Traducción basada en la ayuda de Just In Time Programming – Library de SuperCollider

Taller de audio del Centro Multimedia del CENART. México D.F. 2011

JITLib (Just In Time Programming - Library)

Es una librería hecha por Julian Rohrhuber, la cual muestra los conceptos básicos de programación interactiva con SuperCollider y *ProxySpace.*

ProxySpace de SuperCollider

Un *proxy,* generalmente es un parámetro de sustitución para algo que, en este caso, es algo tocando en el servidor que escribe a un número limitado de *busses* (puede ser por ejemplo un *synth* o *un event stream*).

Cuando es accedido, ProxySpace regresa un NodeProxy. Una clase similar sin un entorno es Ndef.

Proxy

¿Qué es un *Proxy*?

Un *proxy* es un parámetro de sustitución que usualmente es usado para operar en algo que aun no existe. Cualquier objeto puede tener un comportamiento *proxy* (por ejemplo, este puede delegar funciones de llamado a diferentes objetos). Pero especialmente las funciones y referencias pueden ser usados como operadores mientras que ellos conservan su calidad referencial.

Usando un Ref como Proxy

```
// crear un nuevo objeto de referencia
y = `(nil)

// se puede empezar a hacer cálculos con y aunque su valor no se haya dado aun
z = y + 10

// ahora el origen (y) se puede establecer
y.value = 34

// la función z puede ser evaluada ahora
z.value
```

Usando un función como proxy

```
// una función puede servir para el mismo propósito
y = nil // primero vacío
z = {y} + 10
y = 34
z.value
```

NodeProxy

Para una programación interactiva puede ser útil, estar disponibles para usar algo antes de que este ahí. Esto hace el orden de evaluación más flexible y permite posponer decisiones para más tarde. Usualmente algunos preparativos se tienen que hacer. Como anteriormente, una referencia se tiene que crear. En otras circunstancias esta clase de preparación no es suficiente, por ejemplo, si uno quiere hacer cálculos matemáticos mientras algunos procesos corren en el servidor.

```
//dos proxies en un servidor. La velocidad de cálculo es audio rate, el número de canales es 2
y = NodeProxy.audio(s, 2)
z = NodeProxy.audio(s, 2)

// usalos en cálculo
z.play
z.source = y.sin * 0.2

// establece su fuente ahora
y.source = {Saw.ar([300, 301], 4*pi)}

// la fuente puede ser de varios tipos, uno de ellos puede ser un número
y.source = 0.0

// envía la fuente
y.source

y.clear
z.clear
```

En vías de crear un camino más simple para crear nodos *proxy*, se puede usar un espacio *proxy*.

```
p = ProxySpace.push(s.boot) // se guarda un espacio proxy en p para acceder fácilmente

~z.play
~z = ~y.sin * 0.2;
~y  = {Saw.ar([300,301], 4*pi)}

// limpia el espacio (todos los proxies)
p.clear
```

```
// remueve el proxyspace
p.pop
```

Vista normal de un ambiente (*Environment*)

```
currentEnvironment.postln

~a // accedemos al ambiente: no hay nada guardado: nil
~a = 9 // guardamos algo
~a // ahora 9 esta guardado

currentEnvironment.postln // el valor esta guardado en el ambiente

~b + ~a // causa un error, ~b es nil
~b = -90 // fija ~b

~b + ~a // ahora funciona
```

Fíjate que siempre se puede acceder a los *Environments* (o *ProxySpaces*) desde afuera.

```
x = currentEnvironment.postln
x[\a] + x[\b] // equivalente a ~b + ~a
```

O si "know" es verdadero, puedes acceder a cosas nombradas con la sintaxis de mensaje:

```
x.know = true
x.a + x.b
```

***ProxySpace* como ambiente**

Se puede reemplazar el ambiente actual con un tipo de ambiente especial: un *ProxySpace.*

Este ambiente representa procesos que tocan audio en un Servidor.

```
p = ProxySpace.new(s) // crea un nuevo ambiente, guardandolo por ahora en la variable p
p.push // lo empuja, así que se convierte en el ambiente actual

currentEnvironment.postln
currentEnvironment === p // son idénticos

~x // accediendo se crea un NodeProxy automáticamente (sin inicializar).
~x + ~y // esto funciona inmediatamente porque la post no regresa nil, más bien un proxy
p.postln // ahora hay 2 parámetros de sustitución en el ambiente.
```

Usando el *ProxySpace* para cambiar procesos al vuelo.

```
s.boot

// Tan pronto como una función de sonido (o cualquier entrada compatible) es asignada a proxy ese sonido suena en su propio bus privado (aunque no sea audible aun).

(
~x = {
	RLPF.ar(Impulse.ar(4) * 20, [850, 950], 0.2)

}
)

// El proxy ha sido inicializado para su primer encargo. Toca a velocidad audio (porque le hemos asignado una función audio rate) y tiene dos canales (porque la función tiene una salida estéreo)

~x.index // Un nodoproxy posee un bus privado, entonces su señal puede ser usada de
```

maneras diferentes. ¿Cuál es el índice del *bus* del *proxy*? Esto muestra el índice en la *postwindow* antes que este sea .ir(nil), ahora esta inicializado a .ar(2).

~x.bus // ¿cuál es el *bus* del *proxy*?

~x.play // Ahora escúchalo. Un monitor es creado y toca la señal dentro de un *bus* público – por defecto, este es el bus 0, el primer *bus* de salida. Esta función de monitoreo es independiente de *proxy.*

Las funciones pueden ser cambiadas en cualquier momento.

¿Cuándo es que se inicializan los nodos *proxies*?

La inicialización de un nodo *proxy* ocurre tan pronto como este es usado por primera vez. Posteriormente las entradas son ajustadas a ese bus, tan pronto como es posible.

~z2 = {LFNoise0.kr([1,2,3,4])} // Un *proxy control rate* de 4 canales.
~z2.bus.postln

~z100 = 0.5 // Un valor constante crea un *proxy control rate* de canal sencillo.
~z100.bus.postln

~z34.ar(3) // El primer acceso (con un argumento de numChannels) asigna el bus.
~z34.bus.postln // Un *audio proxy* de 3 canales.

~z34.clear // Esta inicialización puede ser removida usando *clear.*
~z34.bus.postln

Esta inicialización ocurre cada vez que el *proxy* es usado por primera vez. Más tarde, el *proxy* puede ser accesado cuando sea necesario con otra combinación de *rate/numChannles* (los *rates* son convertidos, los *numChannels* son extendidos por envoltura (wrapping), las

fuentes con muchos canales son envueltos.

Nota que esto tal vez cause una inicialización ambigua, en cuyo caso el *proxy* debe se siempre inicializado primero. Un típico problema se demuestra aquí:

~u.play(0,2) // Inicializa 2 canales de audio (*default*). 0 es el número de bus de salida. Si el *proxy* no es inicializado, toca por *defualt* en dos canales. Aquí se hace explícito, solo para hacer esto más claro.

~u = {PinkNoise.ar(0.2)} // usa solo uno
~u.numChannels // dos canales
~u.clear

Si se evalúa de la otra manera, solo un canal es usado.

~u = { PinkNoise.ar(0.2) } // Inicializa un canal de audio.
~u.play(0,2) // Toca dos canales: un canal es expandido en dos. *numChannels* de .play por *default* toma el *numChannels* del *proxy*.
~u.numChannels // 1 canal.
~u.clear

Esto puede ser útil para inicializar *proxies* que usan entradas de tipo variable de manera explícita.

~b.kr(8); ~c.ar // Inicialización explícita.
p.postln // Muestra el *ProxySpace* entero.

Moviendonos afuera del *ProxySpace*

// Toca el audio
~x.play

```
~x={PinkNoise.ar(0.5)}

// p es el espacio sustituto.
p.postln

// para eliminar todos los procesos en p, usa end:
p.end(2) // 2 segundos de fade out.

// para eliminar todos los objetos del bus y liberarlos del bus asignado, usa clear.
p.clear

currentEnvironment.postln

// restablece el ambiente original.
p.pop

currentEnvironment.clear // borra todo en el ambiente normal.
```

Usando *ProxySpace* junto con otros ambientes

Usando *ProxySpace* como un esquema de acceso para nodos proxies puede ponerte en el camino del uso normal de ambientes como pseudo variables. Aquí hay algunas formas de arreglarselas con esto.

```
// puedes también acceder al ProxySpace y a los proxies de manera indirecta.
p[\x].play
p[\x] = {SinOsc.ar(450,0,0.1)}

// o cuando el ProxySpace es empujado, puedes usar un ambiente normal indirectamente.
p.push
d = ()
```

```
d[\buffer1] = Buffer.alloc(s, 1024)
d.use {~buffer1.postln; ~zz = 81;} // para más de uno accede al ambiente, usa .use

// para acceder directamente al interior del ambiente de un ProxySpace, por ejemplo para comprobar si un proxy existe, podemos usar .envir:

p.envir.postln
p.envir[\x].postln // un Proxy con este nombre existe.
p.envir[\nono].postln // no existe un Proxy con este nombre.
```

Estructura interna de Nodo *Proxy*, orden de los nodos y el contexto de los parámetros.

Un Nodo *Proxy* tiene dos contextos internos en los cuales los objetos están insertados: el grupo, que es el servidor, y el *nodeMap* que es un un cliente lateral del parámetro en contexto. Tal y como el grupo puede contener un orden de sintes, es una representación al lado del cliente, en donde la fuente de objetos esta guardado.

```
// haz un nuevo espacio

p = ProxySpace.push(s.boot)
~z.play; ~y.ar // explícitamente inicializa los proxies.
```

Ranuras de NodoProxy

Un nodo *proxy* puede contener diferentes objetos en un orden de ejecución. El índice puede ser cualquier integro positivo.

La ranura inicial 0 es usada cuando se asigna directamente. ~y aun no es usada, lo añadiremos más tarde.

```
~z = (~y * pi).sin * 0.1 * { LFSaw.kr(LFNoise1.kr(0.1 ! 3).sum * -18).max(0.2) }
```

Otros números de ranuras son accedidos por enteros positivos:

```
~y[1] = { Saw.ar([400, 401.3], 0.4)}
~y[0] = { Saw.ar([300, 301], 0.4)}

// para remover uno de ellos

~y[0] = nil

// ¿qué se encuentra en el índice 1?
~y.objects[1] // una interfaz que puede ser tocada
~y.objects[1].source.postcs // la función que fue puesta
~y.objects.postcs // esto regresa los objetos en las ranuras
~y.source.postcs // esto regresa solo la función en la ranura 0
```

Múltiple asignación

```
Esta función esta asignada de las ranuras 1 a la 4
~z[1..4] = {SinOsc.ar(exprand(300, 600), 0, LFTri.kr({exprand(1,3)} ! 3).sum.max(0)) * 0.1}

// la función esta asignada a las ranuras 1,2, y 3 (subsecuentemente).
~z[1..] = [ { SinOsc.ar(400) * 0.1 }, { SinOsc.ar(870) * 0.08}, {SinOsc.ar(770) * 0.04}]

// si no se especifíca una ranura, todas las demás ranuras estan vacías.
~z = { OnePole.ar (Saw.ar([400, 401.3], 0.3), 0.95)}

~z.end
~y.end
```

Tiempo de desvanecimiento

Si establecemos el *fadeTime* nos permite hacer *cross fades*. En el caso de un *proxy audio rate* el *fade* es pseudo-gaussiano. En el caso de un *proxy control rate*, este es lineal.

```
~z.play

~z.fadeTime = 5.0; // 5 segundos
~z = { max(SinOsc.ar([300, 301]), Saw.ar([304,304.3])) * 0.1}
~z = { max(SinOsc.ar(ExpRand(300, 600)), Saw.ar([304,304.3])) * 0.1 }

// El fadeTime puede ser establecido efectivamente en cualquier momento
~z.fadeTime = 0.2
~z = { max(SinOsc.ar(ExpRand(3, 160)), Saw.ar([304, 304.3])) * 0.1}
```

Nota que el fadeTime es también usado para las operaciones xset y xmap.

Play/stop, send/free, pause/resume

Hay un par de mensajes que un *Node Proxy* entiende que estan relacionados con *play* y *stop*.

Play/Stop

Este par de mensajes esta relacionado con la función de monitoreo del *proxy. Play* comienza a monitorear, *stop* detiene el monitoreo. Si el grupo proxy esta tocando (esto puede ser probado con .isPlaying), *play* no afectará de ninguna manera el comportamiento interno del *proxy*. Solo si no esta tocando (por ejemplo, porque hemos liberado el grupo mediante cmd-punto) esto empezará el *synth/objects* en el *proxy*.

```
// primero presiona cmd-punto.
~z = { max(SinOsc.ar(ExpRand(3, 160)), Saw.ar([304, 304.3])) * 0.1}
```

```
~z.play // monitorea el proxy
~z.stop // nota que ahora el proxy sigue tocando, pero solo en privado
~z.isPlaying //¿está tocando el grupo? Si.
~z.monitor.isPlaying //¿está tocando el monitor? No.
```

Puedes pasar un argumento *vol* para ajustar el volumen del monitor sin afectar el volumen interno del bus del *proxy*.

```
~z.play (vol:0.3)

// mientras tocas tu también puedes ajustar el volumen.
~z.vol = 0.8
```

Send/Release

Este par de mensajes controla el sinte dentro del *proxy*. Este no afecta el monitoreo (ver abajo). s*end* enciende un nuevo sinte, *release* libera el sinte. s*end* por omisión libera el último sinte. Si el sinte se libera por sí mismo (doneAction:2) *spawn* puede ser usado.

```
~z.play // monitor. Esto inicia también el sinte, si el grupo no esta tocando.
~z = { SinOsc.ar(ExpRand(20, 660) !2) * Saw.ar(ExpRand(200, 960) ! 2) * 0.1}
~z.release // libera el sinte. El fadeTime en uso es usado para el desvanecimiento.
~z.send // envía un nuevo sinte. El fadeTime en uso es usado par la entrada.
~z.send // envía un nuevo sinte, desvanece el viejo.
~z.release
~z.stop
~z.play // monitor. Como el grupo todavía esta tocando, esto no inicia el proxy.
```

Para liberar el sinte y el grupo de manera conjunta, se usa *free.*

```
~z.free // esto tampoco afecta el monitor.
```

~z.play // monitor. Como el grupo no esta tocando, esto inicia el *proxy*.

Para liberar el sinte y el grupo, para el *playback*, *end* es usado.

~z.end(3) // termina en tres segundos.

Para reconstruir el sinte en el servidor, usa *rebuild*. Esto es por supuesto menos eficiente que *send*, pero tiene sentido.

Por ejemplo, el *synthdef* tiene elementos *random*. UGens como rand(300, 400) crean nuevos valores aleatorios en cada envío, mientras que el cliente lateral con funciones *random* como exprand(1, 1.3) solo se consigue construir una vez; para forzar nuevas decisiones con este, uno puede usar *rebuild*.

```
(
~z = {
	Splay.ar(
		SinOsc.ar(Rand(300, 400) + ({exprand(1, 1.3)} ! rrand(1,9)))
		* LFCub.ar({exprand(30, 900)} ! rrand(1, 9))
		* LFSaw.kr({exprand(1.0, 8.0)} ! rrand(1, 9)).max(0)
		* 0.1
	)
};
)

~z.play
~z.rebuild
~z.send // send solamente crea un nuevo sinte, nueva freq, todo se mantiene igual.
~z.rebuild // reconstruye el synthdef, re-decide el número de osciladores.
~z.end
```

Pause/resume

Cuando se pone en pausa, un nodo *proxy* aun permanece activo, pero cada sinte que es comenzado es puesto en pausa hasta que el *proxy* es resumido nuevamente.

```
~z.play
~z.pause // pone en pausa el sinte.
~z = { SinOsc.ar ({ExpRand(300, 600)} ! 2) * 0.1 } // puedes agregar una nueva función que esta en pausa.
~z.resume // reanuda y sigue tocando.
```

Nota que pause/resume causa clicks con *proxies audio rate*, esto no sucede cuando se pone en pausa *proxies de control rate*.

Clear

Clear remueve todos los sintes, el grupo, el monitor, y libera el *bus*.

```
~z.clear
~z.bus // No bus
~z.isNeutral // No esta inicializado.
```

Nota que cuando otros procesos usan el nodeproxy esto no es notificado. Entonces limpiar tiene que ser hecho considerando esto.

El contexto de los parámetros

¿Qué pasa con los argumentos de las funciones?

```
~y.play
~y = { arg freq=500; SinOsc.ar(freq * [1, 1.1]) * 0.1}
```

Ahora el argumento freq es un control en el sinte (justo como en SynthDef) así que puede cambiar con el mensaje *set*.

```
~y.set (\freq, 840)

// a diferencia de los sintes, este contexto es guardado y aplicado a cada nuevo sinte:

~y = { arg freq=500; Formant.ar(50, freq * [1, 1.1], 70) * 0.1}
```

xset es una variante de *set*, para hacer *crossfade* con el cambio usando el actual *fadeTime*:

```
~y.fadeTime = 3
~y.xset (\freq, 600)

// el mismo contexto es aplicado a todos las ranuras:

~y[2] = { arg freq=500; SinOsc.ar(freq * [1, 1.1]) * LFPulse.kr(Rand(1,3)) * 0.1 }
~y.xset(\freq, 300)
```

El contexto de los parámetros también puede mantener los *bus mappings*. Un control puede ser mapeado a cualquier *control proxy*:

```
~c = { MouseX.kr(300, 800, 1)}
~y.map(\freq, ~c)

// aquí el contexto también  es guardado:

~y = { arg freq=500; Formant.ar(4, freq * [1, 1.1], 70) * 0.1}
```

xmap es una variante de *map*, para hacer *crossfade* con el camino usando el actual *fadeTime*:

```
~y.set(\freq, 440)
~y.xmap(\freq, ~c)
```

Para remover un *setting* o *mapping*, usa unmap / unset.

```
~y.unmap
```

También los controles multicanal pueden ser mapeados a un *proxy* multicanal usando map:

```
~c2 = {[MouseX.kr(300, 800, 1), MouseY.kr(300, 800, 1)]}
~y = { arg freq=#[440, 550]; SinOsc.ar(freq) * SinOsc.ar(freq + 3) * 0.05}
~y.map(\freq, ~c2)
```

El contexto de los parámetros puede ser examinado:

```
~y.nodeMap
```

```
// aparte de los parámetros explícitamente fijados, este contiene el índice del bus y el fadeTime
```

```
p.clear(8) // limpia todo el proxy space, en 8 segundos
```

4 El tiempo en Proxy

Los cambios que ocurren a un *NodeProxy,* más importante posicionando su fuente, normalmente son hechos cada vez que el método *put* es llamado (o, en ProxySpace, la asignación de la operación =). Algunas veces es deseable cronometrar esos cambios en relación a un reloj.

A) Reloj

Generalmente, cada nodo *proxy* puede tener su propio tempo base, usualmente un reloj de *tempo*. El reloj es responsable del ritmo de inserción de nuevas funciones, por defecto en el próximo pulso del reloj.

```
p=ProxySpace.push(s.boot)
~x.play; ~y.play;

// estos dos sintes iniciaron al tiempo que ellos fueron insertados.
~x = { Ringz.ar(Impulse.ar(1), 400, 0.05).dup}
~y = { Ringz.ar(Impulse.ar(1), 600, 0.05).dup}

// agregando un reloj común.
~x.clock = TempoClock.default; ~x.quant = 1.0
~y.clock = TempoClock.default; ~y.quant = 1.0

// ahora ellos estan en sincronía
~x = { Ringz.ar(Impulse.ar(1), 400, 0.05).dup}
~y = {Ringz.ar(Impulse.ar(1), 600, 0.05).dup}

// para simplificar, uno puede proveer un reloj y una cuantización para un proxy space entero:
p.clock = TempoClock.default; p.quant = 1.0;
~y = {Ringz.ar(Impulse.ar(1), 800, 0.05).dup}

~z.play
~z = {Ringz.ar(Impulse.ar(1), [500, 514], 0.8).dup * 0.1}
~z = {Ringz.ar(Impulse.ar(1), exprand(300, 400 ! 2), 1.8).dup * 0.1}
~z = {Ringz.ar(Impulse.ar(2), exprand(300, 3400 ! 2), 0.08).dup * 0.2}
~z.end

p.clear
```

Secuencia de eventos:

Cuando insertas una nueva función dentro de un *proxy*, el *synthdef* esta construido, enviado al *server* quien regresa un mensaje cuando es completado. Entonces el *proxy* espera el siguiente pulso para empezar el sinte. Cuando se usan nodos *proxies* con *patterns*, los *patterns* son tocados usando un reloj como programador.

B) Quant y Offset

Para poder habilitar el control del punto de inserción de *quant/offset*, se puede usar la instancia de la variable de *quant* puede, que también puede ser un número o un *array* de la forma [quant, offset] justo como en un pattern.play(quant).

```
~z.play; ~y.play;
~z = {Ringz.ar(Impulse.ar(2), exprand(300, 3400 ! 2), 0.08).dup * 0.2}
~y.quant = [1, 0.3]; // offset de 0.3, quant de 1.0
~y = {Ringz.ar(Impulse.ar(1), 600, 0.05).dup}
~y.quant =[2, 1/3]; // offset de 1/3, quant de 2.0
~y = {Ringz.ar(Impulse.ar(0.5), 600, 0.05).dup}
```

Al programar, *quant* y *offset*, son usados para las siguientes operaciones:
play, put, removeAt, setNodeMap, wakeUp, rebuild (y las operaciones de reconstrucción lag, setRates y bus_)

C) Conectando el tiempo del cliente y el servidor

Un ProxySpace tiene el método makeTempoClock, que crea una instancia de TempoBusClock junto con un nodo proxy (~tempo) que se mantiene en sync.

```
p.makeTempoClock(2.0) // crea un nuevo tempoClock con 2 pulsos en un segundo
~y.play; ~x.play
```

```
~y.quant = 1; // acomoda quant de regreso a 1 y offset a 0
~y = {Ringz.ar(Impulse.ar(~tempo.kr), 600, 0.05).dup}; // Impulse usa tempo
~x = Pbind (\instrument, \default, \freq, Pseq([300, 400], inf)) // Pattern usa tempoClock

p.clock.tempo = 1.0; // acomoda el tempo a 1.0
p.clock.tempo = 2.2; // acomoda el tempo a 2.2

~x.free
~y.free
```

D) Salida precisa de sampleo

Por eficiencia, NodeProxy usa un simple Out Ugen para rescribir a su *bus*. Si se necesita un sampleo preciso para tocar (OffsetOut), la variable de la clase ProxySynthDef llamada sampleAccurate puede ser acomodada como *true*. Observa que para audio, a través de fuentes externas, esto crea un retraso de uno a más bloques (por ejemplo, alrededor de un milisegundo).

Ejemplo

```
ProxySynthDef.sampleAccurate = false;

~x.play

// el grano se libera el mismo
~x ={SinOsc.ar(800) * EnvGen.ar(Env.perc(0.001, 0.03, 0.4), doneAction:2)}

// tono histérico
(
r = Routine {
        loop {
```

```
            200.do { arg i;
                ~x.spawn;
                (0.005).wait;
            };
            1.wait;
        }
}.play;
)

ProxySynthDef.sampleAccurate = true

// tono estable
~x.rebuild
r.stop
p.clear
```

Referencias

Ayuda de SuperCollider. Just In Time Programming.

Live coding. Recuperado de: http://swiki.hfbk-hamburg.de:8888/MusicTechnology/671