RxNetty vs Tomcat
Performance Results

Brendan Gregg; Performance and Reliability Engineering
Nitesh Kant, Ben Christensen; Edge Engineering
updated: Apr 2015
Results based on

- The “Hello Netflix” benchmark (wsperflab)
- Tomcat
- RxNetty
- physical PC
  - Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz: 4 cores, 1 thread per core
- OpenJDK 8
  - with frame pointer patch
- Plus testing in other environments
Incoming HTTP request 

Parallel requests to ws-backend-mock for A, B, C, D, E with different latencies.

Aggregate into single JSON response
RxNetty vs Tomcat performance

In a variety of tests, RxNetty has been faster than Tomcat. This study covers:

1. What specifically is faster?
2. By how much?
3. Why?
1. What specifically is faster?
1. What specifically is faster?

- **CPU consumption per request**
  - RxNetty consumes less CPU than Tomcat
  - This also means that a given server (with fixed CPU capacity) can deliver a higher maximum rate of requests per second

- **Latency under load**
  - Under high load, RxNetty has a lower latency distribution than Tomcat
2. By how much?
2. By how much?

The following 5 graphs show performance vs load (clients)

1. CPU consumption per request
2. CPU resource usage vs load
3. Request rate
4. Request average latency
5. Request maximum latency

Bear in mind these results are for this environment, and this workload
2.1. CPU Consumption Per Request

- RxNetty has generally lower CPU consumption per request (over 40% lower)
- RxNetty keeps getting faster under load, whereas Tomcat keeps getting slower
2.2. CPU Resource Usage vs Load

- Load testing drove the server’s CPUs to near 100% for both frameworks.
2.3. Request Rate

- RxNetty achieved a 46% higher request rate
- This is mostly due to the lower CPU consumption per request
2.4. Request Average Latency

- Average latency increases past the req/sec knee point (when CPU begins to be saturated)
- RxNetty’s latency breakdown happens with much higher load
2.5. Request Maximum Latency

- The degradation in maximum latency for Tomcat is much more severe
3. Why?
3. Why?

1. CPU consumption per request
   - RxNetty is lower due to its framework code and lower object allocation rate, which in turn reduces GC overheads
   - RxNetty also trends lower due to its event loop architecture, which reduces thread migrations under load, which improves CPU cache warmth and memory locality, which improves CPU Instructions Per Cycle (IPC), which lowers CPU cycle consumption per request

2. Lower latencies under load
   - Tomcat has higher latencies under load due to its thread pool architecture, which involves thread pool locks (and lock contention) and thread migrations to service load
3.1. CPU Consumption Per Request

Studied using:
1. Kernel CPU flame graphs
2. User CPU flame graphs
3. Migration rates
4. Last Level Cache (LLC) Loads & IPC
5. IPC & CPU per request
3.1.1. Kernel CPU Flame Graphs
3.1.1. Kernel CPU Time Differences

CPU system time delta per request: 0.07 ms
- Tomcat futex(), for thread pool management (0.05 ms)
- Tomcat poll() vs RxNetty epoll() (0.02 ms extra)
3.1.2. User CPU Flame Graphs
3.1.2. User CPU Time Differences

CPU user time delta per request: 0.14 ms

Differences include:
- Extra GC time in Tomcat
- Framework code differences
- Socket read library
- Tomcat thread pool calls
3.1.3. Thread Migrations

- As load increases, RxNetty begins to experience lower thread migrations.
- There is enough queued work for event loop threads to keep servicing requests without switching.
3.1.4. LLC Loads & IPC

- ... The reduction in thread migrations keeps threads on-CPU, which keeps caches warm, reducing LLC loads, and improving IPC
3.1.5. IPC & CPU Per Request

- ... A higher IPC leads to lower CPU usage per request
3.2. Lower Latencies Under Load

Studied using:
1. Migration rates (previous graph)
2. Context-switch flame graphs
3. Chain graphs
3.2.2. Context Switch Flame Graphs

- These identify the cause of context switches, and blocking events.
  - They do not quantify the magnitude of off-CPU time; these are for identification of targets for further study
- Tomcat has additional futex context switches from thread pool management
Context Switch Flame Graph: Tomcat

ThreadPool
Executor locks
3.2.3. Chain Graphs

- These quantify the magnitude of off-CPU (blocking) time, and show the chain of wakeup stacks that the blocked thread was waiting on
  - x-axis: blocked time
  - y-axis: blocked stack, then wakeup stacks
Normal blocking path:
server thread waits on backend network I/O

Tomcat blocked on itself: thread pool locks
Reasoning

- On a system with more CPUs (than 4), Tomcat will perform even worse, due to the earlier effects.
- For applications which consume more CPU, the benefits of an architecture change diminish.
Under light load, both have similar performance, with RxNetty using less CPU. With increased load, RxNetty begins to migrate less, improving IPC, and CPU usage per request. At high load, RxNetty delivers a higher req rate, with a lower latency distribution due to its architecture.