RMR_nng_perf

RMR vs NNG Sending Performance

RMR uses NNG's push/pull model for TCP connections between application endpoints. This model is the only NNG model which allows multiple concurrent processes to create connections to an application using a single port (e.g. 123.45.67.99:4560). To address expectations both from the perspective of the capabilities of NNG in this mode, and the overhead of RMR two sets of send/receive applications were used to exchange messages and the message rates were measured.   This page presents the observations when testing with these applications.

Environment

The sender applications attempt to send 1 million messages measuring the elapsed time between start and finish. For each message a buffer is allocated, filled with a static payload and sent. If the message send fails for any reason the application records it as a "drop" and continues; no retries at the application level. For the NNG based sender, NNG was allowed to block for up to 1ms, while the RMR based sender configured RMR with a send timeout of 1 (allowing RMR to retry blocked sends up to 1000 times each).   RMR 1.10.0 was used for the RMR tests.

NNG Send Rates

The following are three representative trials with the NNG only send/receive pair. The sender transmitted 1 million messages in round robin fashion to 5 receivers:

NNG send rates

```
sender_1  | <MSEND> finished  attempted 1000000 messages drops=0 elapsed=19230 ms  == 52002 msg/sec
sender_1  | <MSEND> finished  attempted 1000000 messages drops=0 elapsed=19290 ms  == 51840 msg/sec
sender_1  | <MSEND> finished  attempted 1000000 messages drops=0 elapsed=19246 ms  == 51975 msg/sec
```

RMR Send Rates

Using a route table which directed 5 different message types to specific receivers, the following are representative results of the trials:

RMR send results

```
sender_1  | <A2SEND> finished attempted: 1000000  good: 999985   bad: 0  drops: 15 rate: 38461
sender_1  | <A2SEND> finished attempted: 1000000  good: 999980   bad: 0  drops: 20 rate: 37037
sender_1  | <A2SEND> finished attempted: 1000000  good: 999976   bad: 0  drops: 24 rate: 38461
sender_1  | <A2SEND> finished attempted: 1000000  good: 999987   bad: 0  drops: 13 rate: 38461
```

When a route table was used which directed three message types across six different receivers the results were the same.

Reduced Max RMR Retries

If RMR was configured with a send timeout of 0, preventing RMR from retrying a blocked send more than 100 times, the same the number of "retries" reported by RMR to the application increased (expected). The following are representative of these trials:

RMR sends timeout == 0

```
sender_1  | <A2SEND> finished attempted: 1000000  good: 999435   bad: 0  drops: 565 rate: 38461
sender_1  | <A2SEND> finished attempted: 1000000  good: 999442   bad: 0  drops: 558 rate: 38461
sender_1  | <A2SEND> finished attempted: 1000000  good: 999469   bad: 0  drops: 531 rate: 40000
sender_1  | <A2SEND> finished attempted: 1000000  good: 999436   bad: 0  drops: 564 rate: 38461
```

The conclusion here is that allowing RMR to retry blocked sends for 1000 times has minimal effect on message rate, while improving the number of times that the application must deal with a "retry" condition.

Threads

NNG divides session processing across many threads, so many so that for an application with even 6 connections it is impossible to pin threads to a CPU without overlapping. This is illustrated with a screen capture from top while one of the above RMR tests was executing:

top info

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
39398 root      20   0 1727732   2968   2092 R 361.1  0.0   1:44.54 apples_sender2
38938 root      20   0 5140052   3204   2364 S  92.0  0.0   0:28.37 apples_receiver
38567 root      20   0 5140052   3320   2480 S  91.7  0.0   0:28.12 apples_receiver
39168 root      20   0 5140052   3300   2464 S  91.7  0.0   0:28.19 apples_receiver
39247 root      20   0 5140052   3144   2308 S  91.7  0.0   0:28.14 apples_receiver
38800 root      20   0 5140052   3276   2436 S  91.4  0.0   0:28.16 apples_receiver
39012 root      20   0 5140052   3292   2452 S  90.7  0.0   0:28.03 apples_receiver
```

When multiple sending applications are pushing messages to the same application a similar top capture shows that NNG will process each connection on a separate thread and will span CPUs:

top multiple senders

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
41908 root      20   0 5074516   3180   2344 R 324.5  0.0   0:31.75 apples_receiver
42192 root      20   0 1072372   2928   2104 R 162.6  0.0   0:17.33 apples_sender2
42266 root      20   0 1072372   2900   2076 R 160.6  0.0   0:14.36 apples_sender2
```

O/S Scheduling and/or CPU Affiliation

While no efforts were made to pin any of the application threads to CPUs, it was obvious during single sender/receiver tests that either O/S scheduling or CPU "luck" made a difference in overall throughput as illustrated below with some message rates exceeding the pure NNG application's rate.

Single send/receive

```
sender_1  | <A2SEND> finished attempted: 1000000  good: 999070   bad: 0  drops: 930 rate: 43478
app0_1    | =app0= finished  received: 999070  rate: 34450 msg/sec

sender_1  | <A2SEND> finished attempted: 1000000  good: 998847   bad: 0  drops: 1153 rate: 47619
app0_1    | =app0= finished  received: 998847  rate: 39953 msg/sec

sender_1  | <A2SEND> rmr timeout value was set to 1
sender_1  | <A2SEND> finished attempted: 1000000  good: 998681   bad: 0  drops: 1319 rate: 50000
app0_1    | =app0= finished  received: 998681  rate: 41611 msg/sec

sender_1  | <A2SEND> finished attempted: 1000000  good: 998596   bad: 0  drops: 1404 rate: 50000
app0_1    | =app0= finished  received: 998596  rate: 39943 msg/sec

sender_1  | <A2SEND> finished attempted: 1000000  good: 999093   bad: 0  drops: 907 rate: 43478
app0_1    | =app0= finished  received: 999093  rate: 35681 msg/sec

sender_1  | <A2SEND> finished attempted: 1000000  good: 998339   bad: 0  drops: 1661 rate: 52631
app0_1    | =app0= finished  received: 998339  rate: 43406 msg/sec

sender_1  | <A2SEND> finished attempted: 1000000  good: 999116   bad: 0  drops: 884 rate: 38461
app0_1    | =app0= finished  received: 999116  rate: 33303 msg/sec
```

Conclusions

The impact on message rate when using RMR is caused by a couple of factors:

- overhead needed in RMR to map a message type/subscription ID pair to a route table entry, and then to select an endpoint from the entry
- RMR spinning when a send blocks rather than being able to wait on a pollable event as NNG is capable of

The maximum possible message rate through NNG seems to be capped at just over 50K messages/second. The theory is that this limit is in part due to the locking needed to coordinate the asynchronous I/O model that NNG uses (as opposed to Nanomsg), and is needed to coordinate activities between threads.