSC_communication::