Netty
One Framework to rule them all....
Leading the Netty Project
Apache Cassandra MVP 2016 - 2017
Author of Netty in Action
Apache Software Foundation
Eclipse Foundation
A long journey
Some background

- Netty 3.0.0.GA released in 2008
- Netty 4.0.0.Final released in 2013
- Netty 4.1.0.Final released in 2016
- one of the most used Network Framework for the JVM
- founded by Trustin Lee <3
- JBoss Project first, then independent
- very vibrant community
Netty 3.x

- too much garbage
- too many memory copies
- no good memory pool included
- not optimized for Linux based OS
- threading model not easy to reason about

Still it worked great! ….. Kind of at least.
Netty 4.x now!

- create less garbage, less GC
- optimized for Linux based OS + Linux only features
- high performance buffer pool based on jemalloc paper
- well defined, easy to use threading model

And there is more too come....
Inbound events $\rightarrow$ ChannelInboundHandler

Outbound events $\rightarrow$ ChannelOutboundHandler
ChannelPipeline

• Interceptor pattern
• allows to add building-blocks (*ChannelHandler*) on the fly that transforms data or react on events.

Combine handlers as UNIX commands via pipes

```
$ echo "Netty is slow...." | sed -e 's/slow/fast/' | cat
Netty is fast....
```
Too much garbage

Run collector ….run!
Reduce Garbage

- eliminate GC by replace event objects with direct method invocations
- light-weight object pool for heavily allocated objects (like `ByteBuf` instances)

Allocating an Object is often not the problem, collecting it is 😭
JNI to the rescue

- optimized transport for Linux only
- supports Linux specific features
- directly operate on pointers for buffers
- synchronization optimized for Netty’s threading model
Native Transport

*epoll based high-performance transport*

**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);

- 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
Buffers
Performance vs Complexity
ByteBuf

- ByteBufs are reference counted (huh!?!?)
- pooling is used by default
- provide LeakDetector which helps detecting ByteBuf leaks
- direct memory are used by default
- provide special abstractions to iterate over bytes to reduce branching / range-checks
- all buffers are dynamic and can grow

Writing Java as it is C ?!
Buffer Pooling
Allocations are expensive
Allocation times

- **Unpooled Heap**
- **Pooled Heap**
- **Unpooled Direct**
- **Pooled Direct**

The graph shows allocation times in NanoSeconds for different byte sizes.
PooledByteBufAllocator

- based on jemalloc paper (3.x)
- ThreadLocal caches for lock-free allocation
- synchronize per Arena that holds the different chunks of memory
- different size classes
- reduce fragmentation
Threading Model

Writing multi-threaded applications is hard....
Threading-Model

- all events / operations are done by the IO-Thread!
- eliminates the need of synchronization completely (as long as the handler is not shared!)
- writing single-threaded code FTW
Write Semantics

syscalls are expensive...
Write Semantics

- `Channel.write(…)` will only put messages in the `ChannelOutboundBuffer` once processed.
- `Channel.flush()` will flush everything in the `ChannelOutboundBuffer` and so call `writev(…)`. 
Read Semantics
Fine grained control FTW
Read Semantics

- `ChannelConfig.setAutoRead(boolean)` to the rescue.
- `ChannelConfig.setMaxMessagesPerRead(int)` allows to limit max number of messages to read.
- `Channel.read()` allows to explicit trigger a read.
- `RecvByteBufAllocator` gives even more flexibility

```java
while (i < messagesPerRead) {
    read(...);
}
```
IO - Threads

Never-ever block the IO-Thread!
EventLoop(Group)

- IO *Thread* abstracted as *EventLoop*
- easily share the same *EventLoop* between Server and Client
- be able to explicitly use same *EventLoop* for accepted connection and outbound connection (win for proxy applications!)

**Bonus:** *EventLoop* is also a *ScheduledExecutorService*
Work outside the IO-Thread
sometimes you need to block
EventExecutor(Group)

- part of the core itself
- adding ChannelHandler with an EventExecutorGroup will get the job done
- different EventExecutorGroup implementations for serial / non-serial executions.
- supports moving work to other EventLoop
JNI based SSL Engine

... to the rescue
SSLEngine implementations

Requests / Sec

OpenSslEngine
SSLEngineImpl

Transfer(MB) / Sec

OpenSslEngine
SSLEngineImpl
SSLEngine implementations

OpenSslEngine

SSLEngineImpl

VS
OpenSslEngine

- drop in replacement for JDK SSLEngine (SSLEngineImpl)
- gives you up to 6 x performance
- less memory usage
- less GC

SslContextBuilder.forServer().sslProvider(SslProvider.OpenSsl);
Direct memory management

- The whole idea of managing direct memory with via the Garbage-Collector is fundamentally broken.
- Static synchronized in allocation and deallocation methods of direct memory.
- There is also `Thread.sleep(100)` and `System.gc()` ?!?
Memory Layout - ENOCONTROL

- no easy way to control over memory layout (all these hacks ….)
- false-sharing is a real issue on own data-structures
- @Contended does not help at all in practice

Gimme more control now!
• nasty “hacks” needed to be able to get good performance
• includes things like writing structs directly via `sun.misc.Unsafe` (no joke!)
• calling from JNI into Java methods is SUPER-expensive
NIO / IO and others

- NIO.2 no real improvement over NIO
- too much garbage produced and so GC overhead
- `ByteBuffer` API is not user-friendly (flip all the things!)
- `IOException` / `ConnectException` are too generic and not useful
- creating `String` from `byte[]` / `char[]` not possible without memory copy
- `java.util.concurrent.Future` was (and still is) a disaster
Get my book…

Ka-ching!
Questions?
Thanks!