Rtosc - Realtime Safe Open Sound Control Messaging

Mark McCurry
DSP/ML Researche
United States of America
mark.d.mccurry@gmail.com

Abstract

Audio applications which go beyond MIDI processing often utilize OSC (Open Sound Control) to communicate complex parameters and advanced operations. A variety of libraries offer solutions to network transportation of OSC messages and provide approaches for pattern matching the messages in dispatch. Dispatch, however, is performed inefficiently and manipulating OSC messages is oftentimes not realtime safe. Rtosc was written to quickly dispatch and manipulate large quantities of OSC messages in realtime constrained environments. The fast dispatch is possible due to the internal tree representation as well as the use of perfect-minimal-hashing within the pattern matching phase of dispatch.

The primary user of rtosc is the ZynAddSubFX project which uses OSC to map 3,805,225 parameters and routinely dispatches bursts of up to 1,000 messages per second during normal audio processing. For audio applications, rtosc provides a simple OSC serialization toolset, the realtime safe dispatch mechanisms, a ringbuffer implementation, and a rich metadata system for representing application/library parameters. This combination is not available in any other OSC library at the time of writing.

Keywords

Open Sound Control, Realtime, Intra-Process Communications

1 Introduction

Rtosc is a library which provides an OSC 1.1[Freed and Schmeder, 2009] compliant serialization/deserialization, along with a non-compliant matching algorithm. The serialization code was built with general realtime safe use in mind. The matching and dispatch algorithms were designed for simplified integration with existing realtime applications. Rtosc is available under the MIT license at `https://github.com/fundamental/rtosc`.

1.1 Motivation

Rtosc was originally motivated by the need of a messaging protocol within the ZynAddSubFX synthesizer [Paul et al., 2018]. A large number of parameters were directly exposed to the GUI in a manner which made lock-free audio generation difficult and overall make development of new functionality a slow drawn-out process. OSC has been a standard iter-process messaging option since 2002[Wright, 2002], though it was rarely used extensively inside of an application. This characteristic took me by surprise due to the simplicity of the OSC serialization which made it well suited for use in a low computational/memory overhead messaging protocol.

The two primary issues with other implementations of OSC are that they typically used dynamic memory and they had slow dispatch processes. The target for ZynAddSubFX involved processing data on the non-realtime threads as well as the realtime threads, so dispatch, reading messages, and writing messages needed to be done in an efficient realtime-safe manner.

1.2 Other Libraries

Currently there are a variety of OSC libraries available. Common issues with the available implementations at the time of initially writing rtosc were that:

- Many OSC implementation are incomplete
- Almost all OSC implementations did not focus on realtime safe implementation
- Almost all OSC implementations focus on network based inter-app communication
- Some OSC implementations had difficult to use APIs

Based upon their use of C/C++ and the adoption across Linux audio, the most notable

comparable library is liblo. The liblo project [Harris et al., 2018] has a solid reasonably complete implementation with an easy to use API. Other implementations such as oscpack [Bencina, 2016] were examined in initial development, however other C/C++ OSC implementations have limited adoption. Using ubuntu package dependencies as a measure of adoption, the liblo7 package has 42 directly dependent packages (outside of liblo subpackages) and oscpack1 has zero external packages (outside of dev/dbg subpackages).

While liblo has a number of excellent characteristics, it focuses on non-realtime serialization, dispatch, and networking tasks within OSC. For example, message serialization will involve memory allocation and deallocation from the heap, which can take a highly variable amount of time leading to possible time overruns, aka xruns, if used in a realtime context. While liblo acts as a point of comparison within this paper, it is important to note that it targets a different use-case with a number of tradeoffs, which make it suitable for some applications and rtosc for others.

2 C core

Rtosc is broken up into an easily embeddable C core, as well as a set of higher level C++ utility classes. The C core has a variety of methods for encoding/decoding and message matching, though to get started only three functions need to be used:

- rtosc_message(buf, size, path, arg-types, ...)
- rtosc_argument_string(msg)
- rtosc_argument(msg, i)

rtosc_message() will build a OSC message in a provided buffer and will encode all argument types in the OSC 1.1 standard. The types include: **i**:*int32*, **s**:*string*, **b**:*binary-blob*, **f**:*float32*, **h**:*int64*, **t**:*timetag*, **d**:*float64*, **S**:*symbol*, **r**:*rgb*, **m**:*4-byte-MIDI*, **c**:*int8*, **T**:*true*, **F**:*false*, **N**:*nil*, and **I**:*Inf*. rtosc_argument_string() will provide a list of types in an existing OSC message. rtosc_argument() will return the i-th argument through a union. The active union field can be determined via rtosc_argument_string().

Listing 1: Core API example

```
char buffer[128];
const char *value;
//Construct a simple message
rtosc_message(buffer, sizeof(buffer),
              "/test",
              "s", //1 string arg
              "Hello world");
//Say hello world
value = rtosc_argument(msg, 0).s;
printf("%s\n", value);
```

Outside of the simple serialization and deserialization routines there are a number of additional functions

- rtosc_amessage(buf, size, path, arg-types, args[])
- rtosc_message_length(msg, max_len)
- rtosc_itr_begin(msg)
- rtosc_itr_next(itr)
- rtosc_itr_end(itr)

rtosc_amessage() is the non-varargs extension of rtosc_message(), which is more suitable for non-C API bindings. rtosc_message_length() parses a message and verifies if a value message exists in the buffer which is max_len or fewer bytes. The rtosc_itr_* functions quickly iterate through long lists of arguments in complex messages.

3 Message Processing

One of the primary goals of any messaging library is to eventually handle the content of a message. Rtosc application focuses on a largely bi-directional communication between multiple different threads of execution. The four primary responses to a dispatched message are:

- reply - send a message to the client that sent the original message
- broadcast - send a message to all listening clients
- forward - take the current message unmodified and pass it to the next layer
- chain - send a new message to the next layer

Rtosc typically uses a REST-like API, so if OSC application receives "/volume" it should *reply* with the current volume. If "/volume +12.4"(dB) is received, then the OSC application is expected to set the internal volume to

+12.4 dB and then *broadcast* the response to all applications listening to the state of the OSC application in question.

This division between replies and broadcasts makes it possible to attach several different interfaces to a single stateful OSC application. For example, in ZynAddSubFX its graphical user interface will normally be communicating over OSC. While the GUI is running a debug interface, such as oscprompt[1], can be simultaneously connected to the same instance without generating any conflicts.

Chaining and forwarding messages come into play when there are multiple locations a message can be dispatched from. Since rtosc focuses on the realtime dispatch of messages a common configuration is that there is:

1. a dispatch layer on the non-realtime side for handling non-realtime operations such as file loading
2. a dispatch layer on the realtime side for handling most operations and parameter changes
3. a dispatch layer on the non-realtime side for handling responses from the realtime dispatch tree

The first dispatch layer here may choose one or more of several responses. On receiving a message, it can *reply* back to the original source of the OSC message, *forward* it onto the realtime side unchanged, or partially handle the method and *chain* a new message which would go to the realtime side rather than responding to some external client. As these messages are frequently being relayed between the realtime and non-realtime layers, rtosc provides an implementation of a ringbuffer to manage the inter-thread communications.

3.1 Dispatch

Dispatching messages to handlers is a non-trivial portion of each OSC connected application. At a high level dispatch is essentially:

```
handle(message):
 for each callback in callback-list
   if(match(message, callback.path))
     callback(message)
```

OSC complicates this process with pattern such as wildcards in messages. Rtosc, however, targets higher speed matching on a large number of callbacks, so patterns are not typically used in messages, but are used in callback path descriptions. Additionally, rtosc defines dispatch in terms of tree layers.

Consider the OSC path tree shown in Fig. 1. Paths like "/volume" or "/osc3/shape" can be matched to specific callbacks. "osc#5/" indicates a compound pattern consisting of the literal "osc", a number "0", "1", "2", "3", "4", and a trailing "/". Other paths can use optional argument constraints. For example, "detune::f" is composed of the path literal "detune" and then the argument specification ":", ":f" indicates that no arguments are accepted (":") as well as a single float32 argument (":f").

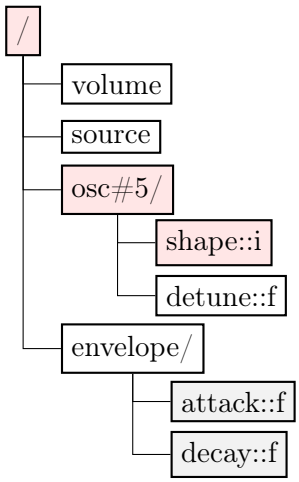

Figure 1: Example Dispatch tree

Other OSC implementations, such as liblo, tend to match an input message directly based upon the full callback path. An example can be matching a OSC message with path "/envelope/attack" or "/*/release" directly on possible destinations "/envelope/attack" or "/envelope/release". Rtosc, on the other hand, favors separate callback definitions/dispatches for each layer. Therefore one dispatch call would try to match "envelope/attack" against "volume", "source", "osc#5/" and "envelope/". Next rtosc would match "attack" against "attack::f" and "decay::f" in a second dispatch layer.

When subtrees are repeated this allows rtosc to have a much more compact representation of the dispatch tree as well as simplifying the difficulty of dispatching at any level. In the case of dispatching "/osc0/shape" (shown in red), the envelope subtree is never dispatched and thus no overhead is produced by the "attack::f" and "decay::f" nodes.

[1] `https://github.com/fundamental/oscprompt`

3.2 Metadata

Moving further outside of the OSC specification rtosc's dispatch structure provides a way to associate metadata with individual callbacks. Rtosc's metadata provides a list of properties which have optional values. Some of the most commonly used metadata properties and definitions are:

documentation - longer descriptions based upon the parameter

shortname - short name useful for labels in user interfaces

min - minimum value

max - maximum value

default - default value when not modified by user

parameter - signifies that this OSC address corresponds to a value which can be read or written to

enumerated - signifies that there are many symbolic values which map onto a series of integer values

map # - mapping of integer value onto a symbolic name for it

scale - specifies the mapping of values to the user perceived range of them (either "linear" or "logarithmic")

unit - states the units that a parameter is in (e.g. Hz, dB, cents)

3.3 Simplified port specification

As callbacks and metadata tends to be repeated, a some syntatical sugar is available for rtosc. Consider a relatively simple parameter accessors/setter with a minimum and maximum value. For a callback in rtosc's tree an associated parameter name and metadata are defined as shown in Listing 2.

Listing 2: Parameter set/get callback

```
{"foo::f", ":parameter\0"
           ":documentation\0"
           "=Foo_parameter\0", NULL,
 [](const char *msg, RtData &data) {
    Obj *obj = (Obj*)data.obj;
    if(rtosc_narguments(msg)) {
        obj->foo =
            rtosc_argument(msg,0).f;
        if(obj->foo > 1.0)
            obj->foo = 1.0;
        if(obj->foo < -1.0)
            obj->foo = -1.0;
        data.broadcast(data.loc, "f",
            obj->foo);
    } else {
        data.reply(data.loc, "f"
            obj->foo);
    }
}}
```

The structure of different accessors are going to share a lot of similar code. Using some macros provided by rtosc, it is possible to instead write an abbreviated form:

Listing 3: Syntactic sugar callback

```
rParamF(foo, rLinear(-1.0, 1.0),
           "foo_parameter")
```

Similar functionality is available via rParamI(), rToggle(), rOption(), and rArrayF(), as well as a few other macros.

Additional utility macros are available for metadata fields as well. One example is rOptions() which is used to define densely packed enums e.g. rOptions(Random, Freeverb, Bandwidth) would define Random as value 0, Freeverb as value 1, and Bandwidth as value 2. rProp(a) defines a generic property 'a' and adds it to the metadata. rMap(a, b) defines a property 'a' which has a value 'b'.

4 Extensions via rtosc messaging & metadata

The metadata associated with rtosc mapped parameters makes it possible to reflect upon the application. While this isn't the primary target of rtosc, there are some notable applications of the metadata so far.

4.1 Self-Documenting

One of the major impacts of having richly documented callbacks is that the API is self-documenting. Each individual OSC based action or parameter can be externally documented in terms of what arguments it requires, what responses should be expected, and what it maps to. At this moment, there are two means of exporting the data: osc-doc and a zyn-fusion specific JSON format. Osc-doc is an XML documentation specification proposed by `https://github.com/7890/oscdoc` and produces a searchable HTML representation of the API similar to doxygen. For Zyn-Fusion a JSON based variant of oscdoc was chosen to avoid the overhead of an XML parser.

Even if the metadata isn't exported to a new format, the existing compiled C-string format

can be transferred to other applications. Oscprompt is one such application and it displays metadata about OSC paths as well as the possible paths which can be tab completed using a reflection based approach.

4.2 Automations/MIDI Learn Support

One use of the metadata exposed through rtosc is a mapping from MIDI or plugin parameters to internal OSC mapped parameters. Given an OSC path (e.g. /part0/PVolume), it is possible to extract the expected type for any OSC messages, the minimum value, the maximum value, and the scaling (linear/logarithmic). Given the metadata, it's possible to define a reasonable default mapping and provide enough information for the user to be able to change the mapping to suit their desires. This functionality is currently being explored within ZynAddSubFX's use of rtosc.

4.3 Undo/Redo support

Within the model that rtosc provides, each OSC message will typically be an action, a state update, or a state read. Since the stream of OSC events contains the state updates that impact the sound engine, the same OSC events can be reused to encode state changes and denote which ones of them are reversible. Rtosc offers one system to capture undoable events and step through their history via undo/redo steps. This approach is similar to non-daw's OSC centric editing 'journal', which stores the programs state as a number of mutations via OSC messages [Liles, 2018].

5 Performance

Rtosc has the goal of providing the necessary information while using a minimum amount of resources. As such, it has been optimized rather extensively and is a very fast tool for handling OSC messages.

One easy point of comparison is against liblo. Liblo is one of the more commonly used OSC implementations within the open source realm; Though by design their API tends to end up allocating memory and producing small data structures. These data structures can produce a notable amount of overhead in OSC heavy systems.

To best compare these two libraries, they were both used to repeatedly encode or decode a message with moderate complexity. The message consisted of the path "/methodname" and arguments: "sif" "this is a string", 123, and 3.14. As can be seen by table 1 rtosc is notably faster in this scenario.

Table 1: Liblo comparison

Impl.	per op	ops per second	speedup
Decoding an average message			
liblo	218 ns	4,600,000	-
rtosc	53 ns	19,000,000	4.1x
Encoding an average message			
liblo	383 ns	2,600,000	-
rtosc	125 ns	8,000,000	3.1x
Dispatch message on single layer			
liblo	530 ns	1,900,000	-
rtosc	54 ns	19,000,000	10x

Rtosc is used in a few performance oriented applications, one of which being the sonic-pi project [Aaron, 2018]. Historically the sonic-pi project used the osc-ruby implementation and then upgraded to an internal subproject, samsosc, which was one effort in producing an optimized OSC implementation. After neither option was satisfactory, the sonic-pi project integrated rtosc via the fast_osc gem[Riley, 2017]. While this isn't an entirely fair comparison, as it crosses different implementation languages, however it provides another picture into the vast performance differences available in such small libraries:

Table 2: Sonic-pi performance stats

Impl	per op	ops per second	speedup
Encoding an average message			
fast_osc	1.2 us	800,000	9.6x
samsosc	3.8 us	260,000	3.1x
osc-ruby	12 us	83,000	–
Decoding an average message			
fast_osc	0.6 us	1,700,000	50x
samsosc	4.7 us	230,000	7.4x
osc-ruby	29 us	34,000	–

In this case, compared to existing options rtosc proved to be significantly faster at reading and writing messages even with the small amount of overhead needed to interface the Ruby and C code.

Beyond the stats recorded for single runs of operations in rtosc, liblo, sonic-pi, etc there are additional scaling behavior to consider. Rtosc's dispatch algorithm scales with the number of subpaths, so using the above numbers it's easy to get a rough approximation for expected DSP load from message dispatch. Message dispatch

time d_t, is roughly a function of path length p_l, time per dispatch layer l_t, and message decoding time m_t:

$$d_t = p_l \times l_t + m_t \quad (1)$$

and DSP load is a function of the rate of messages per second r and the expected average dispatch time per second d_t:

$$\text{dsp\_load} = \frac{r}{d_t} \times 100\% \quad (2)$$

Using the previously calculated timings we can see that even for complex systems with large numbers of messages the overhead of dispatch is low.

Table 3: Projected messaging overhead

path length	msg per second	DSP load
5	100	0.0032 %
20	100	0.011 %
20	10000	1.1 %
10	100000	5.9 %

The following assumptions are used to make the dispatch algorithm more scalable:

1. The tree structure of OSC paths is a way to partition methods.
2. Arrays of parameter should be represented by one port.
3. One OSC message should match one dispatch port.
4. More complex dispatch methods are preferable to a more complex dispatcher.

Item 1 limits the number of matches that need to be considered at each level. The second converts /par0 /par1 /par2 ... /par99 into /par#100. Given the third assumption, it is possible to use techniques such as perfect-minimal hashing to reduce the search space further. Perfect minimal hashing makes it possible to change the matching algorithm for callbacks C and message m from:

```
for c in C:
  if c match m:
     call(c,m)
```

into:

```
c = C[hash(m)]:
if c match m:
  call(c,m)
```

For large collections of parameters, these characteristics help speedup the algorithm immensely. ZynAddSubFX has 3,805,225 unique OSC paths, with an average depth of 6.11 with a maximum depth of 8 subpaths. If perfect hashing occurs at each level, then each input message would on average take 6.11 matches on subpaths, while a flat matching on all possible paths would result in $\sim 1,900,000$ matches with considerably more complexity per match.

Rtosc's approach does impose some restrictions on typical dispatch, however it scales based upon the number of subpaths, or layers. Other solutions will tend to scale based upon the number of possible paths. If the computational complexity is extrapolated from the simple tests for liblo this would result in an average message dispatch time on the order of 18.3 ms while rtosc's dispatch time would be around 380 ns. Equivalently this means the maximum dispatches per second would be ~ 2.6 million for rtosc and ~ 55 for liblo. ZynAddSubFX's use of OSC illustrates an extreme use case which rtosc is well suited for.

6 Conclusions

Prior to rtosc, OSC was a frequently used standard for communication between applications or devices, but library support was lacking for using OSC messages within a realtime safe application. Rtosc provides a realtime safe implementation of OSC messages, message dispatch, as well as several utilities applicable for audio applications. The core interface is written in portable zero-dependencies C code and as has been shown by this paper is performant when compared to other popular implementations. At this moment, the primary users of rtosc are ZynAddSubFX and sonic-pi, though the hope is that other programs will utilize rtosc for efficient and safe OSC message handling in the future.

References

Sam Aaron. 2018. Sonic-pi: The live coding music synth for everyone. `https://github.com/samaaron/sonic-pi`.

Ross Bencina. 2016. oscpack - open sound control packet manipulation library. `https://github.com/RossBencina/oscpack`.

Adrian Freed and Andrew Schmeder. 2009. Features and future of open sound control version 1.1 for nime. In *NIME*, volume 4.

Steve Harris, Stephen Sinclair, et al. 2018. liblo: Lightweight osc implementation. http://liblo.sourceforge.net.

Jonathan Moore Liles. 2018. Non daw. http://non.tuxfamily.org/.

Nasca Octavian Paul, Mark McCurry, et al. 2018. Zynaddsubfx musical synthesizer. http://zynaddsubfx.sf.net/.

Xavier Riley. 2017. fast_osc: A ruby wrapper around rtosc. https://github.com/xavriley/fast_osc.

Matthew Wright. 2002. Open sound control 1.0 specification.