OSC in SuperCollider Server

James McCartney
OSC in SC

- SC server architecture
- OSC usage in SC server
- thoughts on OSC issues
SC architecture

- client-server
- client: scripting language
- server: synthesis engine
- client and server speak OSC
SC client

- dynamic programming language
- Smalltalk object model
- real time garbage collection
- also borrows from Scheme, APL, Icon, Ruby, HMSL.
SC server

- a virtual machine for audio
- as dynamic as possible
- as simple as possible
- higher level representation left to client
an audio virtual machine

- functional units
- operations on the functional units
- OSC messages are a way of dynamically editing the virtual machine program
functional units

- execution tree
- buffer array
- audio and control buses
execution tree

- internal nodes: groups
- leaf nodes: synths
- synth is a collection of unit generators with a shared lifetime
- unit generator is a basic signal processing element
execution tree

- the tree is the virtual machine’s program
- synths are the subroutines
- unit generators are the instructions
- order of execution: depth first, left to right
execution tree

- all nodes are identified by number
- synths have parameters that can be set
- setting a parameter on a group sets the parameter for all synths it contains
• buffers can contain audio or control data

• most buffer commands are asynchronous:
  • read/write soundfiles
  • alloc, free
  • fill by a function
  • async commands send a reply when done
audio & control buses

- synths do not connect directly
- all synth connections are through buses
- unit generators for:
  - reading, replacing, summing, crossfading
usage of OSC

• obeys time stamps! assumes NTP
• single level name space
• notifications of state changes
• queries
• replies for asynchronous commands
• embedded completion msgs for async cmds
single level name space

• conventional OSC name space not practical

• nodes in the tree may come and go in the hundreds per second (granular synthesis)

• wanted constant time access to nodes

• pattern matching unnecessary

• commands can be hashed
node IDs vs. paths

instead of sending this:

/group0/group1/group101/synth200/freq/set 440
/group0/group1/group101/synth200/amp/set 0.1
/group0/group1/group101/synth200/pan/set 0.7

can send this:

/n_set 200 freq 440 amp 0.1 pan 0.7
notifications

- Notifications are sent when there is a change in the execution tree
- Clients register their interest
- Server maintains a list of interested clients
- Notifications bubble up from the RT thread to be sent on the NRT thread
queries

• state of the server
• state of a node
• state of a buffer
• values of synth parameters
• values of control buses
• values in a buffer
replies to async commands

- async commands reply to the sender when done.
- "/done", "commandName"
- "/fail", "commandName", "errorMsg"
completion messages

• a command to execute when an async command completes
• embedded in the async command as a blob
• type tag ‘b’
issues

• identifying replies
• structured data
• nested bundles
• sequenced bundles
• security
identifying replies

• quoting entire message back - wasteful
• unique numbers in every message - painful
• hash code
  • chance of collision negligible
• very low messaging overhead
structured data

• most RPC schemes can represent rich data (e.g. XML-RPC, SOAP, XDR) such as arrays and key-value pairs (a.k.a. structs, records, maps, dictionaries).

• richer data types allow richer interactions

• OSC has arrays already via ‘[’ and ‘]’ tags

• similarly key-value pairs could use ‘(’ and ‘)’
structured data

• unfortunately no one implements ‘[’ and ‘]’

• most current hosts’ have data types that are too limited.

• more people are beginning to use tools like SC, Lisp, Scheme, Python, Ruby, Javascript which can handle richer data types.
sequenced bundles

- asynchronous messages suspend the bundle
- bundle continues to execute when async command completes
- eliminates need for completion messages
sequenced bundle example

- Bundle contains these commands:
  - load a sound file into a buffer
  - start a node that uses the buffer
  - wait for the node to end
  - free the buffer
nested bundles. Why?

• no additional guarantees (atomicity)
• consecutive bundles have same behavior
• larger packets not good for UDP
• requires reference counting in the host
• provide no benefits, so let’s remove or deprecate them
security

• on a network, an open port with a rich command protocol for initiating tasks on a very high priority thread is an invitation for trouble.

• it would be nice for there to be a log in protocol for OSC
http://www.audiosynth.com
asynth @ io.com