# IO Pattern summary

## Objectives

The objective for the IO Pattern testing is to find an IO pattern for RTP connector, current netty based thread model has a configuration issue. The Netty thread model has two type configurations: one is using IO thread plus process thread pool, and another is only has IO thread to process the request. The first one introduces a thread context switch for each message, the second one bind the connection with thread, make the application not scalable.

We want to find a better IO pattern that can be as same effective as netty and easy to configure, the pattern can work with Load balancer, OCS, ABM, etc. It can be work with single connection or multiple connections, and the load in the connection can be symmetric or asymmetric, it should guarantee the fairness for each connection and the most below latency should be very slow. The CPU can be fully utilized and the average latency should be very slow.

## IO patterns

We have tried below patterns:

1. Netty pure IO thread

Each IO thread do select -> read -> process -> write, the connection will be bind with the thread.

1. Netty IO thread + Process thread pool + message blocking queue

IO thread do select -> read, and put request into Queue, the process thread receive message from the blocking queue.

1. Main IO thread + Process thread pool + A blocking queue

The item in the blocking queue is the SelectionKey of the readable channel, the main IO thread do select to get readable channel, then remove the OP\_READ interest and put it in the blocking queue, the process thread will take the readable channel from the queue and take one message, then put it back to the queue if there are more messages, otherwise set the OP\_READ interest and wakeup the selector.

1. One thread pool used Leader follower pattern. The thread pool size equals CPU number.

Actually there are two leader follower patterns in the implementation. First one it use a lock to guard the readable channel assignment, each thread will take a dedicated readable channel. Second one is a lock in the connection to guard the read buffer of a connection. In this pattern, to fully utilize the CPU, we add a queue for processing channels, when a thread is available and there is no readable channel in the selector, the thread can take message from other processing channel which owned by other thread.

During the testing, we found below factors are highly impact performance:

* IO Read - the read is an expensive in non-blocking IO, so you cannot read message from socket one by one, a read buffer will be highly improve performance.
* IO Write – the write is also an expensive operation, and it is synchronized, if there are multiple threads handle messages in one connection, there exists potential write lock contention.
* Sticky connection with Thread – If a thread can sticky with a connection, it will highly reduce the potential read lock contention and writes lock contention on the connection.

### Netty pure IO thread

This pattern supports sticky connection with thread, but the relationship between connection and thread is fixed. It cannot work for asymmetric load, if there are 8 connections, 8CPUs, each CPU will take a connection. If only one connection has messages, only one CPU can be utilized.

### Netty IO thread + Process thread pool + message blocking queue

Compare with the first one, it can fully utilize the CPU, but it did not support sticky, then it will have high potential write lock contention. And a context switch will add the latency.

### Main IO thread + Process thread pool + A blocking queue

This pattern also did not support sticky. Each thread can take a random readable channel, it will cause memory switch between threads. In worse case, each CPU memory cache will have all connection buffers. But this pattern has guaranteed the fairness of each connection.

## Leader follower pattern

During our testing, #4 has the best performance, and it is very easy to configure, only need configure the read/write buffer size, the thread pool size equals CPU number. In theory, #4 is pure LF, there is no dedicated IO thread, and each CPU will be fully utilized if there are messages.

* It supports sticky with thread, a thread will dedicated work for one channel until there is no messages.
* It can fully utilize CPU, because if there is no available channel on the selector, the available thread will steal message from the other threads.
* It reduced the memory copy between threads. In anytime, each CPU cache will only have one connection read buffer. And the read buffer will be valid until there is no message on the read buffer
* It can work with single connection or multiple connections. If there is only one connection, all CPU will work on this connection, if there are multiple connections, each CPU will work on separated connection.
* It is fairness. Since it will take the readable channel first, so a new readable channel will be handled in advance.

State chart of leader follower pattern:



Figure State chart of leader follower pattern

#### Write strategy

During the testing, we found four write strategy:

1. Single Sync write – the response will be write to socket immediately
2. Bulk write – the response will be cached and write to socket when cache full or after all message processed. There are two variants: one is buffer on connection, another one is each thread has a local buffer. The second one is more effective in one connection case, it will less write lock contention. The first one will need a write lock to guard the write buffer of the connection.
3. Async bulk write – it is bulk, and there are a write thread. During our test, there are no big difference between Bulk write and Async bulk write. And Sync bulk write is more effective.
4. Double check sync write – it use try lock and double check pattern to reduce write lock contention between threads, if there are two threads want to write data to socket, only one get the flush lock, and another one will append its data to a buffer, the one who get the flush lock will responsible to write it after the first write to socket finished.

For difference application type, the write strategy may different, for load balancer, it is a IO bounded application, we think the bulk write is feasible, since each message processing time is very minimal. In our testing, the thread local buffer bulk write can reach 160k throughput via the connection write buffer bulk write can reach 150k throughput. For OCS type application, we think the bulk write is not good, because the message processing time is not predictable, if we use bulk write, the latter message processing time will impact previous message latency. We have do testing on the write strategy, in single connection, the single sync write only have 40k throughput, and bulk write can reach 150k throughput and the double check write can reach 120K throughput. Since the double check write guarantee the latency, we think it will be very good for OCS type application.

## Next step

Based on the testing, we think we have got a better solution for the IO Pattern, and it will be easy to configure than current Netty based pattern.

We will continue work on the testing and will evaluate how to migrate current code from the Netty to new pattern.

## Appendix

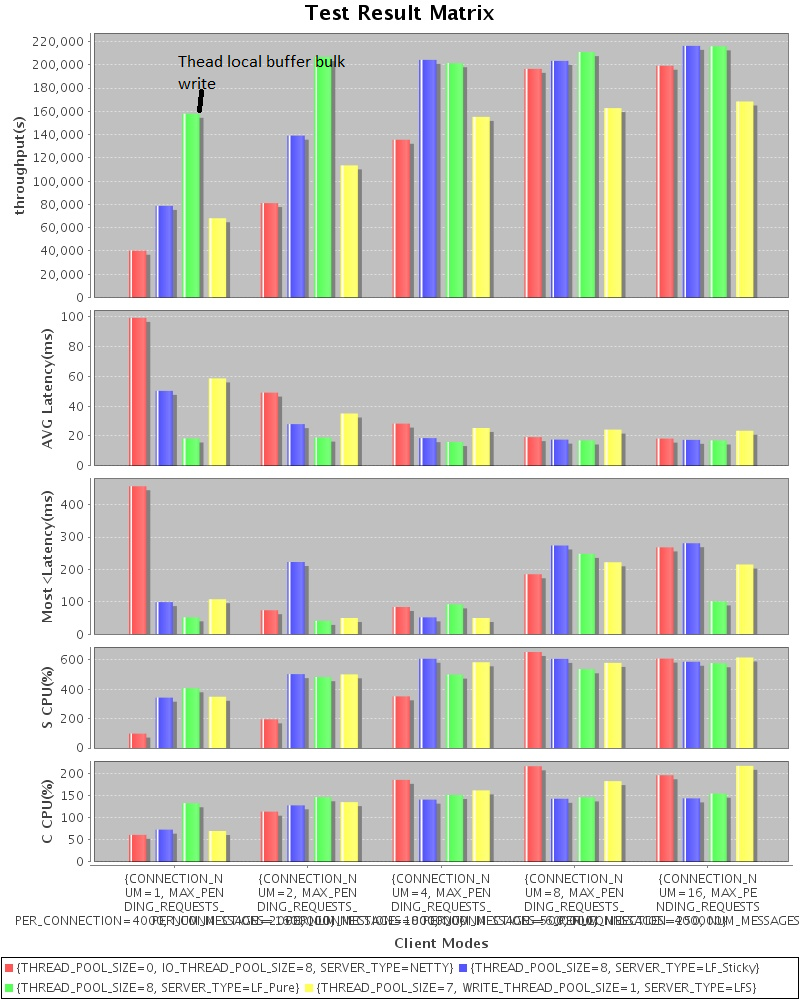


Figure Thead local buffer bulk write

