Suitability of gRPC for RMR Communications

Original: 29 May 2020
Revised: 16 June 2020

SUMMARY

It has been suggested that using Google’s RPC (gRPC)[2] as a transport mechanism under the RIC Message Router (RMR)[12] might be advantageous. Rationale for this considering this shift includes:

- Less maintenance (through the elimination of the SI95 code)
- Potentially better performance

The performance of gRPC, both in terms of throughput (messages per second) and latency, was evaluated by constructing two simple applications; one each for sending and receiving messages. Under identical conditions gRPC supports up to a message rate of 40K msg/sec while RMR/SI95 is capable of supporting up to 230K messages per second. Depending on the sending rate the 99th percentile message latency for a gRPC application is between 0.13 ms and 0.40 ms. As a comparison, the 99th percentile message latency for RMR/SI95 applications is between 0.01 ms and 0.04 ms while maintaining a message rate better than 100K messages per second.

The remainder of this brief provides an overview of the gRPC capabilities and offerings, and describes how gRPC might be used to support RMR message based routing. The results of of experiments designed to compare gRPC performance against RMR, both on top of Nanomsg Next Generation (NNG)[9] and Socket Interface-95 (SI95), are also presented.

THE gRPC FRAMEWORK

The gRPC framework is billed as "a high-performance, open source universal RPC framework"[2] supporting four modes of operation. The gRPC *about page*[3] lists several well known companies and/or projects using the framework, and lists support for ten different programming languages. On the surface, the main benefit seems to be the ability to "connect" client and server applications, written in different programming languages, with very little effort. Under the covers the claim is that using gRPC is superior to REST because gRPC performs better than a REST based mechanism[4]. In the world of mobile applications, where many of them have the need to communicate and/or exchange data with remote applications, this all makes sense. However, the model of *connect, exchange some data, disconnect,* is slightly different than the communications model needed by xAPPs.

Bidirectional Streaming Mode
Of the three communications modes supported by gRPC, only the *bidirectional streaming* mode [5] is capable of supporting RMR message based communication. Primarily this is because the other three modes operate using a more traditional request response paradigm where a response is always expected. For RMR communications, a response is not always necessary; requiring one would introduce unnecessary overhead.

Figure 1 illustrates the basic gRPC message and user programme relationship. For each connection which is accepted, the gRPC library invokes the defined user callback function to process the message. When the callback function is defined to expect a stream of data, messages will continue to arrive from the sender (not pictured but assumed to be on the right side of the related connection arrow) until the sender closes the connection.

If concurrent connections exist, a separate instance of each callback function is involved to process the inbound messages. Should the callback functions need to access shared data during the processing of the message(s), then they must implement some type of concurrent access guards in order to prevent corruption.

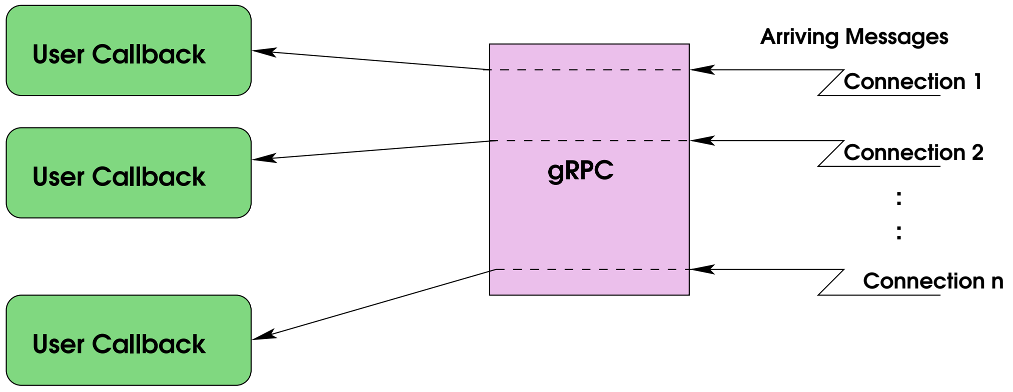

Figure 1: Overview of a gRPC based application.

Message Wrapping and Services

Google Protocol Buffers[11], also known as *protobufs,* are used to wrap the data exchanged between the sending application and the user callback. As with any pair of applications which exchange data, the need exists for the message format to be agreed upon, and the use of protobufs is the the *Google way,* making it no surprise that they were chosen as the base for gRPC.

Further, the protobuf `.proto` file(s) which define the message(s) are also used to define the *services* which are exposed through gRPC. The extension of the standard protobuf description adds information which describes the expected messages and the callback function which should be invoked to parse and react to the message when received.

USING gRPC FOR RMR

RMR is a thin message routing library which provides both high level, message based, routing and transport insulation to user applications[13]. Routing based on *message type* allows the routing to be managed outside of individual applications, and allows for features such as service chaining without the involvement of the sending application. The transport mechanism insulation allows for the best underlying transport (e.g. NNG) to be employed without the need to reimplement parts of each application.

An RMR based application expects messages to be serialised on a single queue regardless of the number of actual senders. Messages are guaranteed to arrive in order from a single sender while possibly interleaved with messages from other senders. To implement this API on top of gRPC, the callback functions illustrated in figure 1 are replaced with a single *receptor* function as illustrated in figure 2.

The Receptor

The *receptor* is a specialised callback which is defined as the service in the protobuf definition[1]. There will be one executing instance of the receptor for every established connection. The sole job of the receptor is to read messages from the connection and to queue them onto a message ring to simulate a single stream to the user programme. This is a similar technique used with the SI95 interface: messages are buffered by RMR until complete (large messages may require multiple reads), then placed onto the message ring. In the case of gRPC, large message construction is handled in the gRPC code, so each read by the receptor yields a complete message.

An immediately obvious disadvantage to this mechanism, which is not a factor when SI95 is used,

[1] The receptor logic would be implemented within RMR in the same manner as the SI95 callback is implemented.

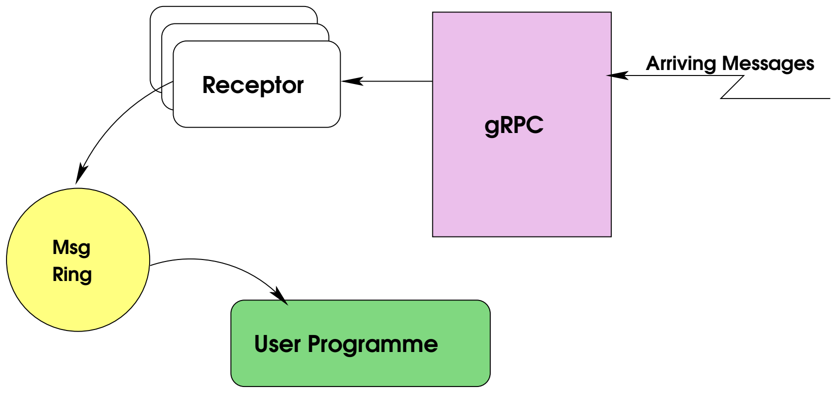

Figure 2: Receptor callbacks feeding a common message ring.

is that an extra buffer copy is necessary to make the message available to the user programme. The message passed to the receptor is *owned* by gRPC, and may not be used after the next buffer is read by the receptor.

Message Encapsulation

With RMR on top of gRPC messages must be encapsulated a second time as the user data is transmitted through gRPC. Because RMR is both the sender and receiver of the data, there is no real need to encapsulate the message, outside of the gRPC requirement. To meet the gRPC requirement for protobuf use, RMR will likely shove its message into the most minimal protobuf possible: a single *bucket of bytes* (BOB). The following illustrates a BOB and is the protobuf definition which was used for the experiments presented in a later section of this brief.

```
message Bob {
   int32  nbytes = 1;   // size of the transmitted message
   bytes stuff = 2;     // a variable size data blob
}
```

Figure 3: The protobuf definition for a bucket of bytes (BOB).

One of the features of RMR when run on top of SI95 is the ability for the user programme to allocate a large message, and to only to transmit the actual bytes used should the resulting payload be smaller than anticipated. Using a BOB allows the underlying transmission to encapsulate only the bytes used ensuring that the bytes transmitted are only the bytes of the payload and no "empty space." Another advantage to using a BOB is that the amount of effort to encode and decode the RMR message is kept to a minimum.

PERFORMANCE

Performance was measured in messages per second so as to be comparable to the data collected when RMR was ported onto SI95. Message counts were collected both in the receptor (white box in figure 2) and in the user programme (green box) with the intent of determining whether or not the multi-writer message ring[2] would have any impact.

[2] Two different message rings/queues were implemented. One was from the Boost library[1] and the other was the message ring code from RMR augmented to support multiple writers. There was no significant difference between the two, and neither had any impact in the overall throughput.

Environment

These tests were carried out on the same hardware as the RMR tests. Each sending and receiving process was executed in a separate container on bare metal. Networking for the containers was set to *host.* The sending application constructed small messages and sent them as quickly as possible (no delay between sends).

Results

For single sender single receiver the average throughput for tests sending at least one million messages was 34763.80 messages per second (measured at the receptor). Table 1 compares the throughput averages for gRPC, NNG, RMR with SI95 and RMR with NNG.

Transport	Messages	Elapsed	Rate
gRPC	3,000,000	85.4 sec	35K msg/sec
NNG	1,000,000	19.3 sec	52K msg/sec [10]
RMR/NNG	1,000,000	missing	38K msg/sec [10]
RMR/SI95	10,000,000	43.5 sec	230K msg/sec

Table 1: Comparison of message rates single sender single receiver.

The gRPC maximum rate of 35K messages per second was observed when sending messages sized to be comparable to the messages sent for both the RMR/SI95 and NNG trials. When message size was reduced to less than 20 bytes a maximum rate of just over 40K messages per second was observed. An expected drop in maximum rate was observed when the message size was increased to 4K; the maximum rate for these tests was about 31K messages per second.

CPU Utilisation

For a single sender/receiver pair, with the sender transmitting as fast as it can, the sender uses the expected 100% of a CPU. The receiver, which starts a thread to digest messages from the queue, allowing the gRPC environment to drive the ingestor callbacks in threads as is needed, uses about 260% (2.6 CPUs).

This is nearly the same as the CPU used by RMR on top of SI95. An RMR sender uses, as expected, 100% of a single CPU when sending full out, while the receiver, one RMR thread reading and queuing data, and the user thread reading from the queue, uses on average 235% (2.35 CPUs).

Latency

A series of tests were conducted with the single sender single receiver applications to measure the message *travel time* (latency) measured from the gRPC send call until it was received in the receptor callback code. Figures 4 through 11 are histograms of this latency data for representative tests at various message rates. Table 2 summarises data from representative executions of the latency experiments.

Several interesting observations can be made when the different executions are compared as a group. First, at low message rates, figure 4, the mean latency is the highest (.3 ms). As message rates are increased the mean latency of messages decreases. This suggests that gRPC is not optimising the underlying TCP connection for low latency, but is attempting to maximise the overall used bandwidth through Nagle's algorithm[14,7,8], or something similar.

Rate msg/sec	Min	Max	50%	95%	99%	Total Sent	Figure
100	.21	.38	.30	.36	.38	1,000	4
850	.14	.60	.15	.25	.32	1,000	
850	.13	.89	.17	.20	.25	100,000	5
4,000	.07	>1.0	.15	.18	.20	100,000	6
8,000	.05	.73	.09	.12	.15	100,000	7
10,000	.05	.75	.07	.12	.13	100,000	8
39,000	.03	>1.0	.05	.08	.14	1,000,000	9
39,000	.03	>1.0	.05	.08	>1.0	1,000,000	10
39,000	.03	>1.0	.05	.08	.20	1,000,000	11

Table 2: Latency data; all times are milliseconds.

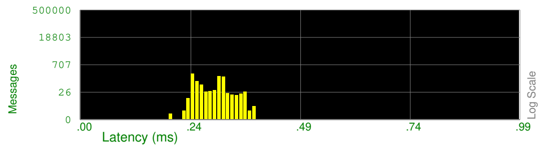

Figure 4: 1,000 messages at 100 msg/sec.

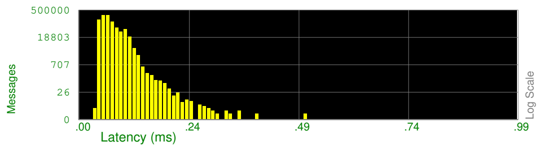

Figure 8: 1,000,000 messages at 10,000 msg/sec.

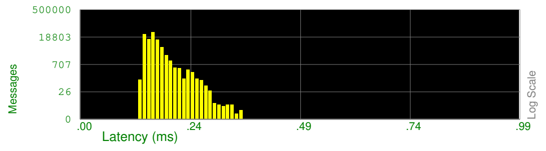

Figure 5: 100,000 messages at 850 msg/sec.

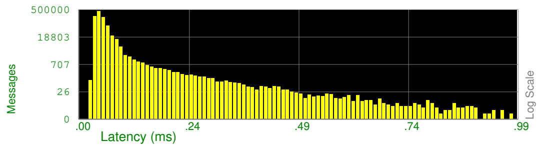

Figure 9: 1,000,000 messages at 39,000 msg/sec.

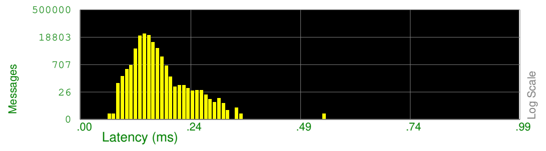

Figure 6: 100,000 messages at 4000 msg/sec.

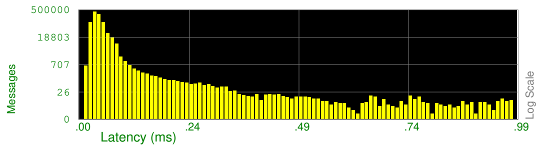

Figure 10: 1,000,000 messages at 39,000 msg/sec.

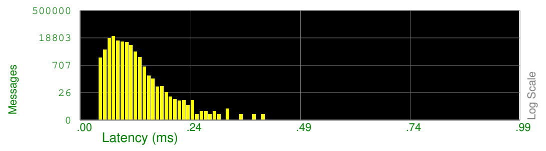

Figure 7: 100,000 messages at 8500 msg/sec.

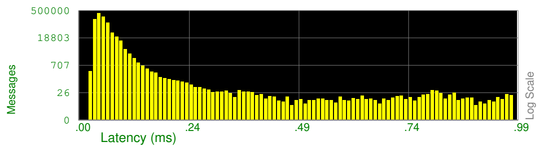

Figure 11: 1,000,000 messages at 39,000 msg/sec.

The maximum rate achieved between to gRPC applications was approximately 39K messages per second. At this rate all of the trials resulted in long *tails* of messages with latency values exceeding one millisecond. For one trial, shown in figure 10, the 99th percentile value itself was over one millisecond.

Latency at low rates

To further examine the latency for applications which might send infrequently a small set of tests were executed to measure latency when sending at rates less than 100 messages per second. The data from representative executions of these tests are presented in table 3.

Rate msg/sec	Min	Max	50%	95%	99%	Total Sent
100	.20	.54	.31	.37	.39	25,000
10	.20	.73	.32	.37	.40	25,000
1	.18	.78	.27	.33	.38	1,800

Table 3: Latency data from low message rate trials; all times are milliseconds.

Comparing to RMR Latency

To compare with gRPC, the SI95 portion of RMR was extended to allow latency to be measured at the same point as it was measured in the receptor; prior to being placed on the message queue. When *untuned* the overall message rate between a single sender and receiver was approximately 300K messages per second, but with latency performance worse than gRPC. This is illustrated in figure 12.

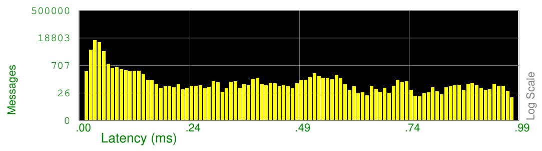

Figure 12: Latency for messages between a single RMR sender/receiver pair at high rate.

Applying some tuning (disabling Nagle's algorithm) the latency was reduced (a 99.5th percentile of .07 ms, and 99.9th percentile of .69 ms). The *cost* of this reduction was throughput which was reduced to less than half (approximately 112K messages per second). The latency distribution with this tuning is illustrated in figure 13.

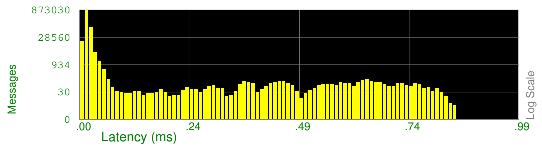

Figure 13: Latency when the RMR sender has disabled Nagle's algorithm.

Further tuning, pinning the receiver processes to individual CPUs, both significantly eliminated the tail, and increased the throughput to approximately 136K messages per second. The 99.9 percentile was .04 ms. Figure 14 illustrates the a representative latency distribution for this configuration.

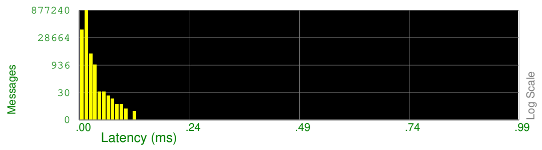

Figure 14: Latency when RMR receiver is pinned and RMR sender has disabled Nagle's algorithm.

The sacrifice of throughput to achieve the lowest possible latency is a well-known networking trade-off, but this doesn't need to impact the performance of xAPPs built on top of RMR. To illustrate, a single sender is able to generate in excess of 250K messages per second when messages are distributed by RMR across more than one receiver. Figures 15 through 18 show the latency distributions for the trials for two different configurations. Figures 15 and 16 show the distribution for the receivers when the sender did **not** disable Nagle's algorithm. The 99.9th percentile latency for this case was .04 ms.

Figures 17 and 18 show the distribution when Nagle's algorithm was disabled. 99.9th percentile latency was further reduced but for this case the throughput was reduced to approximately 50K messages per second.

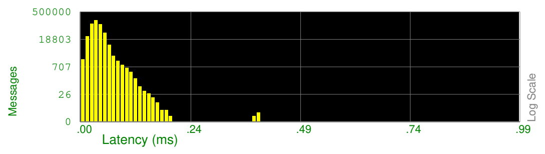

Figure 15: RMR receiver 1; rate approx. 128K msg/sec.

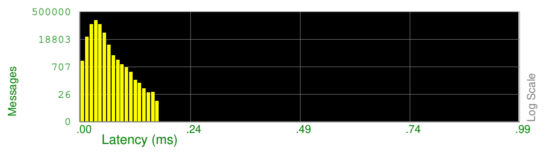

Figure 16: RMR receiver 2; rate approx. 128K msg/sec.

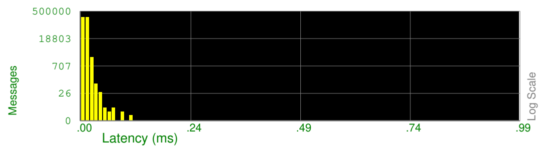

Figure 17: RMR receiver 1; rate approx. 50K msg/sec.

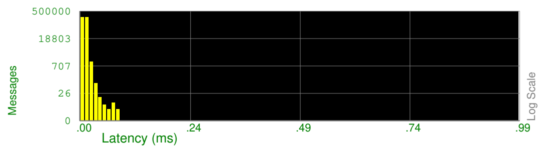

Figure 18: RMR receiver 2; rate approx. 50K msg/sec.

Table 4 summarises the relationship between throughput, latency, use of Nagle's algorithm and CPU pinning. Setting the MTU to 1500 bytes effectively eliminates the impact of Nagle's algorithm, but does limit the total throughput.

Rate	MTU	Nagle	Pinned	99%	Max
300K	9000	ON	neither	>1 ms	>1 ms
112K	9000	OFF	neither	.04 ms	.82 ms
260K	9000	ON	receiver	>1 ms	>1 ms
136K	9000	OFF	receiver	.02 ms	.07 ms
367K	9000	ON	both	>1 ms	>1 ms
138K	9000	OFF	both	.02 ms	.20 ms
137K	1500	ON	both	.05 ms	.13 ms
137K	1500	OFF	both	.02 ms	.22 ms

Table 4: Effect of pinning, Nagel's algorithm and MTU on latency.

Pinning gRPC Processes

Because the effect of pinning the RMR receiver processes had a noticeable impact, the gRPC experiments at max rate were run both pinned and unpinned. Figures 19 and 20 illustrate the latency distributions for representative runs of these experiments. Pinning the receive process had a small effect of reducing the 99th percentile from .87 ms to .52 ms, but did not help to eliminate the tail.

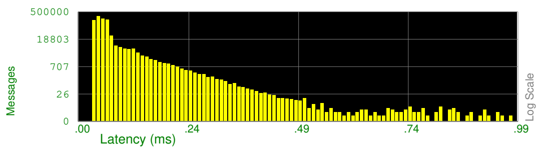

Figure 19: Latency distribution of unpinned gRPC receiver.

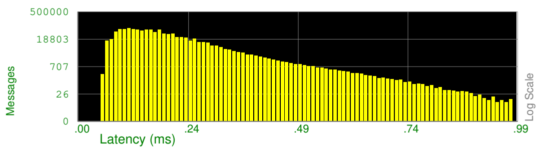

Figure 20: Latency distribution of pinned gRPC receiver.

OTHER COMPLICATIONS

The gRPC library does not support C. There is a C implementation[6], but this is a third party implementation which looks dormant and thus its reliability is unknown; using this library would be a risk. Because RMR is written in C, there would need to be a wrapper layer necessary to allow RMR to interface with the gRPC C++ library; this is necessary to maintain compatibility with existing applications that expect RMR to be a C implementation.

FINAL THOUGHTS

While gRPC provides a nice interface for HTTP styled request response communication between generally unrelated applications, and its claim to be "high speed" is probably valid when compared to REST-like transaction implementations, it just cannot provide the same throughput (messages per second) that can be achieved with RMR on top of SI95. Latency, for the most part, is acceptable, but only for applications which generate messages at a rate above 10,000 per second, and even for those applications the very long latency *tail* observed could cause a significant number of messages to be delayed beyond acceptable limites.

In addition, gRPC is not a replacement for RMR inasmuch as it provides no routing capabilities requiring the user application to know every endpoint address (IP address or DNS name) that is required. Using gRPC directly also requires additional application complexity from the perspective of threading and session management that isn't required with the "single stream" model provided by RMR.

REFERENCES

[1] **"boost C++ Libraries"**
https://www.boost.org/doc/libs/1_73_0/doc/html/lockfree/reference.html

[2] **"gRPC A high-performance, open source, universal RPC framework"**
https://grpc.io/

[3] **"About gRPC"**
https://grpc.io/about

[4] **"gRPC Frequently Asked Questions - FAQ"**
https://grpc.io/faq

[5] **"gRPC Concepts"**
https://grpc.io/concepts

[6] **"Protocol Buffers C RPC implementation"** source repository
https://github.com/protobuf-c/protobuf-c-rpc

[7] **"Nagle's Algorithm"**
https://en.wikipedia.org/wiki/Nagle%27s_algorithm

[8] Nagle, John; **"Congestion Control in IP/TCP Internetworks"**
RFC 896, January 6, 1984
https://tools.ietf.org/html/rfc896

[9] **"NNG Reference Manual"**
https://nng.nanomsg.org/man/v1.3.0/index.html

[10] **"RMR vs NNG Sending Performance"**
https://wiki.o-ran-sc.org/display/RICP/RMR_nng_perf

[11] **"Google Protocol Buffers"**
https://developers.google.com/protocol-buffers

[12] **"RIC Message Routing"**
https://docs.o-ran-sc.org/projects/o-ran-sc-ric-plt-lib-rmr/en/latest/index.html#

[13] **"RMR Overview Manual Page"**
https://docs.o-ran-sc.org/projects/o-ran-sc-ric-plt-lib-rmr/en/latest/rmr.7.html

[14] Trejo, David; **"Nagle's Algorithm"**
http://www.davidromerotrejo.com/2016/09/nagles-algorithm.html?m=1

/usr2/dev/gitlab/grpc_test/doc