ACA task engine refactoring

Futurewei Cloud Lab

January 2022

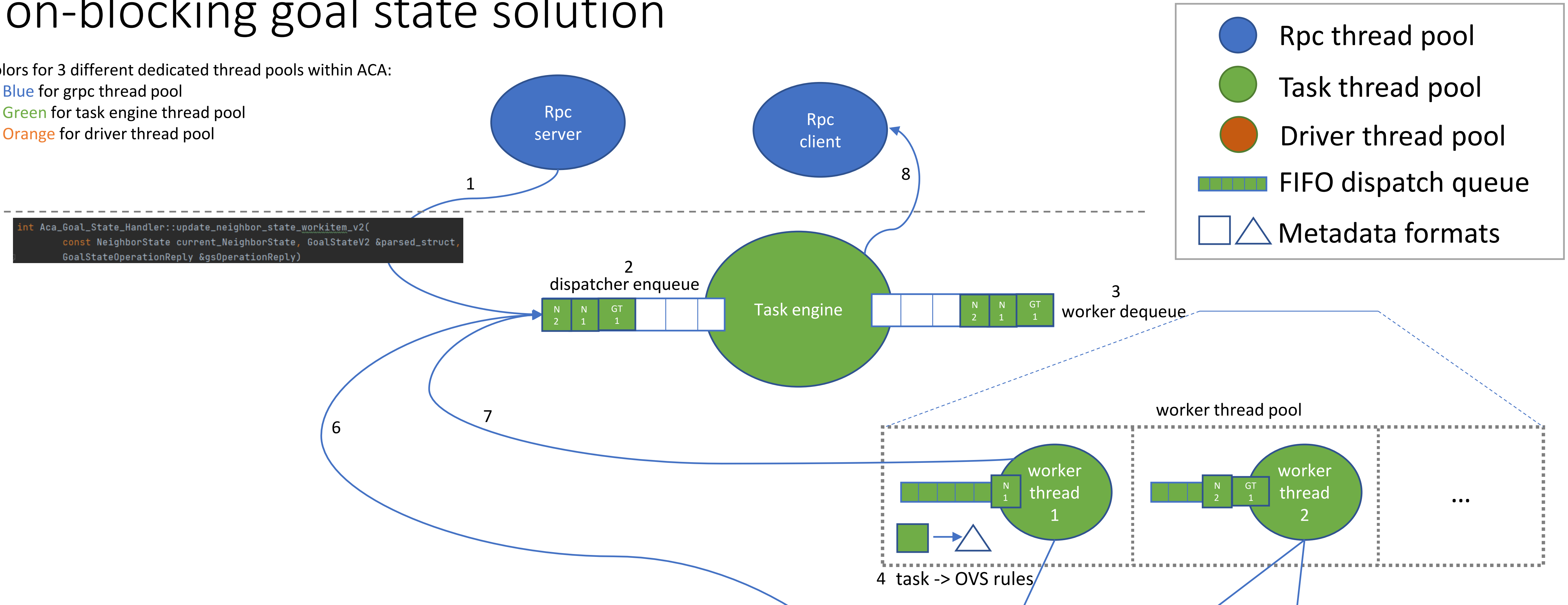
Key performance matrix:

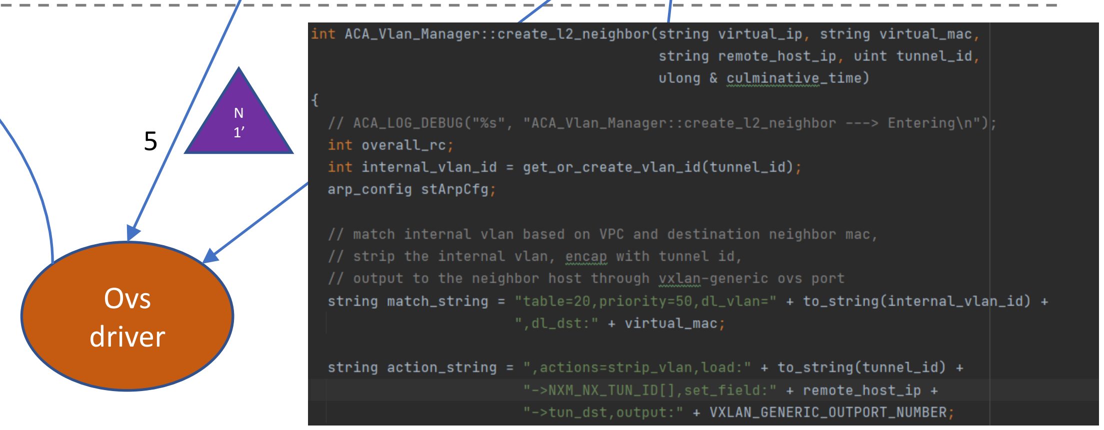
- Within ACA (within localhost) on-demand ping pong throughput
  - ***315k/s*** complete ARP request->response round trips (please find more details in Scenario #1, page 4)
- ACA 1 million goal state propagation latency (see Scenario #2, page 6)
  - Reduced from 58 seconds (10/30/2021 release) to ***13 seconds*** (1/30/2022 release)
  - About ***77k/s*** flow updates (depending on the job complexity)
  - Start time is ACA received rpc goal state (1 giant vpc update including 1m GS updates)
  - End time is when ovs finish all flow updates (add neighbor flows)
- Pure ovs driver throughput (provided in previous 10/30 release)
  - About 110k/s flow updates

Non-blocking goal state solution

3 colors for 3 different dedicated thread pools within ACA:

1. Blue for grpc thread pool
2. Green for task engine thread pool
3. Orange for driver thread pool

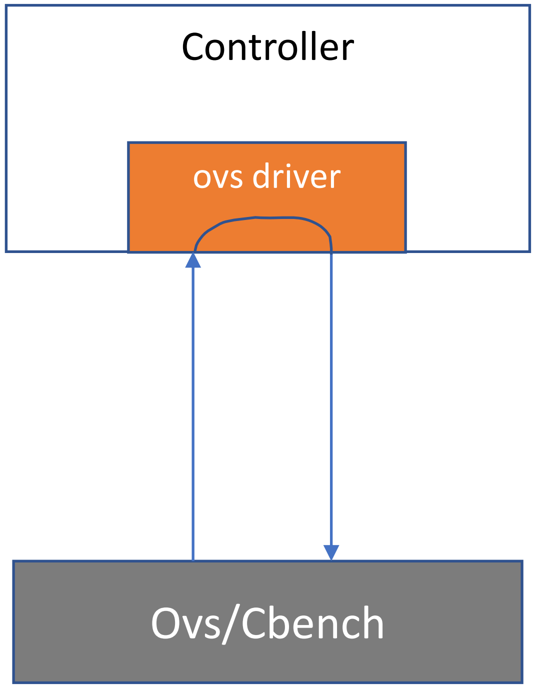




1. ACA's grpc thread pool (as grpc server) receives an GoalStateV2
2. Grpc thread enqueues a task (can be a series of functions in a critical section), and grpc thread is not blocked but keep receiving following ones
3. Inside task framework: worker threads dequeue tasks for parallel/strict-ordered executions
4. For each task: internal logic is mainly about transferring vpc goal states to downstream driver operations (for example, OpenFlow flow-mods)
5. After task translated to flow-mod, then call ovs driver (switch to ovs thread pool) to push flows
6. Besides push flows, ovs driver thread pool also monitors ovs messages for example packet-in events. It's like rpc receiver, it will not handle it but throw to task engine and keep receiving followings
7. If worker thread's current task execution needs to retry, or even divide into more async sub-tasks. It will be an enqueue of new tasks for the engine
8. If anything needs to be reported to NCM, it will be a task executed to call rpc client and send rpc request

Scenario 1: within localhost (ACA) ping pong test

- lib-fluid up and down, no context switching (just to test upper limit of library and also means ACA<->ovs channel capacity)

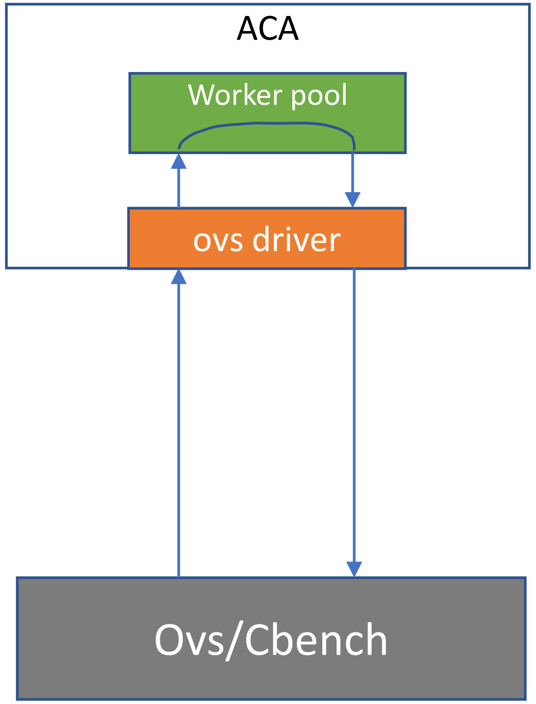


1M fps with both up and down directions combined:

- 500k/s receiving (packet-ins)
- 500k/s sending (packet-outs/flows)

For example conn_id 0 receives and then reply back also to conn_id 0, no context switching and only execute light-weight job within the same thread

- lib-fluid receive -> ACA worker pool -> lib-fluid send (test integration of driver and worker-pool/task-engine)



Before cbench throttling:

1million+/s receiving but only 5k/s sending

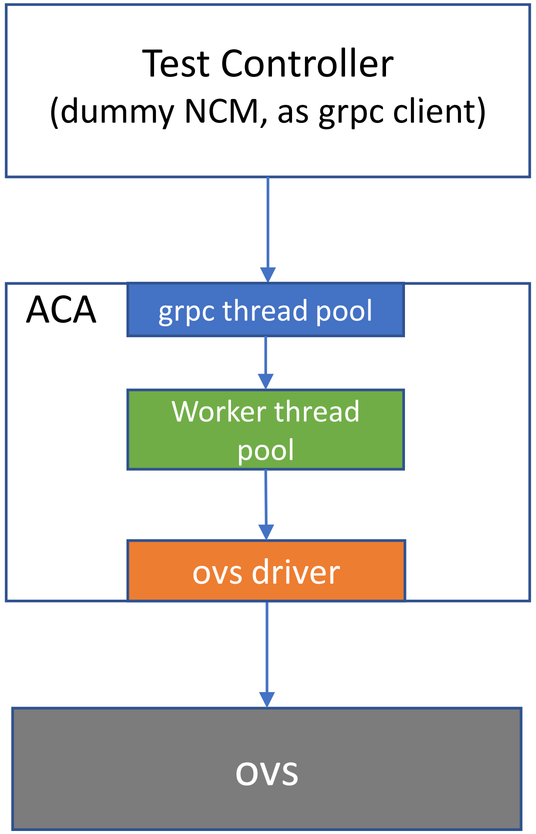
After cbench throttling to balance the system:

500k/s receiving and 450k/s sending (flow-mod)

600k/s receiving and 315k/s sending (on-demand ARP)

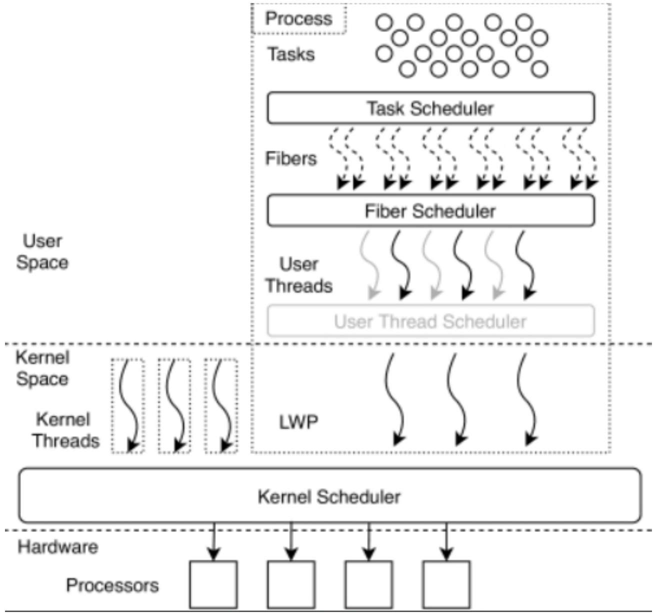
- For example, libfluid ofconn 0 receives packet-in, schedule a job in worker pool (context-switching) and then continue receiving next.
- Job execution is in worker threads, then send ovs updates in libfluid (another context-switching).
- Within the same libfluid connection, receiving and sending throughput are shared since the design is 1 event loop thread per connection.

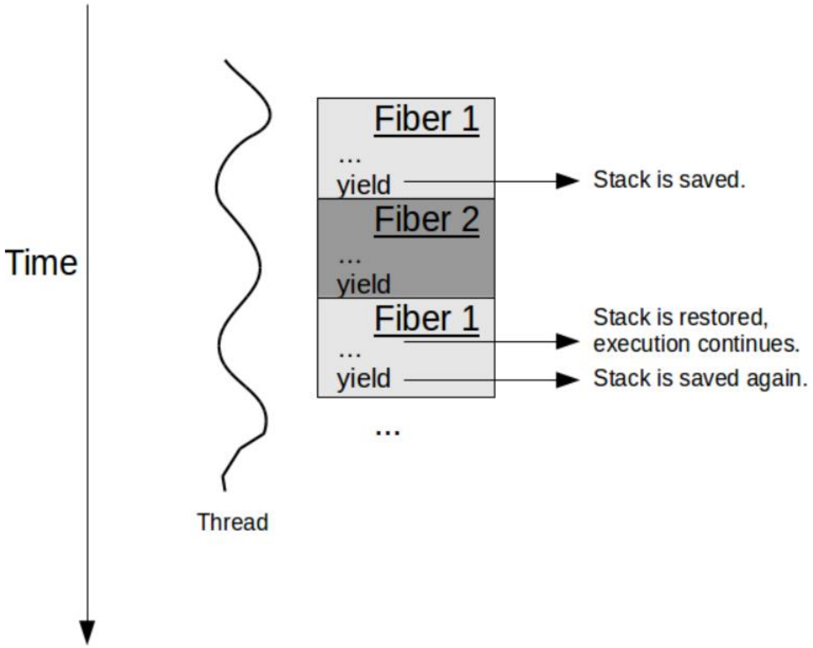
Scenario 2: 1 million GS updates from RPC (NCM)



- Generate ~1m GS updates in a giant vpc update
  - ~1m neighbors in the vpc
  - 20 ports are local
  - 989980 (due to TC tool limit, the maximum GS number is actually 990k)
- Start time: ACA's grpc thread pool (as grpc server) receives the GS update
- Process:
  - ACA grpc thread pool dispatches neighbor update tasks in worker thread pool
  - Worker thread pool picks up tasks to execute, translating vpc neighbor goal states (*GoalStateV2*) to ovs neighbor flows (*actions=strip_vlan,load:21->NXM_NX_TUN_ID[],set_field:10.10.10.10->tun_dst,output:100*)
  - Each task then call ovs driver to update neighbor rules (in OF flow-mod format)
  - Ovs driver thread pool picks up flow-mod messages and send to ovs via connection it maintains
  - Ovs receives flow-mods (and a barrier-request at the end of batch), then send a barrier-reply back
- End time: ACA receives ovs barrier-reply message and log time
- Latency (end_time – start_time): 13 seconds

Appendix I: generic fiber task scheduler



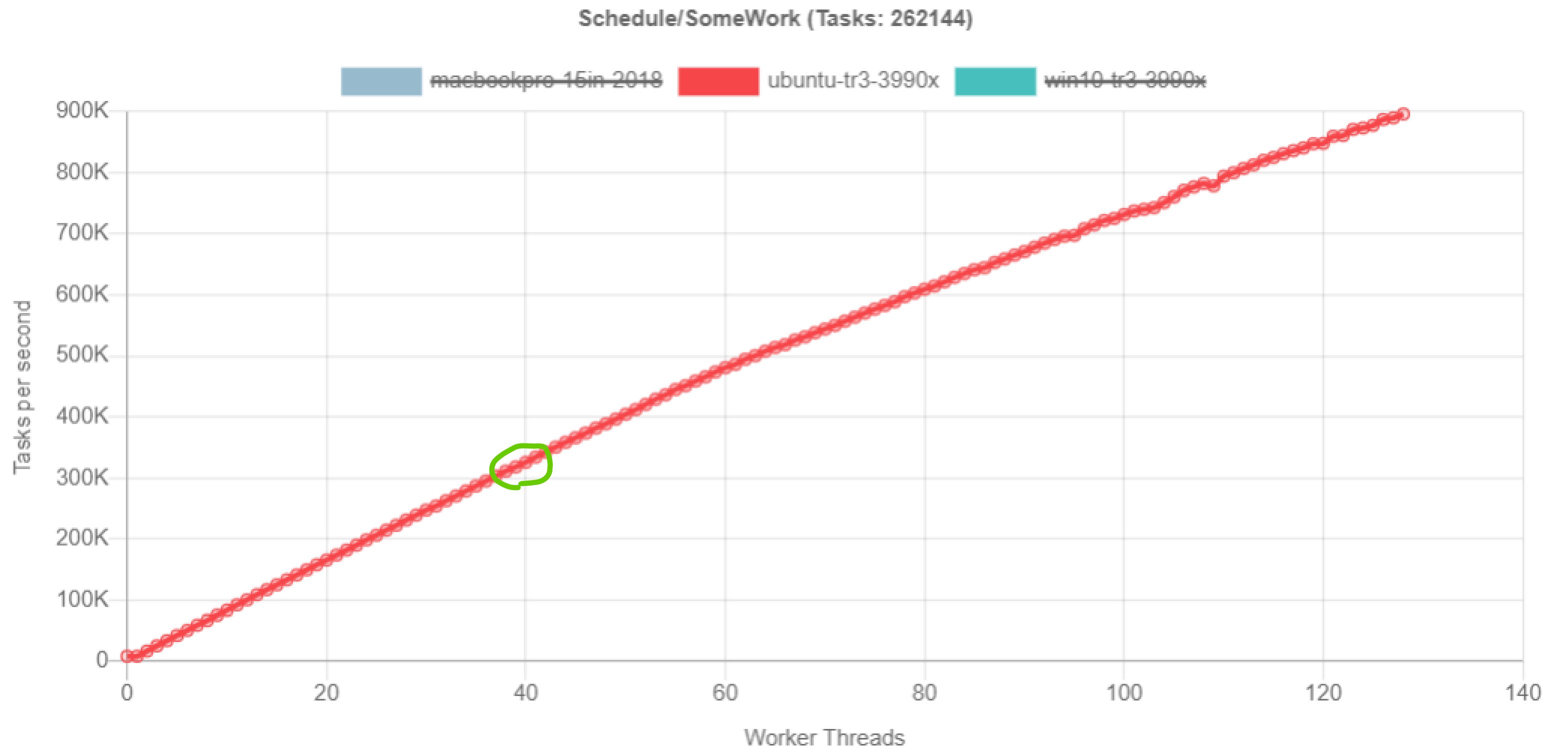


As you already know, fibers describe essentially the same concept as coroutines. The distinction, if there is any, is that coroutines are a language-level construct, a form of control flow, while fibers are a systems-level construct, viewed as threads that happen to not run in parallel.

A fiber stack is attached to a pthread to execute its internal logic, and when it goes to blocked state the library will explicitly call for a suspension and detach from the pthread rather than block the pthread from running something else. Once the monitor finds the suspended fiber stack is ready to be woken, it will find an available pthread (may not be the same pthread it was executed earlier) and resume all local context saved before the suspension.

Appendix II: Marl (fiber task scheduler that ACA leverages)

- https://google.github.io/marl/benchmarks/
- ACA 1/30 release uses 40 worker threads, and achieved 315K tasks per second with ARP regular workload, which approximately matches the official upper limit the framework could offer



Execution performance of calling `marl::schedule()` with a reasonable work load.

Appendix III: performance of popular frameworks

Server	Compiler	Optimization option	Cpu	Mem	Disk	Netcard(in/out)	Requests/sec
cppcms_based	clang++ 3.6.2	-O3	228% / 239% / 228%	9356 / 9416 / 9416	-	-	14252 / 16124 / 15820
asio_based	clang++ 3.6.2	-O3	300% / 305% / 303%	4368 / 4564 /4416	-	-	33069 / 34247 / 33360
libevent_based	clang++ 3.6.2	-O3	763% / 764% / 764%	5560 / 10M / 5520	-	-	113373 / 114072 / 113713
muduo_based	clang++ 3.6.2	-O3	650% / 694% / 658%	6272 / 6324 / 6312	-	-	303202 / 307204 / 305839

As a short summary, 300k qps is a reasonable expectation for tier 1 non-blocking I/O frameworks.

Some claim they can reach 400k or 500k, but with specific scenarios and run against an ideal test environment.

Appendix III cont. – others