Netty @ Apple

Massive Scale Deployment / Connectivity
Norman Maurer

- Senior Software Engineer @ Apple
- Core Developer of Netty
- Formerly worked @ Red Hat as Netty Project Lead (internal Red Hat)
- Author of Netty in Action (Published by Manning)
- Apache Software Foundation
- Eclipse Foundation

This is not a contribution
Massive Scale

This is not a contribution
Massive Scale

What does “Massive Scale” mean…

- Instances of Netty based Services in Production: 400,000+
- Data / Day: 10s of PetaBytes
- Requests / Second: 10s of Millions
- Versions: 3.x (migrating to 4.x), 4.x

This is not a contribution
Part of the OSS Community

- Contributing back to the Community
- 250+ commits from Apple Engineers in 1 year

This is not a contribution
Using an Apple Service?
Chances are good Netty is involved somehow.
Areas of importance

• Native Transport
• TCP / UDP / Domain Sockets
• PooledByteBufAllocator
• OpenSslEngine
• ChannelPool
• Build-in codecs + custom codecs for different protocols
With Scale comes Pain
JDK NIO

... some pains
Some of the pains

- Selector.selectedKeys() produces too much garbage
- NIO implementation uses synchronized everywhere!
- Not optimized for typical deployment environment (support common denominator of all environments)
- Internal copying of heap buffers to direct buffers
JNI to the rescue

- Optimized transport for Linux only
- Supports Linux specific features
- Directly operate on pointers for buffers
- Synchronization optimized for Netty’s Thread-Model
Native Transport

**epoll based high-performance transport**

- Less GC pressure due less Objects
- Advanced features
  - SO_REUSEPORT
  - TCP_CORK,
  - TCP_NOTSENT_LOWAT
  - TCP_FASTOPEN
  - TCP_INFO
- LT and ET
- Unix Domain Sockets

```java
NIO Transport
Bootstrap bootstrap = new Bootstrap().group(new NioEventLoopGroup());
bootstrap.channel(NioSocketChannel.class);

Native Transport
Bootstrap bootstrap = new Bootstrap().group(new EpollEventLoopGroup());
bootstrap.channel(EpollSocketChannel.class);`
Buffers
JDK ByteBuffer

- Direct buffers are free’ed by GC
  - Not run frequently enough
  - May trigger GC

- Hard to use due not separate indices
Buffers

- Direct buffers == expensive
- Heap buffers == cheap (but not for free*)
- Fragmentation

*byte[] needs to be zero-out by the JVM!
Buffers - Memory fragmentation

- Waste memory
- May trigger GC due lack of coalesced free memory

Can’t insert int here as we need 4 continuous slots
Allocation times

![Graph showing allocation times for different byte sizes and allocation types. The x-axis represents byte sizes (0, 256, 1024, 4096, 16384, 65536), and the y-axis represents nanoseconds. The graph compares Unpooled Heap, Pooled Heap, Unpooled Direct, and Pooled Direct allocation times.]
PooledByteBufAllocator

- Based on jemalloc paper (3.x)
- ThreadLocal caches for lock-free allocation in most cases #808
- Synchronize per Arena that holds the different chunks of memory
- Different size classes
- Reduce fragmentation
ThreadLocal caches

- Able to enable / disable ThreadLocal caches
- Fine tuning of Caches can make a big difference
- Best effect if number of allocating Threads are low.
- Using ThreadLocal + MPSC queue #3833

<table>
<thead>
<tr>
<th>Title</th>
<th>Contention Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>0</td>
</tr>
<tr>
<td>No Cache</td>
<td>4000</td>
</tr>
</tbody>
</table>

This is not a contribution
JDK SSL Performance

.... it’s slow!
Why handle SSL directly?

- Secure communication between services
- Used for HTTP2 / SPDY negotiation
- Advanced verification of Certificates

Unfortunately JDK's SSLEngine implementation is very slow :(
### HTTPS Benchmark

**JDK SSLEngine implementation**

<table>
<thead>
<tr>
<th>Response</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1 200 OK</td>
<td>Running 2m test @ <a href="https://xxx:8080/plaintext">https://xxx:8080/plaintext</a></td>
</tr>
<tr>
<td>Content-Length: 15</td>
<td>16 threads and 256 connections</td>
</tr>
<tr>
<td>Content-Type: text/plain; charset=UTF-8</td>
<td>Thread Stats Avg Stdev Max +/- Stdev</td>
</tr>
<tr>
<td>Server: Netty.io</td>
<td>Latency 553.70ms 81.74ms 1.43s 80.22%</td>
</tr>
<tr>
<td>Date: Wed, 17 Apr 2013 12:00:00 GMT</td>
<td>Req/Sec 7.41k 595.69 8.90k 63.93%</td>
</tr>
<tr>
<td>Hello, World!</td>
<td>14026376 requests in 2.00m, 1.89GB read</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>curl -H 'Host: localhost' -H 'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,<em>/</em>;q=0.8' -H 'Connection: keep-alive' -d 120 -c 256 -t 16 -s scripts/pipeline-many.lua <a href="https://xxx:8080/plaintext">https://xxx:8080/plaintext</a></td>
</tr>
</tbody>
</table>

Requests/sec: **116883.21**

Transfer/sec: **16.16MB**
HTTPS Benchmark
JDK SSLEngine implementation

- Unable to fully utilize all cores
- SSLEngine API limiting in some cases
  - SSLEngine.unwrap(…) can only take one ByteBuffer as src
JNI based SSLEngine

... to the rescue

This is not a contribution
JNI based SSLEngine

...one to rule them all

- Supports OpenSSL, LibreSSL and BoringSSL
- Based on Apache Tomcat Native
- Was part of Finagle but contributed to Netty in 2014

This is not a contribution
HTTPS Benchmark
OpenSSL SSLEngine implementation

Response

HTTP/1.1 200 OK
Content-Length: 15
Content-Type: text/plain; charset=UTF-8
Server: Netty.io
Date: Wed, 17 Apr 2013 12:00:00 GMT

Hello, World!

Result

Running 2m test @ https://xxx:8080/plaintext
16 threads and 256 connections

Thread Stats    Avg      Stdev     Max   +/- Stdev
Latency       131.16ms   28.24ms 857.07ms   96.89%
 Req/Sec       31.74k     3.14k   35.75k    84.41%

60127756 requests in 2.00m, 8.12GB read
Socket errors: connect 0, read 0, write 0, timeout 52

Requests/sec: 501120.56
Transfer/sec:  69.30MB

Benchmark

/wrk -H 'Host: localhost' -H 'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8' -H 'Connection: keep-alive' -d 120 -c 256 -t 16 -s scripts/pipeline-many.lua https://xxx:8080/plaintext

This is not a contribution
HTTPS Benchmark
OpenSSL SSLEngine implementation

- All cores utilized!
- Makes use of native code provided by OpenSSL
- Low object creation
- Drop in replacement*

*supported on Linux, OSX and Windows
Optimizations made

- Added client support: #7, #11, #3270, #3277, #3279
- Added support for Auth: #10, #3276
- GC-Pressure caused by heavy object creation: #8, #3280, #3648
- Too many JNI calls: #3289
- Proper SSLSession implementation: #9, #16, #17, #20, #3283, #3286, #3288
- ALPN support #3481
- Only do priming read if there is no space in dsts buffers #3958

This is not a contribution
Thread Model

- Easier to reason about
- Less worry about concurrency
- Easier to maintain
- Clear execution order
public class ProxyHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        final Channel inboundChannel = ctx.channel();
        Bootstrap b = new Bootstrap();
        b.group(inboundChannel.eventLoop());
        ctx.channel().config().setAutoRead(false);
        ChannelFuture f = b.connect(remoteHost, remotePort);
        f.addListener(f -> {
            if (f.isSuccess()) {
                ctx.channel().config().setAutoRead(true);
            } else {
                ...}
        });
    }
}
Backpressure

- Slow peers due slow connection
- Risk of writing too fast
- Backoff writing and reading
Memory Usage

- Handling a lot of concurrent connections
- Need to safe memory to reduce heap sizes
  - Use Atomic*FieldUpdater
  - Lazy init fields

This is not a contribution
Connection Pooling

- Having an extensible connection pool is important #3607
- flexible / extensible implementation

This is not a contribution
We are hiring!