Design considerations - Sync vs Async, Connection pooling vs Not

Inter component communication

Objective: Evaluate performance for Thrift calls at high throughputs

- **Throughput expected: Average case->30k rpm/machine; Worse case->100k rpm/machine**
- Approaches:
    - 1) Sync call
        - With Pool
        - Without Pool
    - 2) Async call
        - With Pool
        - Without Pool

Perf results:

Machine: MacBook Pro, 2.2 GHz Intel Core i7. PhysicalCpu:4. LogicalCpu:8(hyperthreading)

1) Sync client without Pool

- **Using TServiceClient**
- **Iterations=100; 10 ms sleep between each run**
- **Setup - TThreadedSelectorServer thread pool = 1000; Server response time = 500ms(sleep);**

Parallelism	Avg. Response time(per run)	95p Response time(per run)	Observation
100	587 = 82 + 505	678 = 177+501	

200	784 = 278 + 506	1097 = 592 + 505	Response time is well behaved and <510 for almost all requests. Client creation time increases with iterations.
500	1737 = 1032 + 705	2628 = 1633 + 995	Time taken to create client increases sharply to >1s for 20% runs. Significant overhead in response time.

`*Response time = Time taken to invoke parallel requests + Blocked waiting`

2) Async client without Pool

- **Using TAsyncClient**
- **Iterations=100; 10ms sleep between each run**
- **Setup - TThreadedSelectorServer thread pool = 1000; Server response time = 500(sleep);**

Time taken to create connections is significant, ranges from 300ms to 1300ms. Response time still remains close to 500ms for most, but goes to 600ms for a some request. Some outliers take several seconds to respond.

Parallelism	Avg. Response* time(per run)	95 percentile(per run)	Observation
100	598 = 97 + 501	651 = 150 + 501	*Time taken to create connections increases from 30 to 90+. Response time remains constant with 1ms overhead.*
200	842 = 341 + 501	1234 = 734 + 500	*Time taken to create connections 80 to 350+. A few calls take disproportionately high time(>700ms). Response time is still well behaved, reaches upto 530.*
500	2102 = 1601 + 501	7579 = 292 + 7287 6814 = 4470 + 2344	*Time taken to create connections is significant, ranges from 300ms to 1300ms. Response time still remains close to 500ms for most, but goes to 600ms for a some request. Some outliers take several seconds to respond.*
1000	2100 [for 16 iterations]	4478 [for 16 iterations]	***"java.net.BindException: Can't assign requested address" after 16 iterations. Need to check reason, presumbaly the server doesn't scale.***

`*Response time = Time taken to invoke parallel requests + Blocked waiting`

Observations:

- Caveat: A **new** client is created for each request, which means a new TCP connection is used.
    - There is a overhead of two RTT for TCP handshake. There are also the problems of TCP slow start, and HOL blocking.
        - Explore TCP Fast Open https://tools.ietf.org/html/draft-stenberg-httpbis-tcp-03
    - Need to reuse connections -> implement some sort of connection pooling
    - Tried multiple implementations, https://github.com/aloha-app/thrift-client-pool-java, https://github.com/wmz7year/Thrift-Connection-Pool, https://github.com/PhantomThief/thrift-pool-client, all of these only work with *TServiceClient* and not with the Thrift async client.
    - Explored Nifty - Could not find a connection pool for async client, need to explore more.
- Thrift over HTTP - Async HTTP Client can be used make async Thrift calls over HTTP. This way we can get both connection pooling and non blocking IO

3) Sync client with Pool(local)

- ### Using TServiceClient
  - **Iterations=100; 10ms sleep between each run**
  - **Setup - TThreadedSelectorServer thread pool = 1000; Server response time = 500(sleep).**
  - Thrift pooling library - https://github.com/aloha-app/thrift-client-pool-java

Parallelism	Avg. Response* time(per run)	95 percentile(per run)	Observation
100	522 = 1+ 521	542 = 0 + 542	
200	522 = 1 + 521	545 = 0 + 545	
500	671 = 1 + 670	755 = 1 + 754	Server response time increases at this level

****The 'Blocked waiting' component of the Response time is still high; for a usable solution we need to be able to get response with <10% overhead.

In our benchmarks, that would mean having a mean response time of ~550ms(with server sleep = 500ms), with reasonable dispersion. Currently, this value reaches upto 700ms(with some outliers) at higher parallelism.

- Since, the perf numbers were derived from a setup where the server and client were running on the same machine(Mac, with other applications running in the background), there might be some error/aberration.

  - [x] Repeat experiment with the server and client hosted on different machines connected over a network. Use two difference EC2 instances on the same VPC

  - Check if the 'Blocking wait' time is still high. If not, move to **4,** rinse and repeat.
    - Observing high overhead for Response time[Respone time = 798 = 1 + 797(blocking wait)], for a server sleep of 500ms. The client are server have latency of <1ms.
    - Unsure why the blocking wait time is so high.

3a) Sync client with Pool(stage)

Setup:

- Server: c4.xlarge, EC2 Region: ap-southeast-1, EC2 Availability Zone: ap-southeast-1a
- Client: c4.large, EC2 Region: ap-southeast-1, EC2 Availability Zone: ap-southeast-1b
- Network latency: rtt min/avg/max/mdev = 0.809/0.826/0.851/0.035 ms
- Other details same as **3**

Parallelism	Avg. Response* time(per run)	95 percentile(per run)	Observation
100	526	539	
200	693	739	Around 20% of the response(within each run) took >700ms
500	792	822	

4a) ASync client with Pool(local)

Setup:

- Iterations: 500; no sleep between iterations
- Server sleep = 500ms
- AsyncHttpClient as client, Netty based HTTP server(1100 threads)

Parallelism	Avg. Response* time(per run)	95 percentile(per run)	Observation
100	508	511	
200	511	515	
500	533	543	
1000	533	542	This is amazing!

4b) ASync client with Pool(stage)

- Server: c4.xlarge, EC2 Region: ap-southeast-1, EC2 Availability Zone: ap-southeast-1a
- Client: c4.large, EC2 Region: ap-southeast-1, EC2 Availability Zone: ap-southeast-1b
- Network latency: rtt min/avg/max/mdev = 0.809/0.826/0.851/0.035 ms
- Other details same as **4**

Parallelism	Mean Response time(per run)	95 percentile(per run)	Observation
100	505	515	
200	508	519	
500	513	522	
1000	523	535	

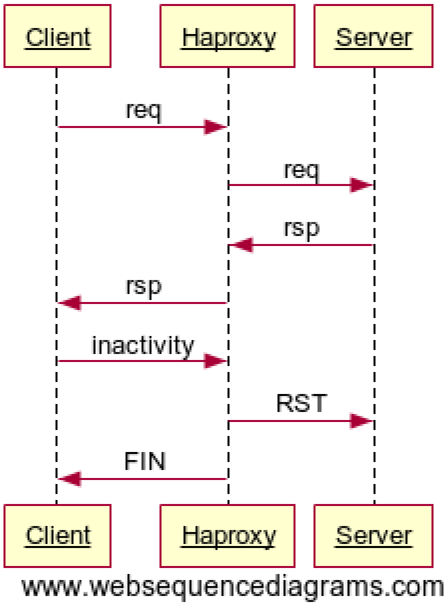
Observations:

Observations(*tentative, running list*):

- Async performs better than Sync in response time as their there is <1% overhead in getting response. Howoever, this is offset by the high cost of SocketChannel creation, especially at higher parallelism.
- Socket/SocketChannel creation time adds a significant overhead, and increases with number of runs. This seriously hinders the parallelism, and overshadows the benefits of async
- A single selector thread is used in the Thrift async client, which is used for executing the callback. Can be a potential bottleneck.
- AsyncHttpClient idioms - https://gitlab.corp.olacabs.com/project-os/optimal-assignment/commit/feb159ba226742765df72bbc0596a9a07a1de5a5

Learnings from Haproxy

Haproxy - client timeout kicks in



Haproxy - client closes connection