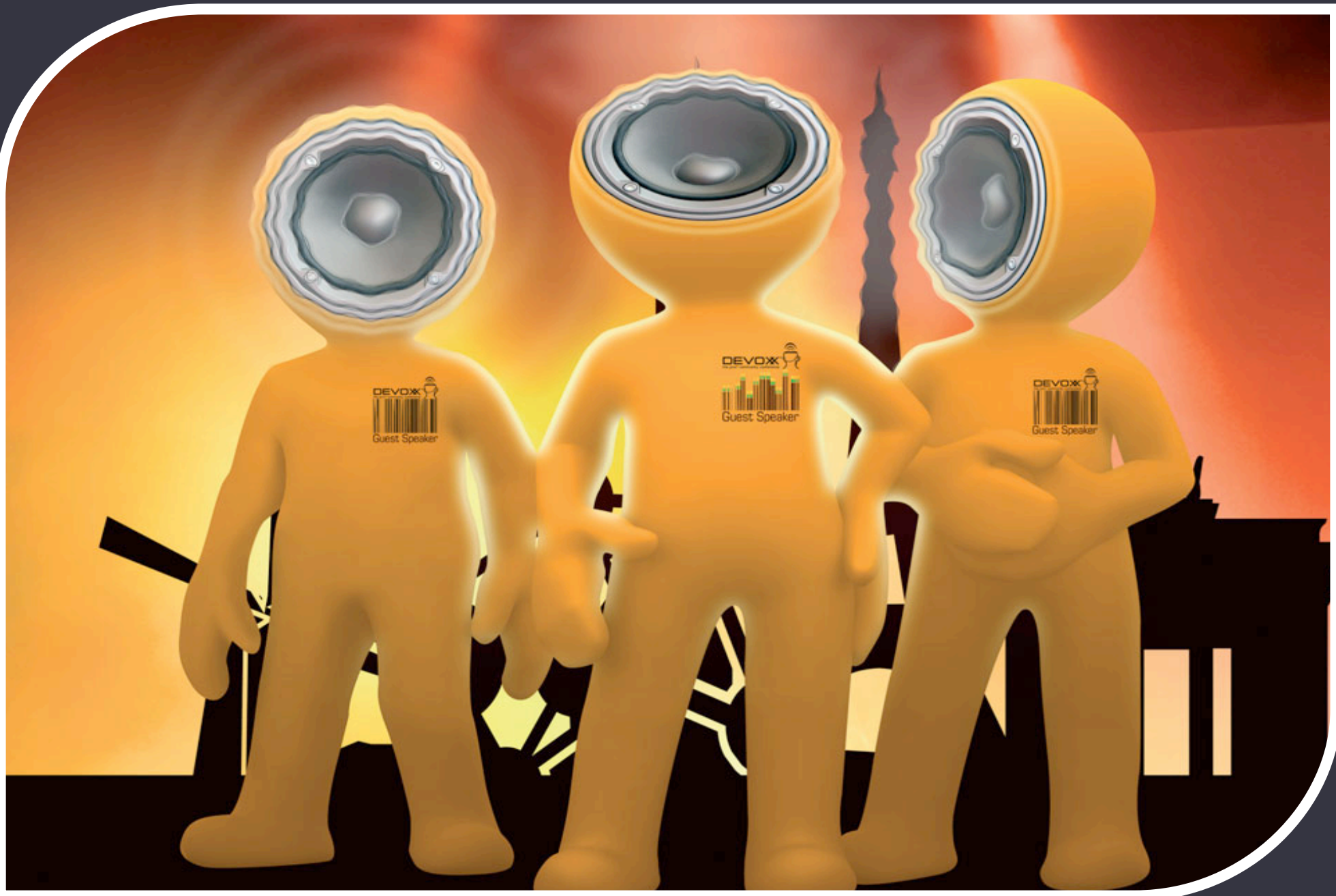

DEVOXX
Guest Speaker
DEVOXX
Guest Speaker
DEVOXX
Guest Speaker

# Introduction to Asynchronous I/O using the Grizzly Framework

Jeanfrancois Arcand
Senior Staff Engineer
Sun Microsystems

# Goal

The Grizzly framework has been designed to help developers to take advantage of the Java™ NIO API and the upcoming JDK 7 NIO.2 (Asynchronous I/O).

This session will introduce NIO.2 API and Concepts using the Grizzly Framework and will compare the performance of NIO with AIO using an asynchronous http server implementation.

# Qui suis-je?

- Jeanfrancois is the founder of Grizzly, an OSS NIO Framework used in many applications and the core of GlassFish, Sun's application server.
- Jeanfrancois writes technical articles for CometDaily and JavaLobby and do the monkey @ JavaOne, FISL, JavaPolis, AjaxWorld, ApacheCon, etc.
- Authors of Tricks and Tips with NIO.
- Having fun on
  - http://weblogs.java.net/jfarcand
  - http://twitter.com/project_grizzly

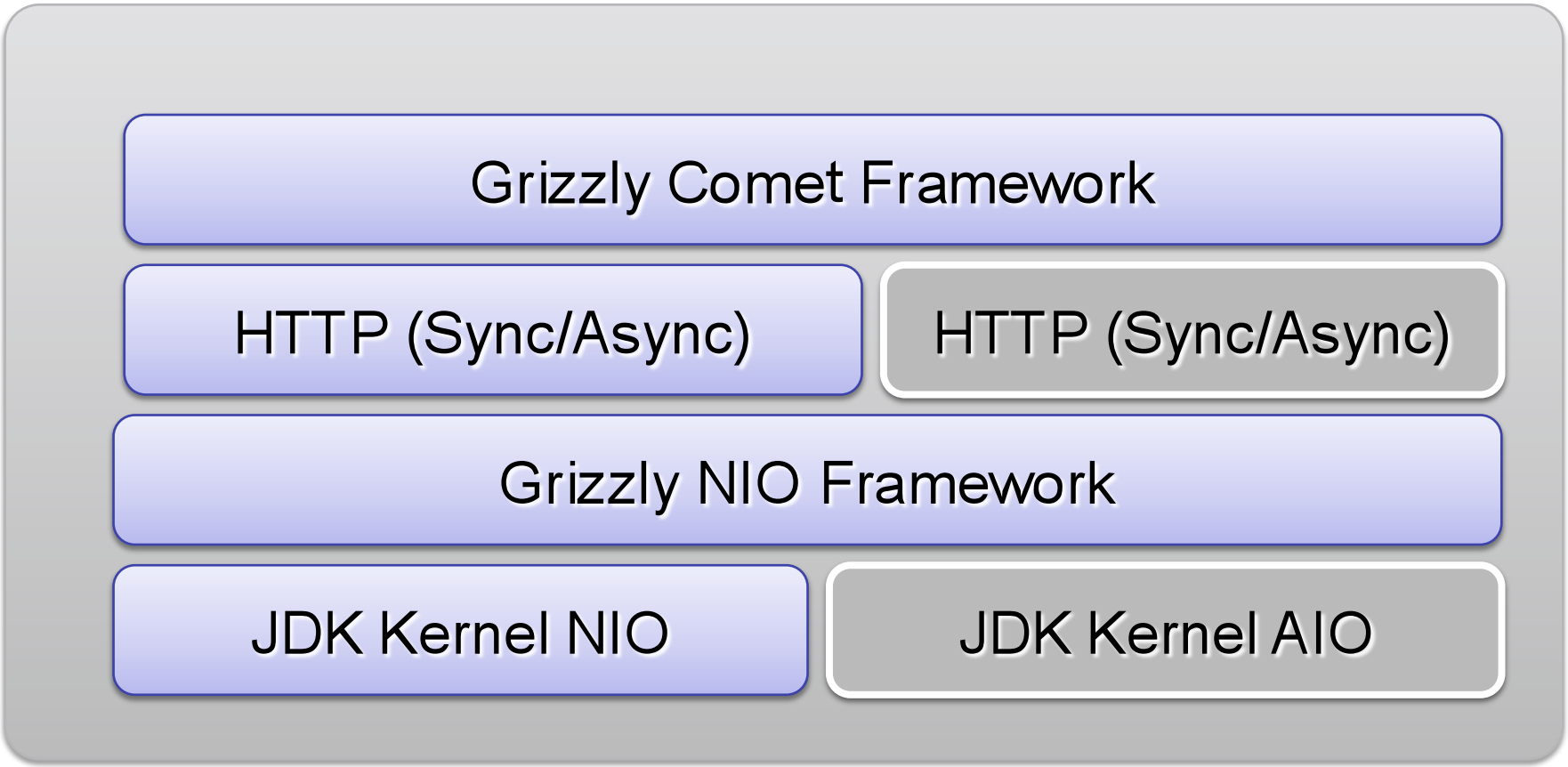
# Agenda

- What is Grizzly
- Introduction to Asynchronous I/O
- The Frightening Thread Pool
- The Evil AsynchronousSocketChannel.read()
- The Mysterious AsynchronousSocketChannel.write()
- Damned ByteBuffer
- Finally I might start using File based Channel…new File I/O system!
- Benchmarking Grizzly against Grizzly
- Conclusion

# What is Project Grizzly

- Open Source Project on java.net,
- Open Sourced under CDDL/LGPL license (soon Apache2)
- Very open community policy.
  - All project communications are done on Grizzly mailing list. No internal, off mailing list conversations.
  - Project meetings open to anyone (public conference call).
- Project decisions are made by project member votes.
- Sun and **non Sun commiters**
- Latest Release: 1.9.0 with AIO Support.

# Grizzly Architecture



# Introduction to AIO

- NIO.1 -> Get events through a Selector when there is some I/O ready to be processed, like read and write operations. As soon as you get notified that an event is ready to be processed, execute an operation:

```
If (key.isReadable(){
    channel.read(byteBuffer);
}
```

- Asynchronous I/O send you a notification when the I/O is **completed**. You get the notification ONLY when the operation has completed (or failed).

# CompletionHandler

- With AIO, you wait for I/O operation using we a CompletionHandler:

completed(...): invoked when the I/O operation completes successfully.

failed(...): invoked if the I/O operations fails (like when the remote client close the connection).

cancelled(...): invoked when the I/O operation is cancelled by invoking the cancel method on a Future.

- As an example, you specify a completion handler when you want to execute an asynchronous read.

# Future

- Almost all I/O operations returns a Future which can be used to monitor the I/O transaction and be used to block for the I/O transaction to complete/times out.

```
Future f = asynchronousSocketChannel.connect(…);

// Wait for the connect operation to complete.

f.get(30,TimeUnit.SECONDS);
```

# The Frightening Thread Pool

- With AIO, you can configure the thread pool (ExecutorService) used by both the AIO kernel and your application

  AsynchronousChannelGroup.withFixedThreadPool

  (ExecutorService, maxCorePoolThreads)

  AsynchronousChannelGroup.withCachedThreadPool

  (ExecutorService, initialSize)

  … or use the preconfigured/built in Thread Pool that comes by default…

# FixedThreadPool

- An asynchronous channel group associated with a fixed thread pool of size N creates N threads that are waiting for already processed I/O events.
- The kernel dispatch event directly to those threads:
  - Thread first complete the I/O operation (like filling a ByteBuffer during a read operation).
  - Next invoke the CompletionHandler.completed() that consumes the result.
  - When the CompletionHandler terminates normally then the thread returns to the thread pool and wait on a next event.

# FixedThreadPool

- If the completion handler terminates due to an uncaught error or runtime exception, the thread is returned to the pool.
- For those cases, the thread is allowed to terminate
- The reason the thread is allowed to terminate is so that the thread (or thread group) uncaught exception handler is executed.

# Booo…my application freezes!

- What about if all threads "dead lock" inside a CompletionHandler?
    - Bang! your entire application can hangs until one thread becomes free to execute again.
    - The kernel is no longer able to EXECUTE anything!
- Hence this is critically important CompletionHandler complete in a timely manner and avoid blocking.
- If all completion handlers are blocked, any new event will be queued until one thread is 'delivered' from the lock.
- So try to avoid blocking operations inside a completion handler.

# CachedThreadPool

- An asynchronous channel group associated with a cached thread pool submits events to the thread pool that simply invoke the user's completion handler.
- Internal kernel's I/O operations are handled by one or more internal threads that are not visible to the user application.
- That means you have one hidden thread pool that dispatch events to a cached thread pool, which in turn invokes completion handler
- Wait! you just win a price: a thread's context switch for free!!

# CachedThreadPool

- Probability of suffering the hangs problem as with the FixedThreadPool is lower.
- Still might grows infinitely (those infinite thread pool should have never existed anyway!).
- At least you guarantee that the kernel will be able to complete its I/O operations (like reading bytes).
- Oups…CachedThreadPool must support unbounded queueing to works properly (grrr not sure I like that!!!).
- So you can possibly lock all the threads and feed the queue forever until OOM happens. Great!
- Of course, nobody will ever do that!

# Build-in Kernel Thread Pool

- Hybrid of the above configurations:
  - Cached thread pool that creates threads on demand
  - N threads that dequeue events and dispatch directly to CompletionHandler
- N defaults to the number of hardware threads.
- In addition to N threads, there is one additional internal thread that dequeues events and submits tasks to the thread pool to invoke completion handlers.

# Some monster's code

AsynchronousChannelGroup:

com.sun.grizzly.aio.AIOHandlerRunner

CompletionHandler

com.sun.grizzly.aio.TCPAIOHandler

AsynchronousServerSocketChannel

com.sun.grizzly.aio.TCPAIOHandler

# The evil AsynchronousSocketChannel.read()

- Once a connection has been accepted, it is now time to read some bytes:

```
AsynchronousSocketChannel.read(ByteBuffer b,
                               Attachement a,
                               CompletionHandler c);
```

## Hey Hey → You see the evil, right?

Who remember when I was scared by the SelectionKey.attach()?

# The evil AsynchronousSocketChannel.read()

- Trouble trouble trouble:
  - Let's say you get 10 000 accepted connections
  - Hence 10 000 ByteBuffer created, and the read operations get invoked
  - Now we are waiting, waiting, waiting, waiting for the remote client(s) to send us bytes (slow clients/network)
  - Another 10 000 requests comes in, and we are again creating 10 000 ByteBuffer and invoke the read() operations.
- BOOM OOM!

# The evil AsynchronousSocketChannel.read()

- Let's not be too negative here. So far we have tested with more than 20 000 clients without any issues
- But this is still something you have to keep in mind!!
- Might want to throttle the read() operation to avoid the creation of too many ByteBuffer
- I strongly recommend the use of a ByteBuffer pool, specially if you are using Heap ByteBuffer (more on this later). Get a ByteBuffer before invoking the read() method, and return it to the pool once the read operations complete.

# Blocking AsynchronousSocketChannel.read()

- Hein? Blocking?
- When invoking the read operation, the returned value is a Future:

```
Future readOp = AsynchronousSocketChannel.read(…);

readOp.get(30, TimeUnit.SECONDS);
```

- The Thread will blocks until the read operation complete or times out.
- Be careful as you might lock your ThreadPool (specially the fixedThreadPool)

# Some monster's code

AsynchronousSocketChannel.read()

com.sun.grizzly.aio.AIOContextTask

Blocking read()

com.sun.grizzly.aio.util.InputReader

# The mysterious AsynchronousSocketChannel.write(..)

- Now let's execute some write operations:

**AsynchronousSocketChannel.write(ByteBuffer b,**

**Attachement a,**

**CompletionHandler c);**

- Wait wait wait. Since we are asynchronous, invoking write(..) will not block, so the calling thread can continue its execution.
- What's happen when the calling thread invokes the write method again and the CompletionHandler has not yet been invoked by the previous write call?

# The mysterious AsynchronousSocketChannel.write(..)

- Aille!! You get a WritePendingException
- Hence when invoking the write operation, make sure the CompletionHandler.complete() has been invoked before initiating another write.
- Better, store ByteBuffer inside a queue and execute write operations only when the previous one has completed (will show code soon)
- As for read, I strongly recommend the use of a ByteBuffer pool for executing write operations. Get one before writing, put it back to the pool after.

# Some monster's code

AsynchronousSocketChannel.write()

com.sun.grizzly.aio.OutputWriter

Blocking write()

com.sun.grizzly.aio.OutputWriter

# Damned ByteBuffer!

- If you are using Heap ByteBuffer, be aware the kernel will copy the bytes into a direct ByteBuffer during every write operation:

Free byte copy ☺

- Direct ByteBuffer performance have significantly improved with JDK 7, so use them directly.
- Scattered ByteBuffer write operations still offer you free copy, using direct ByteBuffer or not!

# The cool AsynchronousFileChannel.open()

- Before, the nightmare:

```
File f = new File();

FileOutputStream fis = new FileOutputStream(f);

FileChannel fc = fis.getChannel();

fc.write(…);
```

**…… typing so much lines hurts ☺**

Now let's execute some write operations:

```
AsynchronousFileChannel open(Path file, Set<? extends OpenOption> options,
                             ExecutorService executor, FileAttribute<?>... attrs);

afc.write(…);
```

# New File I/O system API

- New packages (More info can be found by downloading JavaOne'08)
  - java.nio.file, java.nio.file.attribute
- Main classes:
  - FileSystem:
    - Interface to file system
    - Factory for objects to access files and other objects in file system
  - FileRef
    - Reference to file or directory
    - Defines methods to operate on file or directory
  - Path
    - A FileRef that locates a file by a system dependent path
    - Created by FileSystem by converting path string or URI
  - FileStore
    - Underlying storage pool, device, partition...

# Benchmarking Grizzly against Grizzly

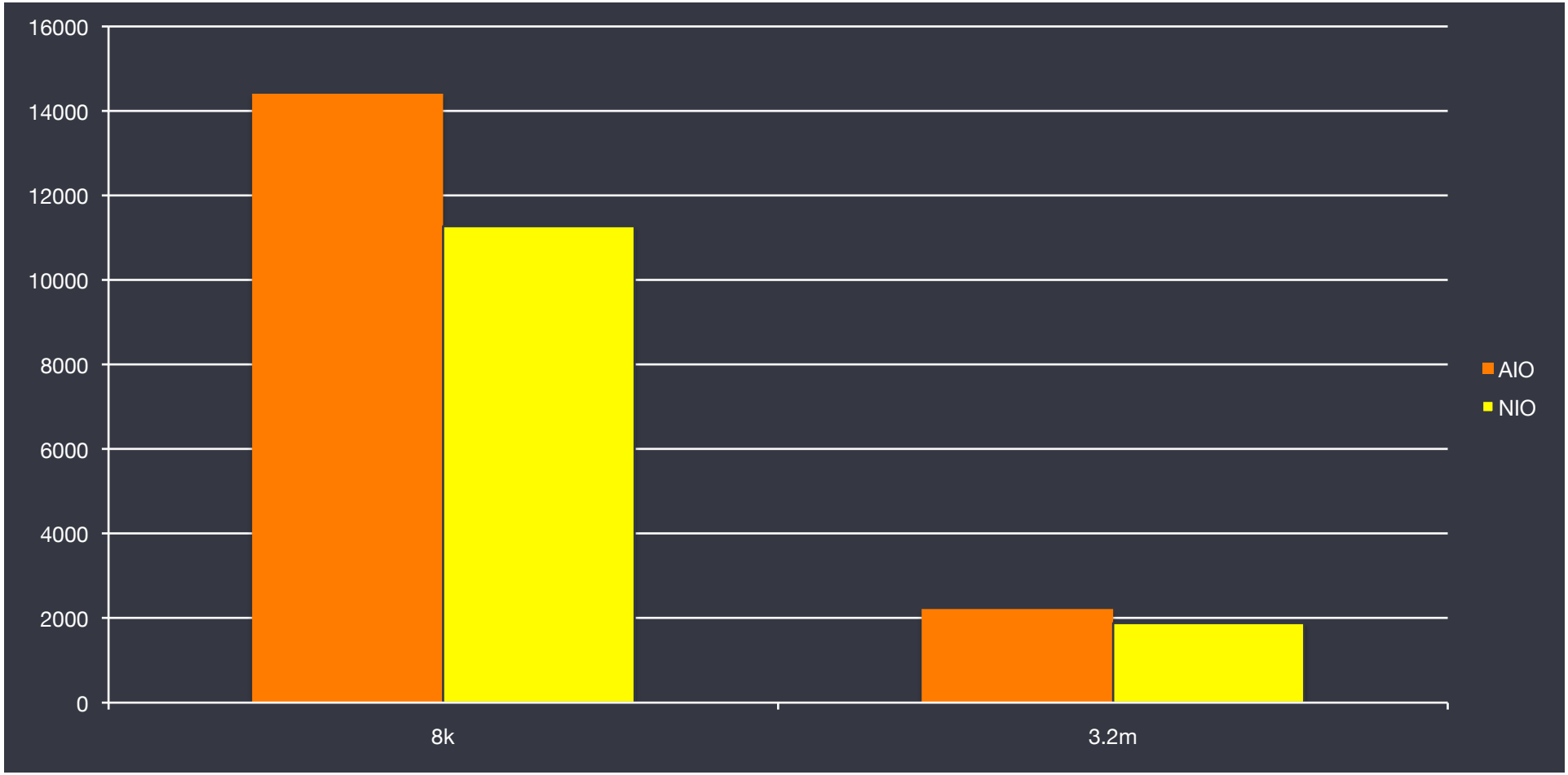
- Goal: NIO vs AIO using Grizzly WebServer
- Use Faban (http://faban.sunsource.net/) as the load driver.
- Using a load of 10 000 “users”, measure the operations per second and the average response time we are getting with NIO and AIO.
- Using JDK 7 b98 taken from http://openjdk.java.net/projects/nio/
- Launching Grizzly using:

```
java –jar grizzly-http-webserver.jar –p 8080 –a /var/www
```

- Use a FixedThreadPool of size 5


www.devoxx.com

# Benchmarking Grizzly against Grizzly



# Benchmarking Grizzly against Grizzly

| | 8k | 3.2m |
|---|---|---|
| AIO | 17401.075/0.045 | 2212.700/0.812 |
| NIO | 14265.967/0.086 | 1884.883/0.852 |

# Conclusion

Asynchronous I/O significantly improve the way scalable I/O based application can be constructed.

Use AIO **NOW** by downloading Grizzly or by inspecting our AIO implementation.

# Q&A

- Download Grizzly AIO:

  http://grizzly.dev.java.net

- Open JDK NIO.2 page:

  http://openjdk.java.net/projects/nio/

- My Tricks and Tips series on AIO

  http://weblogs.java.net/jfarcand

- AIO docs

  http://openjdk.java.net/projects/nio/javadoc/

- Join the Grizzly's buzz

  users@grizzly.dev.java.net

# Thanks for your attention!

http://twitter.com/
project_grizzly