A DISTRIBUTED MEMORY FOR NETWORKED LIVECODING PERFORMANCE

Andrew Sorensen

Queensland University of Technology, Brisbane, Australia

andrew@moso.com.au

ABSTRACT

The symbolic and improvisational nature of Livecoding requires a shared networking framework to be flexible and extensible, while at the same time providing support for synchronisation, persistence and redundancy. Above all the framework should be robust and available across a range of platforms. This paper proposes tuple space as a suitable framework for network communication in ensemble livecoding contexts. The role of tuple space as a concurrency framework and the associated timing aspects of the tuple space model are explored through *Spaces*, an implementation of tuple space for the Impromptu environment.

1. INTRODUCTION

Livecoding is a performance practice that involves writing, modifying and executing computer programs as a means for generating sound and/or visual material in live performance [2]. Often livecoding performances are improvised, with programmers beginning their performances from a conceptual blank slate [8]. Ensemble performances are common in livecoding concerts with two or more livecoders performing co-operatively. Usually the members of these ensembles are connected across a Local Area Network (LAN).

There are a number of livecoding ensembles that have designed bespoke solutions for networked collaboration during livecoding performance, including, but not limited to; SLUB [2], Powerbooks Unplugged [6] and aa-cell [8]. Most of these ensembles have used either application specific interprocess communication (IPC) or the ubiquitous OSC messaging format.

OSC is a simple content format for encoding an address string with a tuple payload. As a messaging format, OSC does not specify the delivery mechanism nor the semantics of a full inter-application communications protocol [3]. Generally OSC is used as a simple abstraction for encoding tuple messages for delivery over UDP [1]. While OSC provides a flexible format, that is suitable for runtime use, there are a number of abstractions that would be beneficial in livecoding performances.

[1] Although more robust delivery protocols such as TCP can also be used to deliver OSC.

Firstly, it would be advantageous for message delivery to be de-coupled, such that performers (and their host machines) could easily enter and leave a performance at any time. A central Cloud supports this idea by removing the need for individual performers to directly connect to one another. A Cloud also supports the notion of persistence, allowing data to live beyond any particular message transaction. New connections to the Cloud could retrieve the current global state, and old connections leaving the Cloud would not destroy existing state. In order to co-ordinate global state between participants, the Cloud should support synchronous access to data. Co-ordination at the level of the Cloud is required to provide multi-platform concurrency in order to avoid race conditions. Any such Cloud would need to be able to support a range of operating systems and language environments. Essentially, the Cloud should act like a distributed global memory, accessible to a range of computer music systems. Finally, a networked environment should allow livecoding musicians to opt in or out of any shared synchronous state and to support runtime decisions about what data types to use and what naming conventions to impose - without the need for prior agreement.

This paper proposes the use of tuple space as a simple and robust framework for a shared, distributed, livecoding memory. The tuple space framework is briefly described, before Impromptu's *Spaces* implementation is introduced. Timing and synchronisation using the *Spaces* implementation are then investigated in some detail.

2. TUPLE SPACE

Tuple space, originating with the Linda co-ordination language [4], is a concurrency framework supporting a model of distributed, associative memory. A tuple space provides a scratchpad where processes may loosely co-ordinate tuples across both time and space. Time, by providing asynchronous storage and retrieval of tuples, and space, by providing access to tuples from any process on a distributed computer network. A tuple space is straight forward to implement and can be provided as a library on top of any extant language. There are tuple space implementations available for many languages including Java, C, Python, Ruby etc..

A tuple space can be thought of as a type of remote bulletin board where tuples can be posted and removed via their unique value signature [2]. Synchronisation is supported by serialising access to any tuples found to match a given signature. A tuple signature can be either an exact match or a conditional match. Conditionals such as TRUE (which matches any tuple element), regular expressions (for string matching) and numeric predicates are common.

Three primary procedures exist for interacting with the tuple space. A `write` procedure; which enters new tuples into the tuple space. A `read` procedure, which returns one or more tuples matching a given signature. And a `take` procedure, which operates similarly to `read` but removes the returned tuple from the tuple space.

If a `read` or `take` cannot find a matching tuple, the call will block the current process. The process will remain blocked until a matching tuple enters the tuple space (via another process calling `write`). If more than one matching tuple is returned, only the first tuple is either `read` or `taken`.

By synchronising on tuples in the tuple space, distributed processes can co-ordinate access to a central shared resource. Consider a single tuple which defines the current tempo <"tempo",120>. All distributed processes are able to share the value by calling `(read ''tempo'' TRUE)` (TRUE will match any value in that position). If any process wished to modify the current tempo then they would proceed by; first taking the tempo tuple `(take ''tempo'' TRUE)`; this would block any other processes trying to access the tempo tuple as it would no longer exist in the tuple space; the taking process would then modify the tempo by writing the tuple `(write ''tempo'' 90)` back to the tuple space; writing the new tuple value back to the tuple space would release any currently blocked processes.

A full introduction to tuple space is beyond the scope of the current paper. The interested reader is encouraged to read one of the many tuple space introductions available on the world wide web before proceeding.

3. IMPROMPTU *SPACES*

Impromptu *Spaces* is a tuple space implementation for the Impromptu programming environment [7]. A *Spaces* server is automatically instantiated whenever Impromptu is started. *Spaces* presence, as an active network service, is broadcast to the LAN using the service discovery protocol Bonjour (Zeroconf). Other Impromptu hosts will receive immediate notification in their logs and can also query for extant services. Impromptu offers two protocols for communicating with the *Spaces* server. Firstly, Impromptu provides access through it's own Inter-Process Communication (IPC) framework. Additionally, Impromptu supports access to the *Spaces* server via OSC [9]. OSC provides systems other than Impromptu with first class access to the *Spaces* server.

Impromptu *Spaces* supports tuples whose elements consist of strings, integers, floats, binary data and arrays, which may contain any of the aforementioned types as well as nested arrays [3]. Tuples may contain any mix of these types, in any order, although by convention a string is often used as the first element. This string element acts in a similar role to the address string used in the OSC format. Tuple elements may be matched using regular expressions for string matching, the numeric operators $=, >, <, <>, <=, >=$ for floating point and integer matches, and the *true* symbol which matches against any value in a given elements position. In addition to predicate matches, Impromptu *Spaces* supports symbol binding on the client side. By specifying a symbol in a particular elements position, the value of that element, in the first returned tuple, will be bound to the given symbol in the current environment [4].

In addition to write, take and read, Impromptu *Spaces* also includes wait. Wait is a blocking read that instead of waiting for a successful match, waits for a failed match.

Impromptu supports an asynchronous concurrency model based around the idea of a temporal recursion [7]. Many temporal recursions run concurrently using a co-operative multi-tasking model. The Impromptu *Spaces* client conforms to Impromptu's overall asynchronous concurrency model. A read or take procedure is responsible for capturing a continuation before sending an asynchronous request to *Spaces* and then immediately breaking to the top-level. This effectively stalls the temporal recursion. As soon as a result is returned from the server, the stored continuation is activated with the returned result and the temporal recursion continues on from where it left off. This asynchronous style makes it very natural to spawn hundreds or thousands of concurrent "green" threads.

Impromptu *Spaces* can be serialised to disk for persistent storage, allowing an extant tuple space to be loaded from disk. Impromptu *Spaces* also supports distribution of the tuple space across any active *Spaces* on the LAN. This provides redundancy in the case of any particular host dropping from the network.

4. SYNCHRONISATION AND TIME

Spaces provides synchronisation between distributed processes by blocking on tuple access. This allows distributed processes to co-ordinate access to shared data. A straight forward synchronisation test can be run to discern the overall latency of the *Spaces* implementation. Ideally we would like to use *Spaces* not only to provide causal order, but also

[2] Traditionally tuple spaces matched against tuple type signatures, not value signatures

[3] *Spaces* was always intended to inter-operate with OSC and this choice of primary types reflects this decision

[4] The symbol is replaced with the *true* element match before being sent to the server. On return the element is bound to the original symbol.

to provide clocked order. The overall latency of the *Spaces* implementation will need to be within perceptual bounds (auditory) if we are to use it's synchronisation primitives for temporal ordering across multiple hosts.

The testing framework we will use sends the audio output of three remote hosts each sending a single channel of audio impulses to a mixing desk. The three input channels are recorded to disk, and then analysed to find the offset between the impulse onset times of each channel. In these example we use notes as the impulse with each host playing one pitch from a c minor chord (60, 63, 67). Pitched notes, while not ideal, are accurate enough for discerning offset, but have the added advantage of providing a more useful listening experience for those interested in a perceptual examination without the bother of calculating sample offsets. The tests were run on a 100M ethernet office LAN with a high level of background network traffic.

Three *reader* processes, each running on a separate host machine, execute a function called loop. The loop function immediately blocks on a `take` while waiting for a given tuple to be made available in *Spaces* (i.e. for another process to `write` the required tuple). As soon as a matching tuple enters the tuple spaces, the `take` returns, binding the pitch symbol to the returned value. Execution immediately proceeds to the play-note command which uses the value bound to `pitch`. Finally loop is recursively called and the process repeats, again blocking waiting for the correctly matching tuple to enter the tuple space.

A single *writer* process, running as an additional process on one of the reader hosts, or on it's own host, is responsible for writing unique tuples for each of the three *readers* twice per second. Each time the *writer* process writes a tuple the *reader* process should immediately unblock and perform the requested note. It is this latency, between one *writer* and the three *readers* that we are interested in analysing.

The code which is run on each of three *reader* hosts is in Listing 1. The hostname would usually come from a call to (sys:hostname) but is artificially represented here as a * to make it easier to follow.

```
(define loop
   (lambda ()
      (spaces:take "note/host*" 'pitch)
      (play-note (now) piano pitch 80 1000)
      (loop)))

;; this starts the recursion
(loop)
```

Listing 1. Sync test, reader code

The code which is run on the *writer* host is in Listing 2. Hostnames in the code below would usually come from Bonjour - here they are specified as host1,host2 and host3.

```
;; repeat once every second
(define timed-loop
   (lambda ()
      (spaces:write "note/host1" 60)
      (spaces:write "note/host2" 63)
      (spaces:write "note/host3" 67)
      (schedule (+ (now) *second*) timed-loop)))

;; this starts the timed loop
(timed-loop)
```

Listing 2. Sync test, writer code

A 30 minute recording of the three hosts output is provided on the website [1]. Analysis of the audio output reveals a skew that is usually below 1ms but can range up to 3ms.

This level of skew can be significantly reduced by introducing a global clock and scheduling events ahead of time. Impromptu includes an NTP clock with a local offset that is calculated using the algorithms 1 and 2 taken from the SNTP RFC [5]. Where t1 is the time that the request is sent by the client; t2 is the time that the request is received by the server; t3 is the time that the response is sent by the server and t4 is the time the that the reply is received by the client.

$$delay = (t4 - t1) - (t2 - t3) \quad (1)$$

$$offset = ((t2 - t1) + (t3 - t4))/2 \quad (2)$$

A burst of messages (10 by default), each calculating message delay and clock offset, are rapidly exchanged and a std-deviation is applied to the round trip delay times. Any combined burst delays which do not fall within a given tolerance are ignored. Impromptu uses these tools to provide a (clock) call with microsecond synchronisation between all Impromptu hosts on a local LAN. The (clock) call returns an NTP timestamp.

The code in Listing 3 and Listing 4 shows the small changes required to make use of the global clock. Another three channel 30 minute audio recording shows the microsecond accuracy achieved by introducing a global clock [1]. Analysis of this audio file confirms that sample offset skew between the three audio channels is less than 100 microseconds - or 5 samples at a sample-rate of 44100.

```
(define loop
   (lambda ()
      (spaces:take "note/host*" 'pitch 'ctime)
      (play-note (clock->samples ctime) piano pitch 80 1000)
      (loop)))
```

Listing 3. Global Clock Test - reader code

```
;; once every second
(define timed-loop
   (lambda ()
      (let ((time (clock)))
         (spaces:write "note/host1" 60 time)
         (spaces:write "note/host2" 63 time)
         (spaces:write "note/host3" 67 time)
         (schedule (+ (now) *second*) timed-loop))))
```

Listing 4. Global Clock Test - writer code

So far we have been discussing the idea of a periodic clock time. By introducing a metronome it is possible to

extend this same principle to an adjustable tempo and beat synchronisation. Impromptu includes a metronome object, a linear function representing the clock time for any given beat at the current tempo. Impromptu's metronome maintains the current tempo, a real number representing beats per minute (BPM), a clock-time (NTP timestamp) specifying the time at which the BPM was set, and the current beat count (as a rational value). *Spaces* provides a unique tuple <"spaces:tempo" time bpm beats> that provides the current state of the *Spaces* metronome. Processes can use this tuple to accurately set their own local metronomes. Any process can update the *Spaces* metronome by setting the BPM at a specific clock-time.

The following test of the metronome functionality uses the same testing framework as the previous two examples. However, in this example, all note synchronisation is performed by the local metronome, which is updated only when a global metronome change occurs. Each client host now has two separate processes running, a player, which plays one note per beat at whatever tempo is currently being set, and a *reader* process which listens for tuple tempo changes. A final "server" process randomly sets the *Spaces* metronome between the ranges of 60 and 260 BPM. The tempo changes may occur either rapidly or slowly. A 30 minute recording is available [1] and the the code is presented in Listing 5. It is worth noting the robustness of the algorithm as it quickly adjusts to any delayed tempo updates. Again the offset accuracy is very high.

```
;; reader plays 1 note every beat
(define loop
   (lambda (beat)
      (play-note (*metro* beat) piano 67 120 1000)
      (callback (*metro* (+ beat 1)) 'loop (+ beat 1))))

(loop (*metro* 'get-beat 4))

;; reader listening for tempo changes
(define tempo-watch
   (lambda ()
      (spaces:read "spaces:tempo" 'ctime 'bpm 'beats)
      (set! *metro* (make-metro bpm (cons ctime beats)))
      (spaces:wait "spaces:tempo")
      (tempo-watch)))

(tempo-watch)

;; writer randomly sets tempo changes
(define random-tempo
   (lambda (beat dur)
      (spaces:set-tempo (clock) (cosr 160 100 1/32))
      (callback (*metro* (+ beat dur)) 'random-tempo
                (+ beat dur)
                (random (cons .95 1/4) (cons .05 8)))))

(random-tempo (*metro* 'get-beat 4) 2)
```

Listing 5. A Metronome example - reader and writer code

5. CONCLUSION

A more detailed exploration of the *Spaces* architecture is currently underway, particularly support for *Spaces* on other livecoding systems. The author is also developing a standalone UNIX version.

This paper has detailed some of the synchronisation requirements for a network communications framework for livecoding. Tuple space has been proposed as a suitable framework and *Spaces* has been introduced. The temporal accuracy of the *Spaces* system has been tested and simple code examples provided.

6. ACKNOWLEDGEMENTS

This work has been supported by the Australasian CRC for Interaction Design.

7. REFERENCES

[1] [Online]. Available: http://impromptu.moso.com.au/extras/icmc_2010.html

[2] N. Collins, A. McLean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Organised Sound*, vol. 8, no. 03, pp. 321–330, 2004.

[3] A. Freed and A. Schmeder, "Features and future of open sound control version 1.1 for nime."

[4] D. Gelernter, "Generative communication in linda," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 80–112, 1985.

[5] D. Mills, "Rfc2030: Simple network time protocol (sntp) version 4 for ipv4, ipv6 and osi," *RFC Editor United States*, 1996.

[6] J. Rohrhuber, A. de Campo, R. Wieser, J. van Kampen, E. Ho, and H. Holzl, "Purloined letters and distributed persons," in *Music in the Global Village Conference (Budapest)*, 2007.

[7] A. Sorensen, "Impromptu: an interactive programming environment for composition and performance." *Proceedings of the Australasian Computer Music Conference*, 2005.

[8] A. Sorensen and A. Brown, "aa-cell in practice: An approach to musical live coding'," in *Proceedings of the International Computer Music Conference*, 2007, pp. 292–299.

[9] M. Wright, "Open sound control 1.0 specification," *Published by the Center For New Music and Audio Technology (CNMAT), UC Berkeley*, 2002.