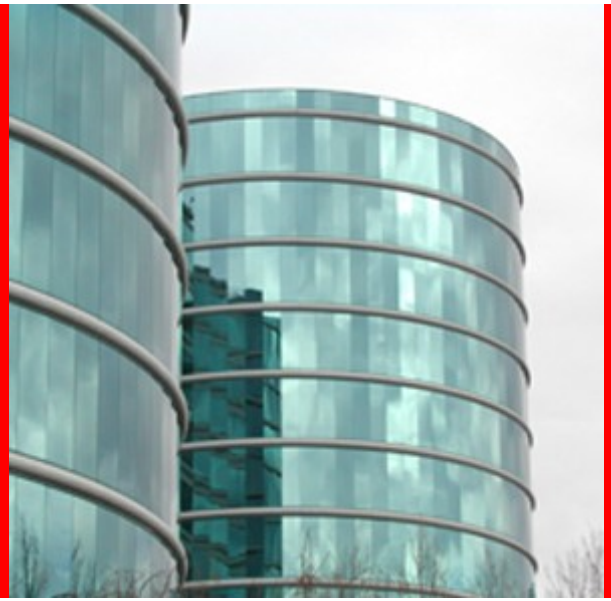


# Grizzly Overview

Oleksiy Stashok

# Agenda

- Introduction
- Grizzly core
- HttpServer
- WebSockets
- Port unification

# What is Grizzly?

ORACLE

# Introduction

- Open Source Project on java.net,
  http://grizzly.java.net
- Open Sourced under CDDL/LGPL license.
- Very open community policy.
- Development branches:
  - 2.0 – new rearch. API, main branch;
  - 1.9.x – legacy branch

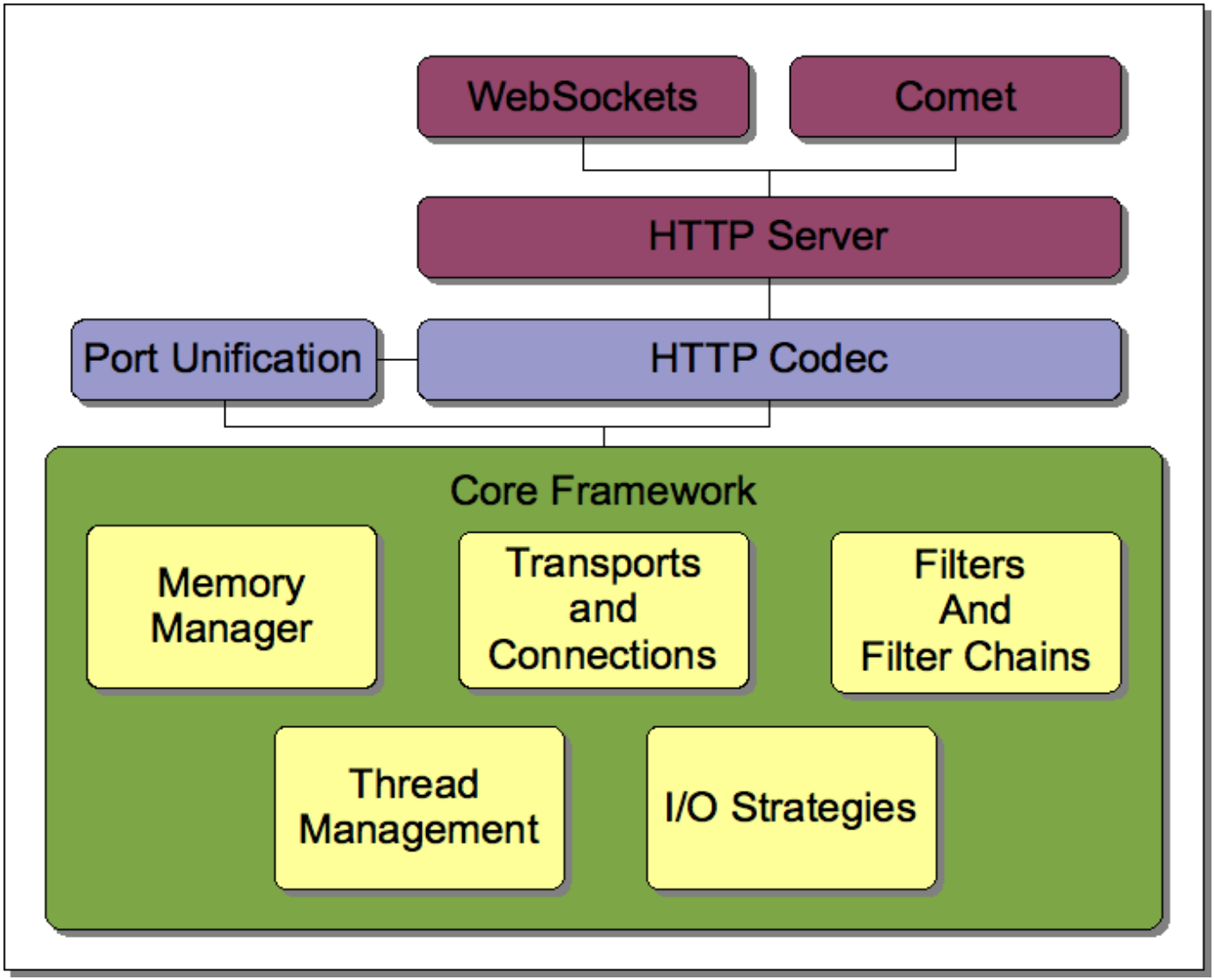
# Introduction

The Grizzly framework has been designed to help developers to take advantage of the Java™ NIO, help them to build scalable and robust client and server applications.

# Introduction

- Grizzly framework built using Java NIO primitives
  - Provides higher level abstractions over Java NIO
  - Hides complexity of programming with Java NIO
- NIO Framework:
  - Easy to use APIs for TCP/UDP/TLS components to build server/client applications
  - Brings non-blocking sockets to the protocol processing layer
  - Efficient Buffer management
  - Configurable threading strategies – choice of built-in/customizable ThreadPool implementations
  - Port Unification support
- HTTP Framework:
  - Support for non-blocking HTTP processing
  - Comet Support
  - WebSocket Support

# Introduction

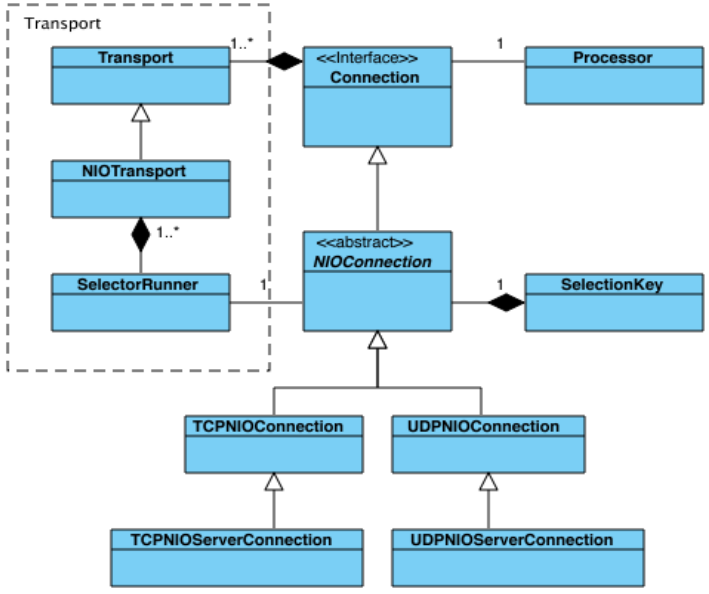



# Grizzly Core

ORACLE

# Transport and Connection

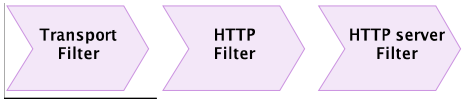
- Transport represents network transport implementation, for example TCPNIOTransport, UDPNIOTransport
- Connection is a Transport connection unit in Grizzly, like Socket for blocking I/O or Channel for NIO
- Transport relationship with Connection is 1-to-many



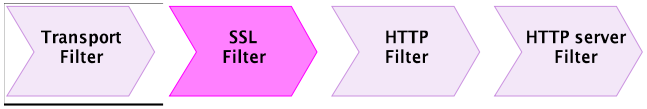
# Filter and FilterChain

- Filter represents a unit of processing work to be performed, whose purpose is to examine or/and modify the state of a transaction that is represented by a Context
- FilterChain models a computation as a series of Filters, that could be combined into a chain

*HTTP FilterChain*




*HTTP**S** FilterChain*

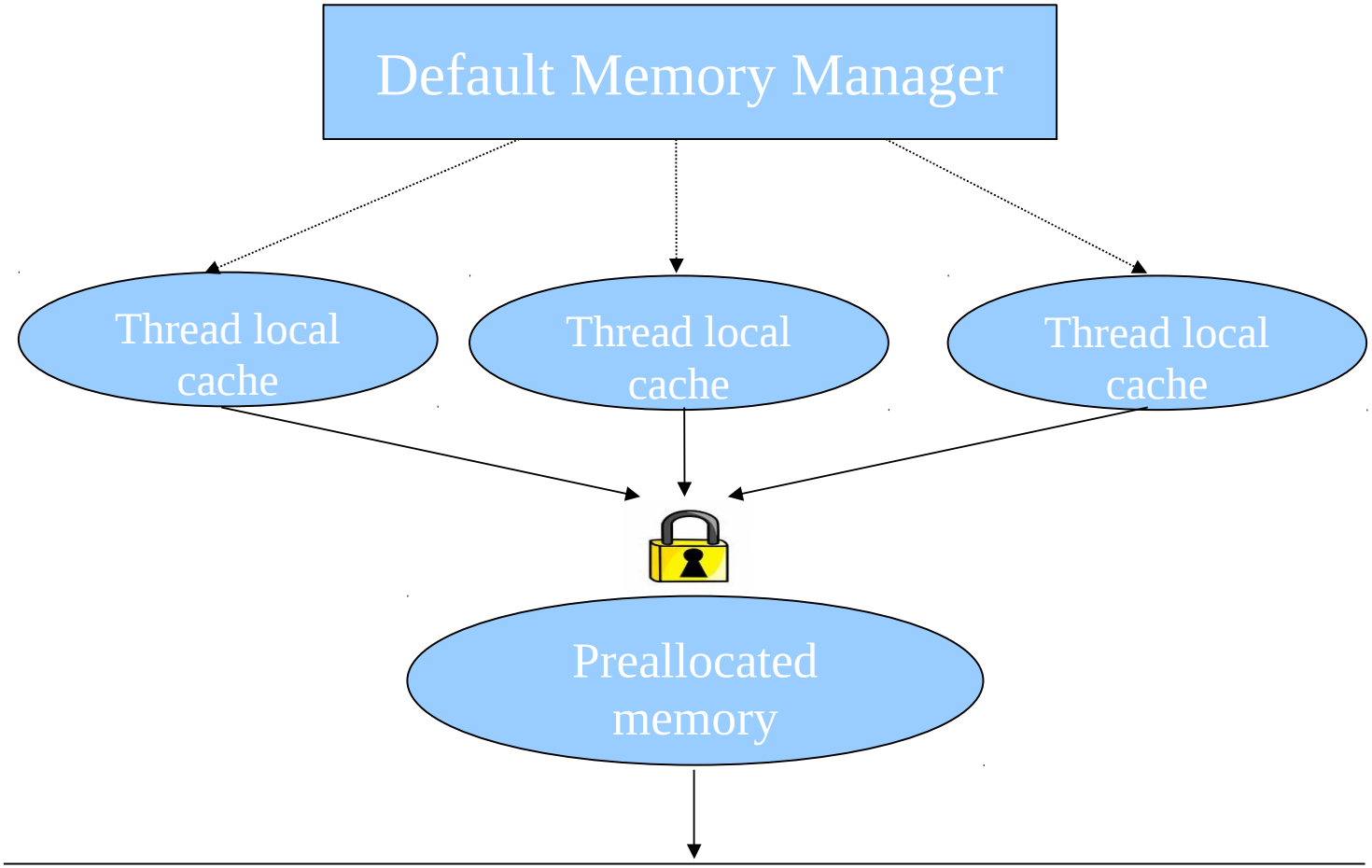


# Buffer

- Buffer represents allocated memory chunk in Grizzly
- Buffer API is similar to java.nio.ByteBuffer
- Buffer could be trimmed or released and freed memory will be reused by next allocate operation

```
/* Example of Buffer trimming. In this sample Grizzly will allocate only 1024 bytes from heap */
MemoryManager mm = transport.getMemoryManager();
Buffer buffer1 = mm.allocate(1024); //allocate 1024bytes
buffer1.put(array, 0, 512); // put 512 bytes to buffer1
buffer1.trim(); // return unused 512 bytes to cache
Buffer buffer2 = mm.allocate(512); // allocate 512 bytes
```

# How default MemoryManager works



# Echo server

```
01:         // Create a FilterChain using FilterChainBuilder
02:         FilterChainBuilder filterChainBuilder = FilterChainBuilder.stateless();
03:         // Add TransportFilter, which is responsible
04:         // for reading and writing data to the connection
05:         filterChainBuilder.add(new TransportFilter());
06:         filterChainBuilder.add(new EchoFilter());
07:
08:         // Create TCP transport
09:         TCPNIOTransport transport =
10:                 TransportFactory.getInstance().createTCPTransport();
11:         transport.setProcessor(filterChainBuilder.build());
12:
13:         // binding transport to start listen on certain host and port
14:         transport.bind(HOST, PORT);
15:
16:         // start the transport
17:         transport.start();
```

# EchoFilter

```
01: public class EchoFilter extends BaseFilter {
02:
03:     /**
04:      * Handle just read operation, when some message has come and ready to be
05:      * processed.
06:      *
07:      * @param ctx Context of {@link FilterChainContext} processing
08:      * @return the next action
09:      * @throws java.io.IOException
10:      */
11:     @Override
12:    public NextAction handleRead(FilterChainContext ctx)
13:            throws IOException {
14:        // Peer address is used for non-connected UDP Connection :)
15:        final Object peerAddress = ctx.getAddress();
16:
17:        final Object message = ctx.getMessage();
18:
19:        ctx.write(peerAddress, message);
20:
21:        return ctx.getStopAction();
22:    }
23: }
```

# HttpServer

# Server side APIs

APIs to make non-blocking interactions simple

- HttpHandler – think Servlet
- Asynchronous request processing (similar to what Servlet 3.0 offers)
- NIO{InputStream/OutputStream} – non-blocking reading/writing of binary data
- NIO{Reader,Writer} – non-blocking reading/writing of character data

ORACLE

# HttpHandler

Think Servlet – interact with Request/Response

```
private static class SimpleHandler extends HttpHandler {
        @Override
        public void service(final Request request,
                            final Response response) throws Exception {
            // CODE HERE
        }
}

public static void main(String[] args) {
        HttpServer server = HttpServer.createSimpleServer("0.0.0.0", 8080);
        server.getServerConfiguration().addHttpHandler(new SimpleHandler());
        server.start();
}
```

ORACLE

# Non-Blocking Reading

## Enabled via ReadHandler interface

- ReadHandler interface with three methods:
  - onDataAvailable()
  - onError()
  - onAllDataRead()
- Registered on NIOInputSource (NIOInputStream or NIOReader). When registering, you can optionally specify the minimum number of bytes that may be available before being notified.
- onDataAvailable() – data chunk ready to be consumed
- onAllDataRead() – all data sent by client

ORACLE

# Non-Blocking Reading

## Example…

```
response.suspend();
NIOInputStream in = request.getIntputStream(); // can also call getReader()

in.notifyAvailable(new ReadHandler() {
    public void onDataAvailable() throws Exception {
            echoAvailableData(in, response.getOutputStream());
            in.notifyAvailable(this);
    }
    …
    public void onAllDataRead() throws Exception {
            echoAvailableData(in, response.getOutputStream();
            response.resume();  // this should be in a finally block, but we ran out of room…
    }
});
```

# Non-Blocking Reading

This is far from optimal...

```
private static void echoAvailableData(NIOInputStream in,
                    NIOOutputStream out) throws IOException {

    // check how much can be read without blocking
    final int available = in.readyData();
    final byte[] data = new byte[available];
    in.read(data);
    out.write(data);

}
```

# Non-Blocking Reading

The zero-copy approach...

```
private static void echoAvailableData(NIOInputStream in,
                    NIOOutputStream out) throws IOException {

    out.write(in.readBuffer());

}
```

# Non-Blocking Writing

## Enabled via WriteHandler interface

- WriteHandler interface with two methods:
  - onWritePossible()
  - onError()
- Registered on NIOOutputSink (NIOOutputStream or NIOWriter). You may optionally specify the number of bytes that need to be written. The handler will not be invoked until that free space requirement is met.
- onWritePossible() will be invoked when a write operation may be performed without blocking

ORACLE

# Non-Blocking Writing

## Example…

```
response.suspend();
NIOOutputStream out = response.getOutputStream(); // can also call getWriter()

out.notifyCanWrite(new WriteHandler() {
    public void onWritePossible() throws Exception {
        // write content.  If  more data needs to be sent in another chunk, re-register the
        // WriteHandler by calling out.notifyCanWrite(this).  When complete, don't forget
        // to resume the response (response.resume()).
    }
    …
});
```

# NIO Streams are basic building blocks

- Using these streams, we've been able to create useful server-side processing constructs:
  - Non-blocking upload/download of content – server side multipart upload handling (many samples in our repository)
  - Non-blocking HTTP tunnels, proxies

ORACLE

# WebSockets

ORACLE

# Definition

“The WebSocket Protocol is designed to supersede existing bidirectional communication technologies that use HTTP as a transport layer to benefit from existing infrastructure (proxies, filtering, authentication). The WebSocket Protocol attempts to address the goals of existing bidirectional HTTP technologies in the context of the existing HTTP infrastructure; as such, it is designed to work over HTTP ports 80 and 443 as well as to support HTTP proxies and intermediaries, even if this implies some complexity specific to the current environment.”

RFC 6455

# Overview

- Bi-directional
- Full-duplex
- Operates over single TCP socket
- HTML5 standard
- Available via plain or SSL transports
- Looks a lot like HTTP, but really isn’t
- Has initial handshake phase

ORACLE

# Client Handshake

```
GET /demo HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

# Server Handshake

HTTP/1.1 101  Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

Sec-WebSocket-Protocol: chat

# ChatApplication

```
public class ChatApplication extends WebSocketApplication {
        @Override
        public void onMessage(final WebSocket senderWebsocket,
                              final String message) throws Exception {
            // broadcast the message to all connected clients except sender
            for (WebSocket websocket : getWebSockets()) {
                If (!websocket.equals(senderWebSocket)) {
                    websocket.send(message);
                }
            }
        }
}
```

# ChatApplication.main

```
public static void main(String[] args) {
        HttpServer server = HttpServer.createSimpleServer(“0.0.0.0”, 8080);

        // Register the WebSockets add on with the HttpServer
        server.getListener("grizzly").registerAddOn(new WebSocketAddOn());
        // initialize websocket chat application
        final WebSocketApplication chatApplication = new ChatApplication();
        // register the application
        WebSocketEngine.getEngine().register(chatApplication);

        server.start();
}
```

# Pros and Cons

- Pros
  - Effective bi-directional, full-duplex communication
  - Uses single TCP socket
  - HTML5 standard
    - Proxy and firewall friendly
- Cons
  - Adoption takes some time

ORACLE

# Port Unification

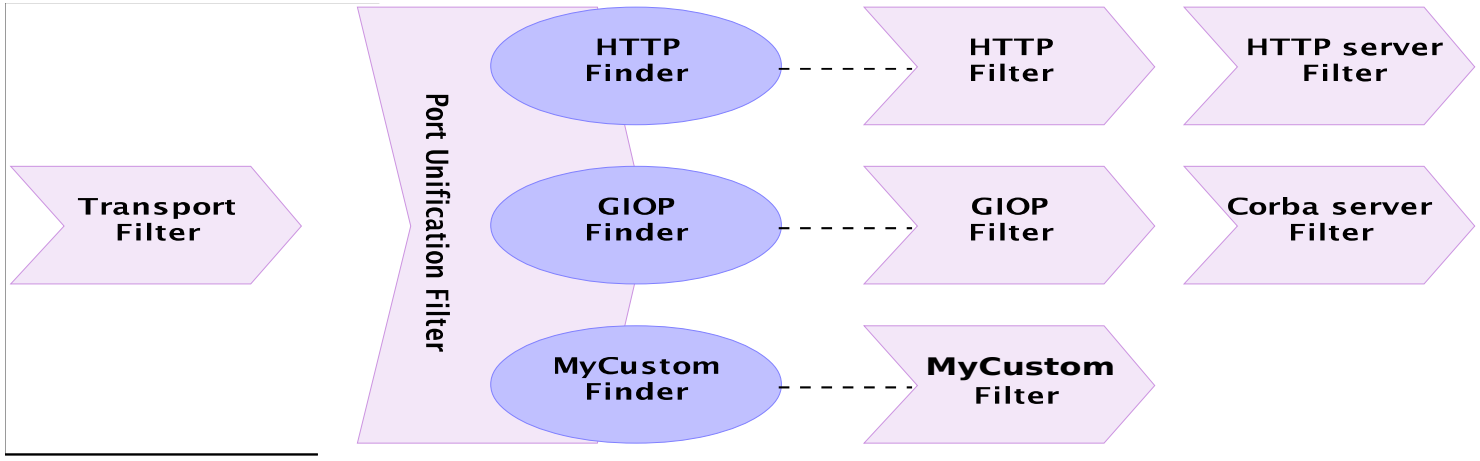
ORACLE

# Port Unification

- Use single TCP port to publish different types of services
  - HTTP
  - HTTPS
  - IIOP
  - Custom
- Pros
  - Easier administration (both client and server side)
- Limitations
  - Communication has to be initiated by a client

# Port Unification

- Each unified protocol/service is represented by
  - ProtocolFinder; figuring out if the data belongs to the specific protocol/service
  - FilterChain; represents specific protocol/service FilterChain



# Port Unification

- More info
  - http://grizzly.java.net/nonav/docs/docbkx2.0/html/portunif.html
  - http://java.net/projects/grizzly/sources/git/show/samples/portunif/src/main/java/org/glassfish/grizzly/samples/portunif

# HttpClient

ORACLE

# Grizzly 2.2 introduces HTTP Client

- Grizzly-based provider for the Sonatype Async HTTP Client API
- Has been released with their AHC version 1.7.0.
- We provide a Grizzly-only bundle (no Netty or Apache dependencies) under our maven namespace
- Uses the same non-blocking HTTP handling that the server-side uses

ORACLE

# Client Example

```
AsyncHttpClientConfig config = new AsyncHttpClientConfig.Builder().build();

AsyncHttpClient client = new AsyncHttpClient(new GrizzlyAsyncHttpProvider(config), config);

Request request = new RequestBuilder("GET")
                .setUrl("http://foo.com/foo.html")
                .addQueryParameter("q", "a b”)
                .build();

Response response = client.executeRequest(request).get();
```

ORACLE

# AsyncHttpHandler

Optional – may be passed to the executeRequest method

- Provides callbacks that will be invoked at various stages of response processing.
  - onBodyPartReceived()
  - onStatusReceived()
  - onHeadersReceived()
  - onCompleted() (returns result of processing)
- Interface uses generics, so the result of onCompleted() can be whatever is needed by the application.

ORACLE

# Further API Details…

- AHC could be a talk on itself.
- We recommend reading the documentation provided by Sonatype on their GitHub repository.

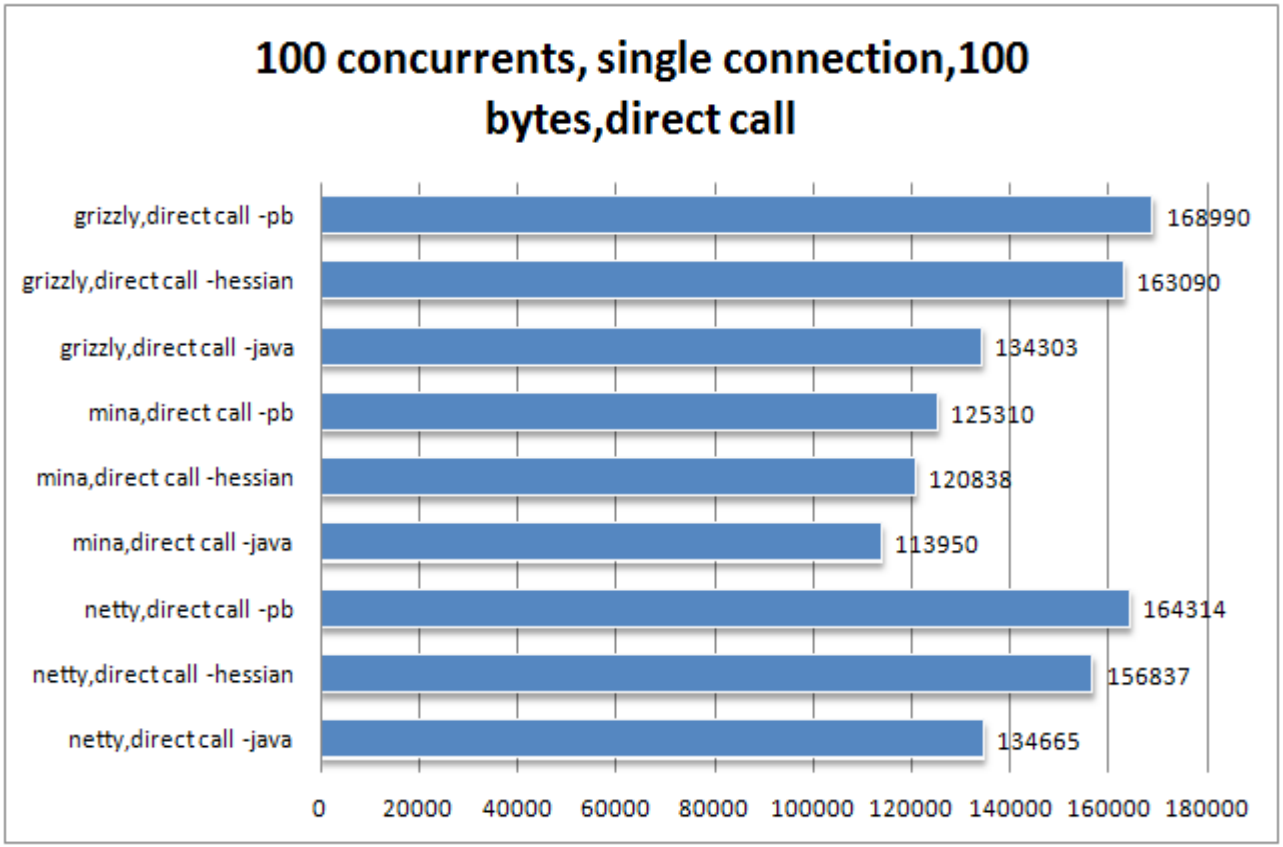
ORACLE

# Performance

# General Framework Performance

3rd Party RPC framework provided numbers for us

- http://code.google.com/p/nfs-rpc



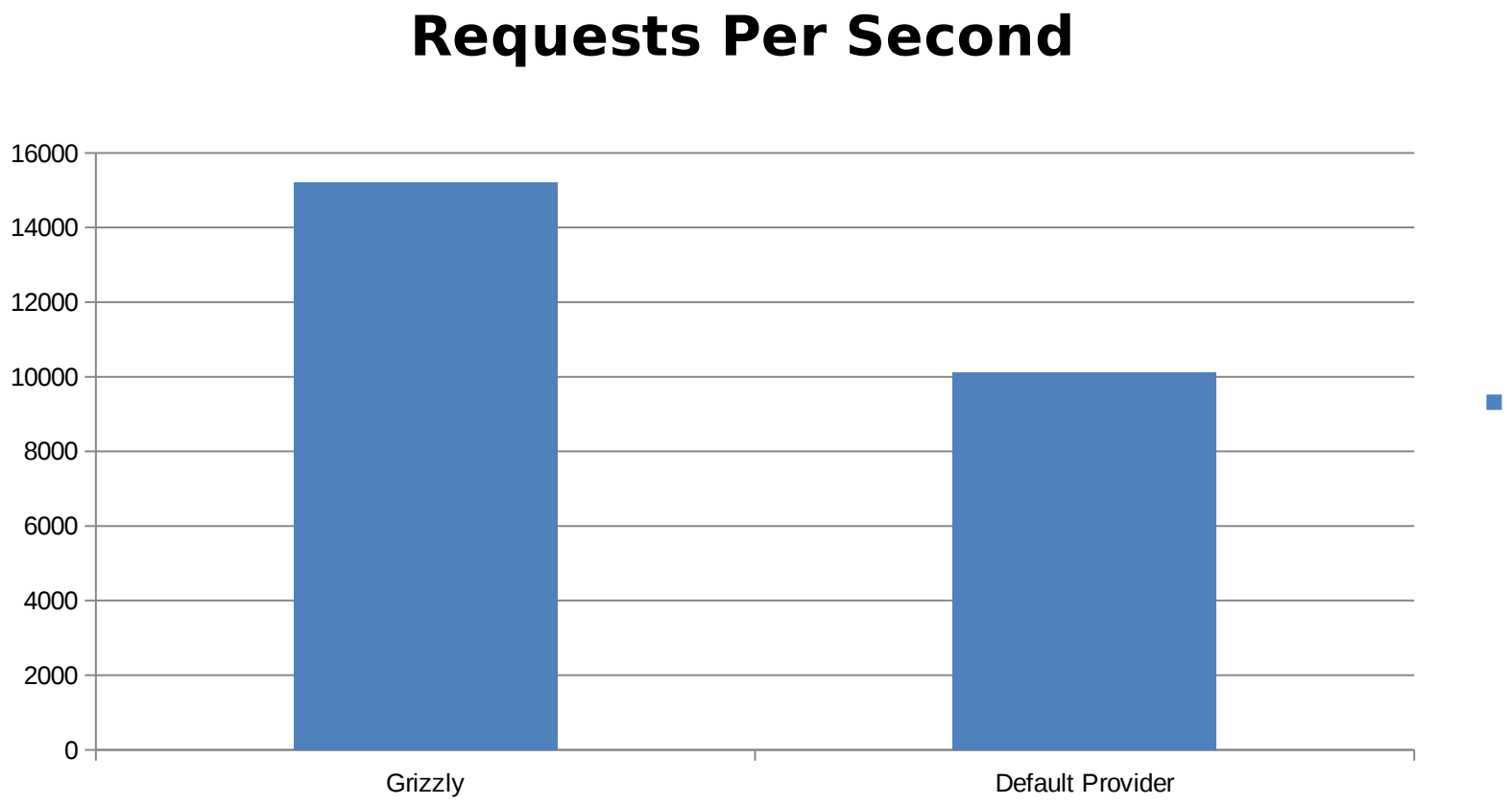
# Grizzly-based AHC Performance

Benchmark can be obtained here:

- git@github.com:rlubke/java-http-client-benchmark.git
- We ran the benchmark on an 8-core laptop using localhost communication.
- Numbers we show here don’t include Jetty or Apache. However, Apache is currently faster but they also use Blocking I/O.
- Chart compares Grizzly against the default non-blocking provider.

ORACLE

# Grizzly-based AHC Performance



# For more information

- Project Grizzly:
  - http://grizzly.java.net
- Project Grizzly mailing lists:
  - dev@grizzly.java.net
  - users@grizzly.java.net
- Blogs:
  - http://blogs.oracle.com/oleksiys
  - http://antwerkz.com
  - http://blogs.oracle.com/rlubke/
  - http://weblogs.java.net/blog/jfarcand/