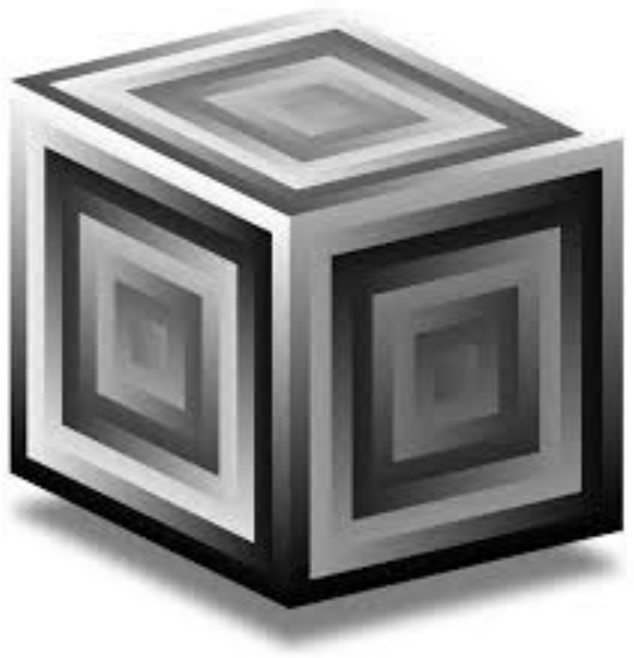
OSC Protocol
(Open Sound Control)

Topics Addressed

- Protocols
- OSC (Open Sound Control)
- Principles of Networking
- Networking in SuperCollider

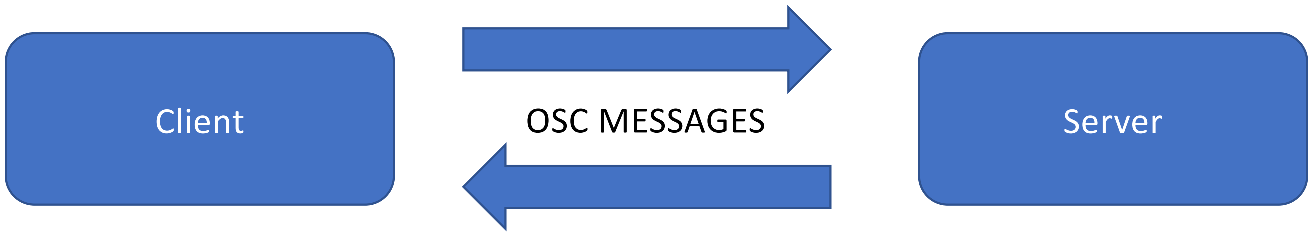


Protocols

- In computer science, a protocol is a set of rules for transmitting data between devices.
  - Example: MIDI!
  - Other protocols: HTTP (HyperText Transfer Protocol), FTP (File Transfer Protocol), etc.
- All data is stored/transferred as bytes. A protocol establishes how to *interpret* those bytes
  - Sidebar: File extensions describe the encoding of bytes in files and how to interpret them as data
- We have seen how MIDI chooses to encode musical data as bytes. OSC (Open Sound Control) is another, more flexible, way of encoding musical data.

Overview of OSC in SuperCollider

- The work done by SuperCollider is divided between two separate processes running simultaneously: the client (sclang) and the server (scsynth).
- How do the two processes communicate with each other? OSC!
- Most of the communication is done from the client to the server, but there are important messages sent from the server to the client.
  - How is the data transferred? Over a network. That network could be local or it could be over something like the internet if the server is on a different computer.
- The details of the messages and the communication process are hidden from the user but can be unearthed with several classes/methods



OSC

- OSC (Open Sound Control) is a protocol for digital music/multimedia devices developed at UC Berkeley at their Center for New Music and Audio Technology (CNMAT).
- Like MIDI, OSC is designed for real-time control of sound/media.
- Benefits (from opensoundcontrol.org):
  - Open-ended, URL-style symbolic naming scheme
  - Symbolic and high-resolution numeric argument data
    - Remember with MIDI data many of the parameters were limited from 0-127
  - Pattern matching language to specify multiple recipients of a single message
    - Single messages can be routed to multiple procedures on an OSC server
  - High resolution time tags (64 bit values)
  - “Bundles” of messages whose effects must occur simultaneously

OSC Protocol

- The unit transmission of OSC is a packet. This is a contiguous block of binary data containing not only the *data* but the *size* of the packet as well (always a multiple of 4). A packet can be one of two things:
  - **Message:** essentially a single command/message to be processed by the server
  - **Bundle:** a series of messages to be executed simultaneously. Includes a 64-bit time tag.
- OSC atomic data types:
  - **32-bit integer:** two’s complement, big-endian
  - **OSC-timetag:** 64 bit, big-endian, fixed-point floating point number
  - **32-bit float:** IEEE floating point encoding, big-endian
  - **OSC-string:** a sequence of non-null ASCII characters followed by 1-4 null characters – total string bytes must be a multiple of 4
  - **OSC-blob:** a 32-bit size count followed by that many bytes of arbitrary binary data (total must be a multiple of 4) – flexibility to send any encoding

OSC Messages

PACKET SIZE	OSC ADDRESS PATTERN	OSC TYPE TAG STRING	ARGUMENTS			
48	/sounds/sine	,ffff	401.5	0.0	1.0	0.0

- OSC Address pattern is a “/” separated address encoding (similar to URL encoding) that specifies the address of a procedure/function to execute on the server
- OSC type tag string defines the type of the arguments for the procedure at the specified address. Earlier versions of the OSC protocol omitted type tag strings. The comma is used to distinguish that there is in fact an OSC type tag string.
- All arguments are multiples of 4 bytes

Type Tags

The standard options for type tag strings are listed below. Implementation specific OSC servers may respond to the type tags on the right but are not guaranteed by the protocol. SuperCollider, for example, does not respond to the type tags S, r, [,].

OSC Type Tags	Type
i	32-bit int
f	32-bit float
s	String
b	Blob

Standard

Non-standard

OSC Type Tag	Type
h	64-bit big-endian two's complement integer
t	OSC-timetag
d	64-bit double
S	Alternate type for strings for systems that distinguish between symbols and strings, for example
c	ASCII character sent as 32-bits
r	32-bit RGBA color
m	4 byte MIDI message
T	True (no bytes allocated)
F	False (no bytes allocated)
N	Nil (no bytes allocated)
I	Infinity (no bytes allocated)
[	Indicates beginning of array (subsequent tags are part of the array until closing bracket)
]	Indicates end of array

sclang UGen Classes

- Classes like `SinOsc` or `Synth` are client side abstractions to formulate and send OSC messages to the client server. Remember the server actually produces the sound.
- sclang is a client side abstraction of the numerous OSC messages that get sent to scsynth
- We could create our own client side program to handle the messaging and eschew sclang entirely. Examples!

Peeling Back the Abstraction

```
(
SynthDef(\sine, {
	arg out = 0, freq = 440, mul = 0.1;
	Out.ar(out, SinOsc.ar(freq, 0, mul)!2);
}).add;
)
```

How do we play \sine?

```
x = Synth(\sine);
x.free;
```

Peeling Back the Abstraction

- The UGen `Synth` is the sclang abstraction for sending an OSC message to the server to create a new instance of the `\sine` SynthDef.
- It turns out that the main role of the class `Synth` is to send OSC messages to the server to create and free synths on the server.
- To send a message to the server, use the instance method `.sendMsg`
  - The 1st argument for `.sendMsg` is the OSC address
  - Subsequent arguments are the arguments for the OSC procedure on the server matching the address

Peeling Back the Abstraction

- The constructor for the Synth object in sclang sends a message to the server.

```
*new { arg defName, args, target, addAction=\addToHead;
        var synth, server, addActionID;
        target = target.asTarget;
        server = target.server;
        addActionID = addActions[addAction];
        synth = this.basicNew(defName, server);
        synth.group = if(addActionID < 2) { target } { target.group };
        server.sendMsg(9, //"s_new"
                defName, synth.nodeID, addActionID, target.nodeID,
                *(args.asOSCArgArray)
        );
        ^synth
}
```

Peeling Back the Abstraction

```
(
SynthDef(\sine, {
	arg out = 0, freq = 440, mul = 0.1;
	Out.ar(out, SinOsc.ar(freq, 0, mul)!2);
}).add;
)
```

How do we play \sine?

```
x = Synth(\sine);
x.free;
```

```
s.sendMsg("/s_new", "sine", 1000, 1, 1);
s.sendMsg("/n_free", 1000);
```

Unique node number

Add Action
(add to tail)

Add to Target
(add to default)

EQUIVALENT!

Peeling Back the Abstraction

```
(
SynthDef(\sine, {
	arg out = 0, freq = 440, mul = 0.1;
	Out.ar(out, SinOsc.ar(freq, 0, mul)!2);
}).add;
)
```

Play a synth with multiple arguments!

```
x = Synth(\sine, [\freq, 880, \mul, 0.15]);
x.free;
```

EQUIVALENT!

```
s.sendMsg("/s_new", "sine", 1000, 1, 1, \freq, 880, \mul, 0.15);
s.sendMsg("/n_free", 1000);
```

OSC vs. MIDI

- Portability
  - The MIDI protocol is universally recognized by music hardware/software. Can essentially play MIDI file/data anywhere
  - OSC can be recognized by any OSC server. Client/Server need to agree on addressing space/methods. OSC provides a syntax for communicating but doesn't agree on a "language" to communicate musical data.
- Flexibility
  - MIDI is restrictive. Data is generally confined to one byte. Only can encode certain musical data though there are alternatives using system exclusive messages.
- Resolution
  - MIDI has poor resolution, especially for parameters like pitch. Restricted to a range of 0-127
  - OSC has up to 64-bit resolution for any datatype and the ability to send even higher, user-defined resolutions via OSC blobs

NETWORKING

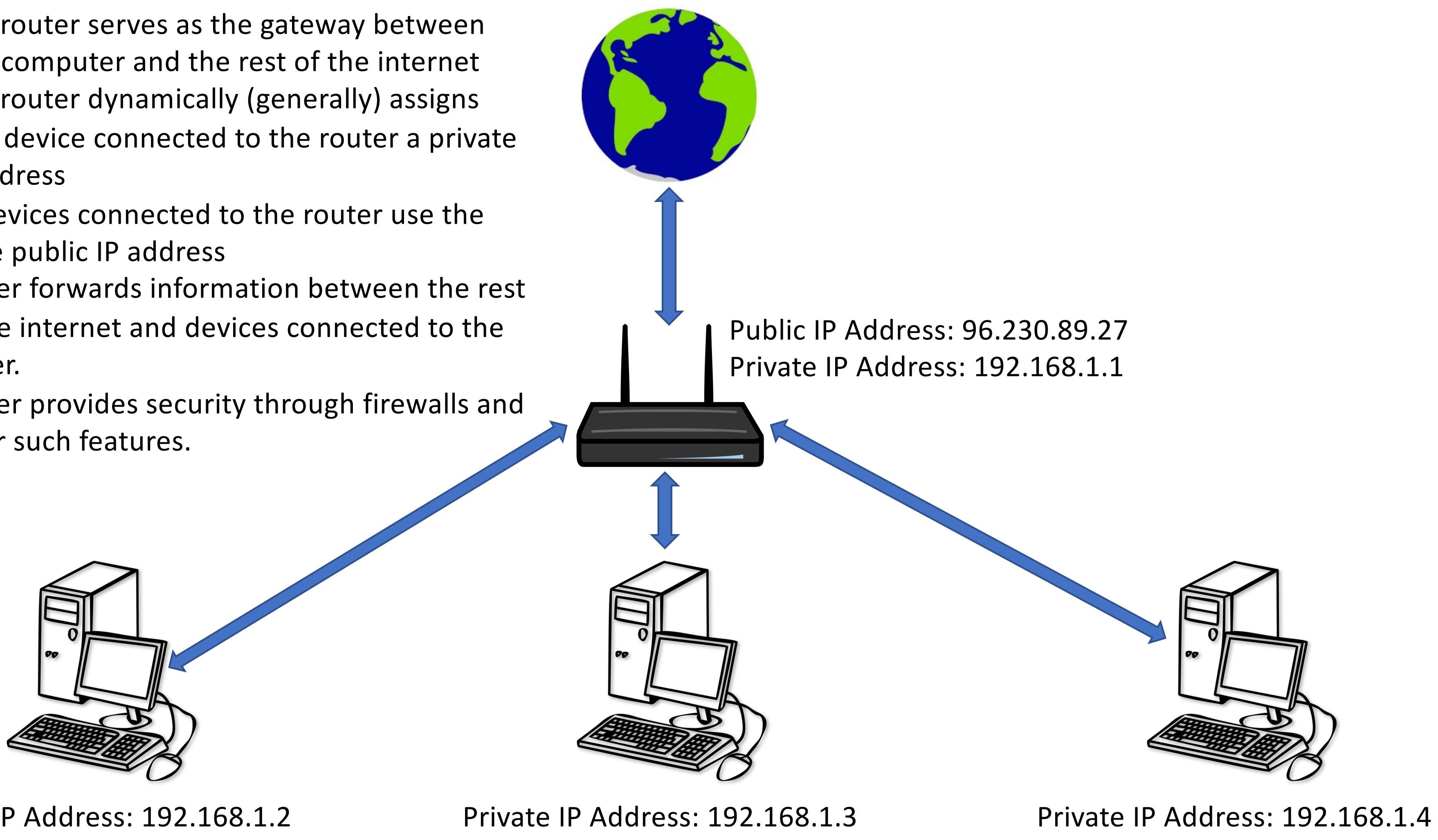
IPv4/IPv6

- IP (Internet Protocol) address: uniquely identifies a device on the internet. Two main addressing conventions:
  - IPv4: 32-bit value (Ex. in human readable form -> "96.230.89.27")
  - IPv6: 128-bit value (Ex. in human readable form -> "2001:0DBB:AC10:FE01::")
  - Why IPv6? We’ve run out of unique IPv4 addresses and now need to provide more address spacing. Only 2^{32} = 4,294,967,296 (4+ billion) combinations out there. We have too many devices!
- Your internet service provider (Comcast, Verizon, etc.) assigns unique IP addresses to each customer
  - Internet service provider is commonly abbreviated as ISP
  - Worldwide authority called IANA (Internet Assigned Numbers Authority) handles the allocation of internet addresses across the world. Your ISP gets a slice of them and assigns them to their customers.
  - Your IP address is likely **dynamic** – ISP could change it as opposed to **static**

Public/Private IP Addresses

- Most computers connected to the internet behind a router
- Routers forward messages from your computer to some destination across the internet
- Your router dynamically (usually) assigns to each device connected to the router a **private** IP address.
  - IPv4 – private IP addresses space ranges from 10.0.0.0 – 10.255.255.255; 172.16.0.0 – 172.31.255.255; 192.168.0.0 – 192.168.255.255
  - Most routers have the private IP address of 192.168.1.1 or 192.168.0.1
- Devices on the router's local network can communicate with each other via the router
- Your router is configured with the **public** IP address given to you by your ISP. All devices connected to the router share the same public IP address.

- Your router serves as the gateway between your computer and the rest of the internet
- Your router dynamically (generally) assigns each device connected to the router a private IP address
- All devices connected to the router use the same public IP address
- Router forwards information between the rest of the internet and devices connected to the router.
- Router provides security through firewalls and other such features.


Public IP Address: 96.230.89.27
Private IP Address: 192.168.1.1
Private IP Address: 192.168.1.2
Private IP Address: 192.168.1.3
Private IP Address: 192.168.1.4

Localhost

- The term localhost in networking refers to the destination address to your own computer when using internet protocols.
  - Provides a loopback interface
  - IP address – 127.0.0.1 (so private IP address)
- When you run SuperCollider on your own computer, sclang communicates with scsynth via localhost
  - `s.addr;` posts -> `NetAddr(127.0.0.1, 57110)`

There's no place like

127.0.0.1

Ports

- A port is a communication endpoint on a device.
- Messages need to be directed to specific programs running on a device. Ports are addresses for destination messages on a given device.
    - For example, scsynth generally uses port 57110 if it is available to receive communication
    - sclang generally uses port 57120 if it is available to receive communication
        - scsynth does need to send messages to sclang in certain instances. Think about the method .poll.
    - Messages sent via HTTP to a web server are received on port 80.
- Servers assign ports to running programs (i.e., processes) so that incoming messages get forwarded properly to the correct application.
    - `s.addr;` posts -> `NetAddr(127.0.0.1, 57110)`

UDP/TCP

- Two types of networking protocols – establishes how computers communicate between each other across a network like the internet
  - UDP – User Datagram Protocol
  - TCP - Transmission Control Protocol
- UDP – Connectionless, unreliable
  - Quick, efficient – packets (i.e., datagrams) can get lost though
  - Good for realtime services like video stream where latency is worse than lost information
- TCP – connected, reliable
  - All packets are guaranteed to reach destination
  - Slower, more latency (if packets need to be resent)
  - Establishes a stream between client and server
- With SuperCollider we will use UDP, but depending upon your project you may favor TCP if going across a remote network and latency is not a huge issue.
  - Note: by default, scsynth is configured with UDP but can be configured with TCP communication.

NETWORKING IN SUPERCOLLIDER

OSCDef

- sclang can be configured to listen to incoming OSC messages and process them via the class `OSCDef`
- Each OSCDef requires a function that will process the incoming message

```
OSCdef.new(
    \msg, // Name in the global dictionary
    {
         |msg, time, recAddr, recPort|
         // msg in the form of [OSC address, arg1, arg2, ...]
         postln("Message from " ++ recAddr ++ ": " + msg[1]);
    },
    '/msg', // A path essentially detailing the name of the listening function
    nil, // nil means to listen to all incoming IPs/ports
);
```

See lecture code

NetAddr

- To communicate across the network with another process, use the class `NetAddr`
- The class NetAddr takes two arguments:
  - 1st Argument: Destination IP Address
  - 2nd Argument: Destination Port
- Messages can be sent using the instance method .sendMsg which sends an OSC message at the destination address and port

```
c = NetAddr("127.0.0.1", NetAddr.langPort);
c.sendMsg("/msg", "Hello from Myself");
```

Loopback to sclang

Laptop Orchestras

- Laptop Orchestra (LO or lork) is a chamber music ensemble where laptops are integrated together to produce musical performances.
  - Examples: SLork (Stanford Laptop Orchestra) - others throughout the country
- SuperCollider, because it’s implementation relies upon networked communication between a server and client, makes it an ideal language to implement laptop orchestra performances.