Project Grizzly – High Performance for Server Applications

Jeanfrancois Arcand and Charlie Hunt

Senior Staff Engineers

Sun Microsystems, Inc.

Agenda

- Introduction
- What is Project Grizzly
- Grizzly Performance
- History of Grizzly
- Grizzly Architecture
- Who is using Grizzly
- Where to find more information

Agenda

- Introduction
- What is Project Grizzly
- Grizzly Performance
- History of Grizzly
- Grizzly Architecture
- Who is using Grizzly
- Where to find more information

Sun microsystems

Introduction

Writing scalable server applications in Java has always been difficult.

Before the advent of Java NIO, thread management issues made it impossible for a server to scale to thousands of users. But using Java NIO presents its own problems in terms of program design and implementation. Project Grizzly removes these problems by providing a framework to free you of the performance traps many developers have encountered using Java NIO

In this session, we discuss Project Grizzly which we developed and how use it to build scalable, robust server applications.

What we will talk about

- In this presentation we will:
  - > describe Project Grizzly
  - > show its performance
  - > giving a brief history of Grizzly
  - > provide a architecture overview of Grizzly
  - > tell you who is already using Grizzly, who is looking at Grizzly
  - > where to find additional information on Grizzly

Agenda

- Introduction
- What is Project Grizzly
- Grizzly Performance
- History of Grizzly
- Grizzly Architecture
- Who is using Grizzly
- Where to find more information

What is Project Grizzly

- Open Source Project on java.net, (https://grizzly.dev.java.net)
- Open Sourced under CDDL license
- 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 vote
  - No project member has more voting power than any other project member

What is Project Grizzly

- Uses Java NIO primitives and hides the complexity programming with Java NIO
- Easy-to-use high performance APIs for TCP, UDP and SSL communications
- Brings non-blocking sockets to the protocol processing layer
- Utilizes high performant buffers and buffer management
- Choice of several different high performance thread pools

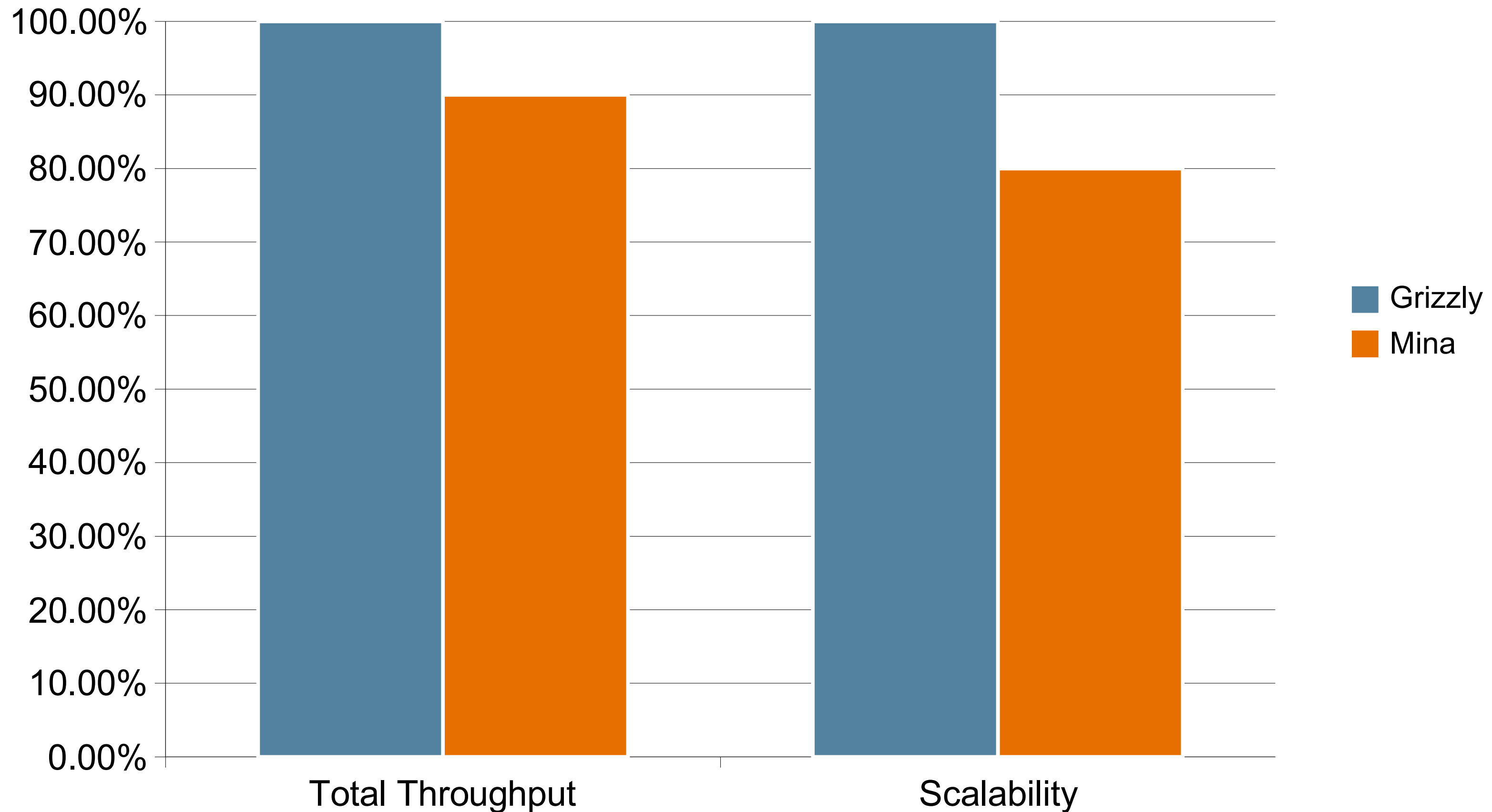
Agenda

- Introduction
- What is Project Grizzly
- Grizzly Performance
- History of Grizzly
- Grizzly Architecture
- Who is using Grizzly
- Where to find more information

Grizzly Performance

- Comparing Grizzly to Apache MINA on AsyncWeb
- What is MINA?
- Apache MINA (Multipurpose Infrastructure for Network Applications) is a network application framework which helps users develop high performance and high scalability network applications easily.
- What is AsyncWeb?
- AsyncWeb is a high-throughput, non blocking Java HTTP engine - designed throughout to support asynchronous request processing. AsyncWeb is build on top of MINA.

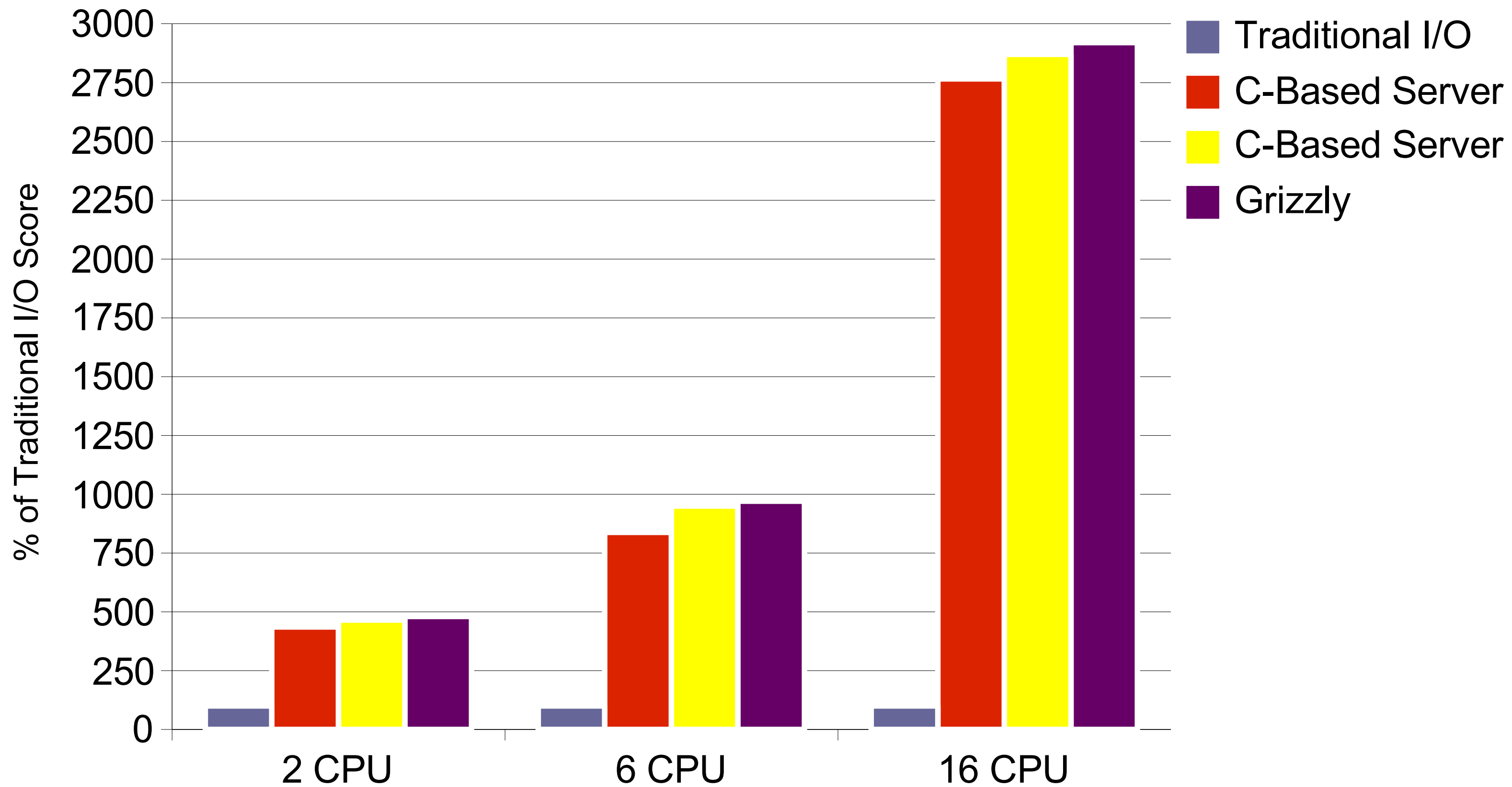
Grizzly versus Apache MINA



Grizzly HTTP Performance

- Internally developed benchmark
- Designed to measure scalability
- How many concurrent clients can we support:
  - Avg. client think time, 8 seconds
  - 90% response time within 3 seconds
  - Error rate < .1%

Grizzly HTTP vs other HTTP Servers



Agenda

- Introduction
- What is Project Grizzly
- Grizzly Performance
- History of Grizzly
- Grizzly Architecture
- Who is using Grizzly
- Where to find more information

Grizzly History

- Grizzly was born in 2004 under the GlassFish project, (https://glassfish.dev.java.net). This later became known as Grizzly 1.0
- HTTP over TCP/SSL was the first implementation
- Grizzly 1.0 shipped with Sun Java System Application Server 8.1 PE, 8.2 PE/EE and all GlassFish distributions, replacing native Sun WebServer runtime.
- Initially used to build an HTTP Web Server, replacing Tomcat's Coyote Connector and Sun WebServer 6.1

Grizzly History

- Grizzly 1.0 became extremely popular in 2006. Multiple protocol implementations were built on top of it
- But Grizzly 1.0 had HTTP protocol specific implementation details included in its transport logic
- The main class, SelectorThread, contained several artifacts specific to http such as file caching, request monitoring, etc

Grizzly History

- Several classes needed to be extended in order to use the framework
- Example: JettySelectorThread extends SelectorThread
- Example: SSLSelectorThread extends SelectorThread
- The Grizzly 1.0 mixed 'extension' and 'implementation'

Grizzly History

- But, Grizzly 1.0 was still a good choice for nearly all TCP/HTTP based protocols.
- Several projects successfully utilized Grizzly 1.0:
  - JRuby On Grizzly
  - Alaska's HTTP BC component
  - GlassFish v3 micro kernel
  - Phobos in NetBeans
  - SOAP over TCP integration in GlassFish
  - Comet / Cometd
  - AsyncWeb on Grizzly
  - GlassFish v2
  - Sun Web 2.0 Developer pack (REST HTTP Server)

Grizzly History

- Grizzly 1.5 began development in 2006
- Currently under review and will release very soon
- Grizzly 1.5 objectives
  - Remove all dependencies to HTTP and/or GlassFish
  - All 1.0 applications must still work with 1.5
  - Support all tricks and tips learned during development of Grizzly 1.0 (performance, NIO performance traps, etc.)
  - **Keep it simple!!**
- Open Source Grizzly occurred February 6, 2007 with Grizzly 1.5 and it is under final code review now

Agenda

- Introduction
- What is Project Grizzly
- Grizzly Performance
- History of Grizzly
- Grizzly Architecture
- Who is using Grizzly
- Where to find more information

Grizzly Architecture

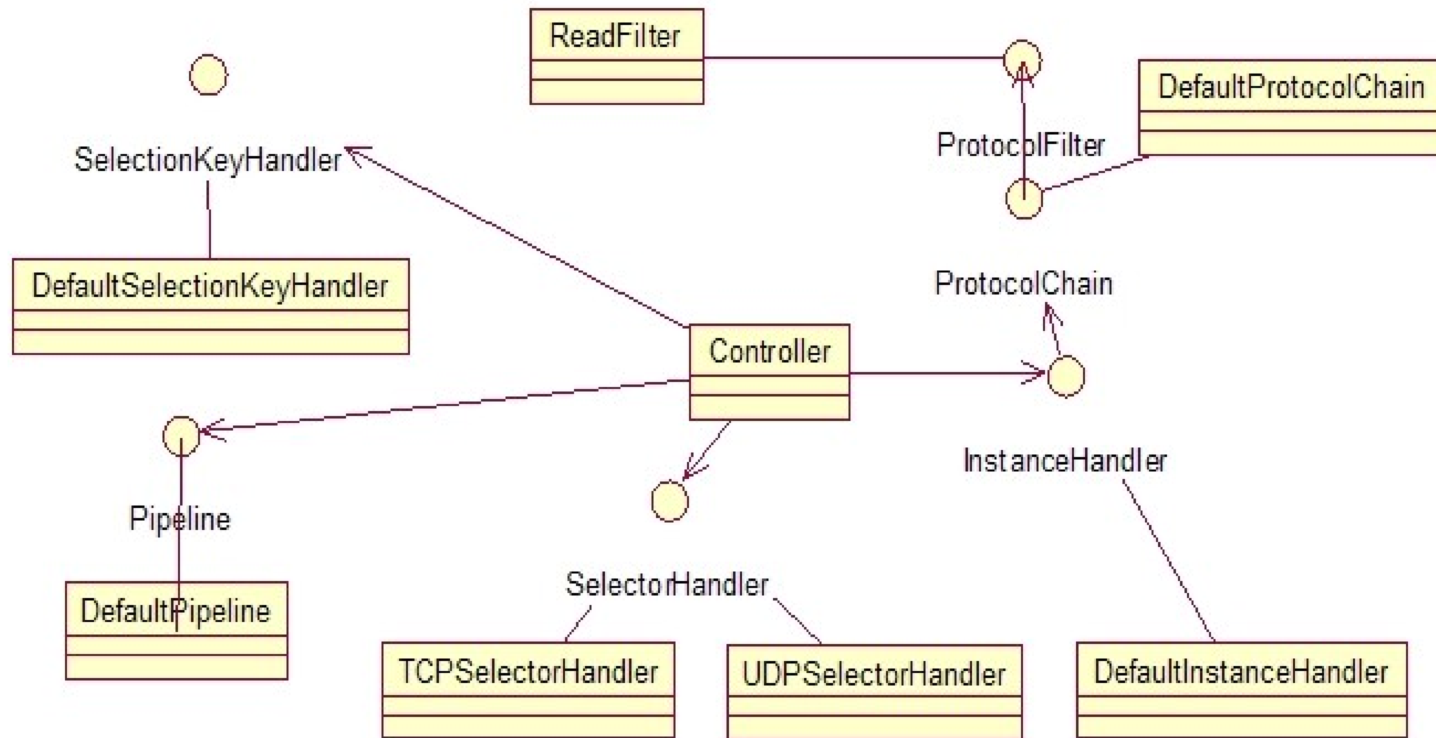
- Grizzly 1.5 project is now under https://grizzly.dev.java.net
- Uses Subversion (svn) instead of cvs for source code control
- Uses Maven2 instead of Ant
- Multiple modules instead of a single (large) one
  - **grizzly**
  - comet
  - cometd
  - http
  - http-utils

Grizzly Architecture

- The entire Grizzly framework source code classes are located under the modules/grizzly folder
- The grizzly module only contains the framework classes, without any dependencies on third party code or contains an protocol specifics
- The Grizzly WebServer is located under the http module. The http module exposes the same interfaces as Grizzly 1.0. Hence all http based on Grizzly 1.0 implementation continue to work without any modifications.

Grizzly Class Diagram

JGD



Controller

JGD

- Main entry point when using the Grizzly Framework -- Controller
- A Controller is composed of
  - > SelectorHandler
  - > SelectionKeyHandler
  - > ProtocolChainInstanceHandler
  - > ProtocolChain
  - > Pipeline
- All of these components are configurable using the Grizzly Framework

SelectorHandler

- A SelectorHandler handles all java.nio.channels.Selector operations. One or more instance of a Selector are handled by SelectorHandler.
- The logic for processing of SelectionKey interest (OP_ACCEPT,OP_READ, etc.) is usually defined using an instance of SelectorHandler.
- This is where the decision of attaching an object to SelectionKey occurs.

SelectorHandler (Cont.)

JGD

```
/**
 * This method is guaranteed to always be called before
 * Selector.select().
 */
public void preSelect(Context controllerCtx)  throws IOException;;
/**
 * Invoke the Selector.select() method.
 */
public Set<SelectionKey> select(Context controllerCtx) throws
IOException;
/**
 * This method is guaranteed to always be called after
 * Selector.select().
 */
public void postSelect(Context controllerCtx)  throws IOException;
```

SelectionKeyHandler

- A SelectionKeyHandler is used to handle the life cycle of a SelectionKey.
- Operations likes cancelling, registering or closing of SelectionKeys are handled by a SelectionKeyHandler.

SelectionKeyHandler (Cont.)

JGD

```
/**
 * Expire a SelectionKey.
 */
public void expire(SelectionKey key);
/**
 * Cancel a SelectionKey and close its Channel.
 */
public void cancel(SelectionKey key);
/**
 * Close the SelectionKey's channel input or output, but keep alive
 * the SelectionKey.
 */
public void close(SelectionKey key);
```

InstanceHandler

- An InstanceHandler is where one or several ProtocolChain(s) are created and cached.
- An InstanceHandler decides if a stateless or stateful ProtocolChain needs to be created.
- Note: InstanceHandler is being renamed to a ProtocolChainInstanceHandler for improved clarity

InstanceHandler (Cont.)

JGD

```
/**
 * Return an instance of ProtocolChain.
 */
public ProtocolChain poll();
/**
 * Pool an instance of ProtocolChain.
 */
public boolean offer(ProtocolChain instance);
```

Pipeline

- An interface used as a wrapper around any kind of thread pool.
- There are several implementation of Pipelines in Grizzly 1.5.
- The best performing implementation is the default configured Pipeline.

ProtocolChain

- A ProtocolChain implements the "Chain of Responsibility" pattern (for more info, take a look at the classic "Gang of Four" design patterns book).
- The ProtocolChain API models a computation as a series of "protocol filter" that can be combined into a "protocol chain".

ProtocolChain (Cont.)

- Important: The owning ProtocolChain must call the postExectute() method of each ProtocolFilter in a ProtocolChain in reverse order of the invocation of their execute() methods

ProtocolChain (Cont.)

JGD

```
/**
 * Add a ProtocolFilter to the list. ProtocolFilter
 * will be invoked in the order they have been added.
 */
public boolean addFilter(ProtocolFilter protocolFilter);
/**
 * Remove the <code>ProtocolFilter</code> from this chain.
 */
public boolean removeFilter(ProtocolFilter theFilter);

public void addFilter(int pos, ProtocolFilter protocolFilter);
```

ProtocolFilter

JGD

- The API for ProtocolFilter consists of a two methods:

  execute(Context)

  postExecute(Context)

- which are passed a "protocol context" containing the dynamic state of the computation

ProtocolFilter

- A ProtocolFilter encapsulates a unit of processing work to be performed, whose purpose is to examine and/or modify the state of a transaction that is represented by a ProtocolContext.
- Individual ProtocolFilter(s) can be assembled into a ProtocolChain.

ProtocolFilter

- When using the default ProtocolChain, ProtocolFilter implementations should be designed in a thread-safe manner
- In general, this implies that ProtocolFilter classes should not maintain state information in instance variables.

ProtocolFilter (Cont.)

JGD

- Instead of maintaining state information in a ProtocolFilter, state information should be maintained via suitable modifications to the attributes of the ProtocolContext which are passed to the execute() and postExecute() methods.

ProtocolFilter (Cont.)

JGD

```
/**
 * Execute a unit of processing work to be performed. This ProtocolFilter
 * may either complete the required processing and return false,
 * or delegate remaining processing to the next ProtocolFilter in a
 * ProtocolChain containing this ProtocolFilter by returning true.
 */
public boolean execute(Context ctx) throws IOException;
/**
 * Execute any cleanup activities, such as releasing resources that were
 * acquired during the execute() method of this ProtocolFilter instance.
 */
public boolean postExecute(Context ctx) throws IOException;
```

Example 1 - TCP

- By default, the Grizzly Framework bundles default implementation for TCP and UPD transport. The TCPSelectorHandler is instantiated by default.
- As an example, supporting the TCP protocol should only consist of adding the appropriate ProtocolFilter like:

Example – 1 TCP (Cont.)

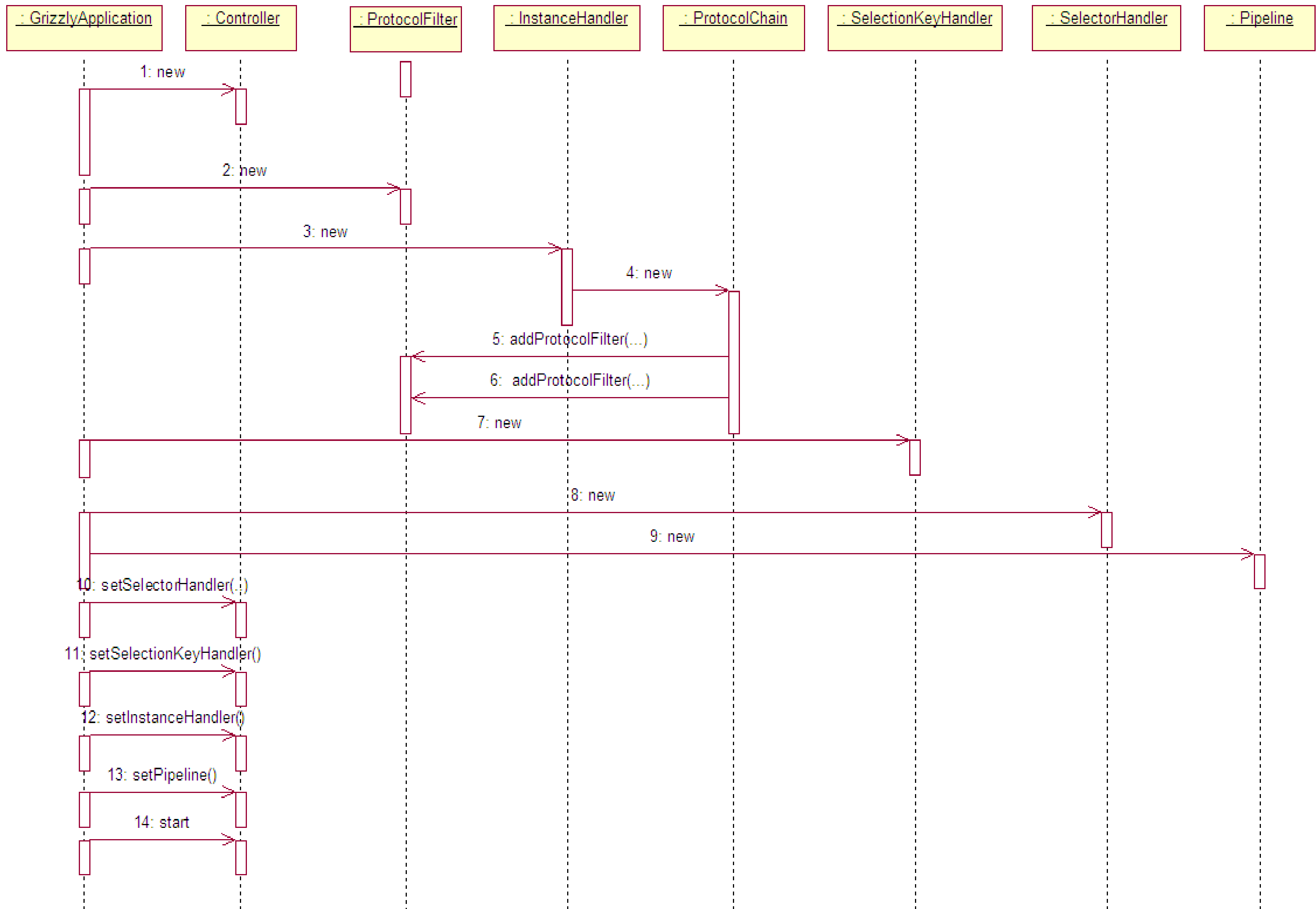
JGD

```
Controller con = new Controller();
con.setInstanceHandler(new DefaultInstanceHandler(){
  public ProtocolChain poll() {
      ProtocolChain protocolChain = protocolChains.poll();
      if (protocolChain == null){
          protocolChain = new DefaultProtocolChain();
          protocolChain.addFilter(new ReadFilter());
          protocolChain.addFilter(new HTTPParserFilter());
     }
     return protocolChain;
   }
});
```

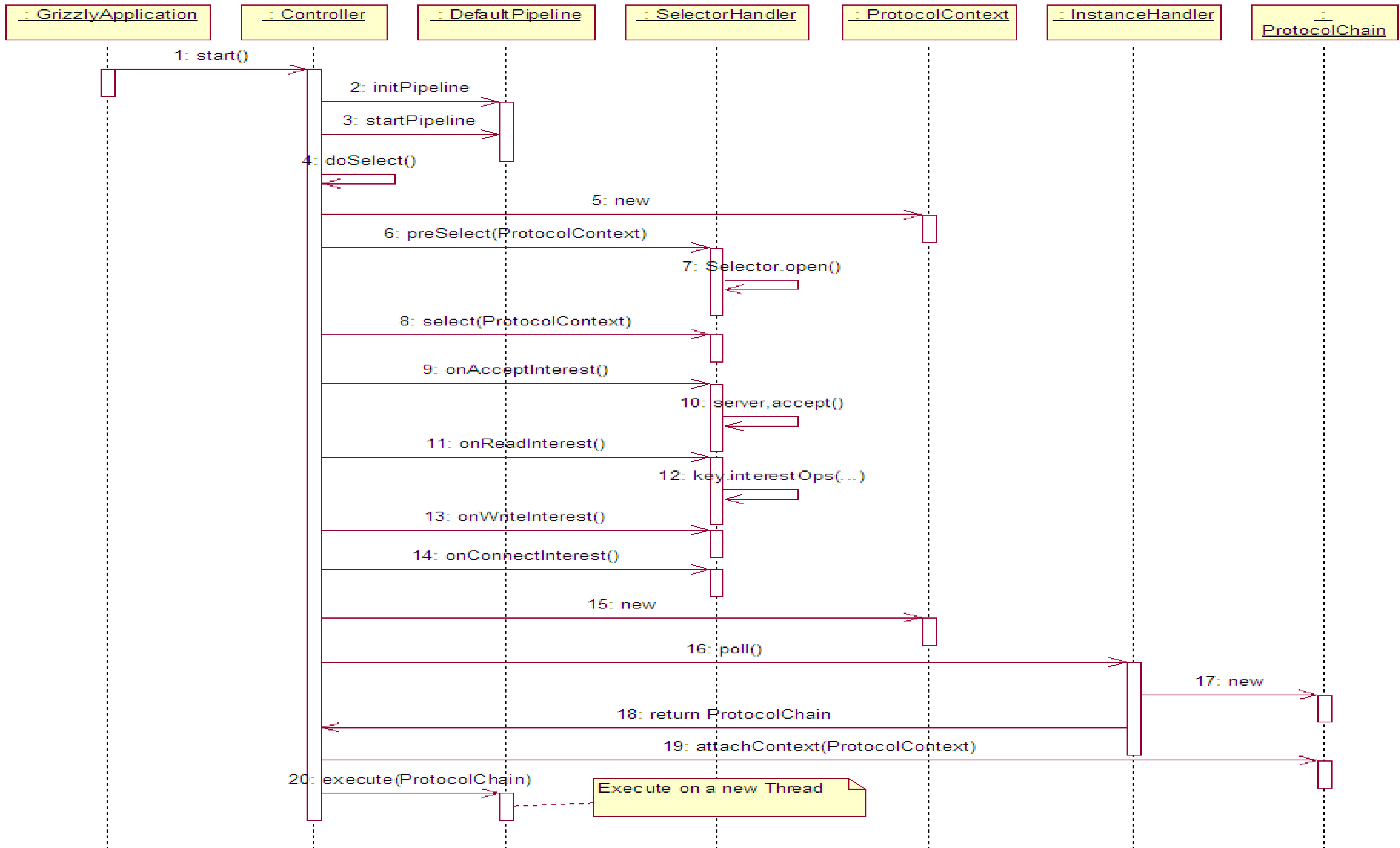
Creating a Controller

Sun microsystems

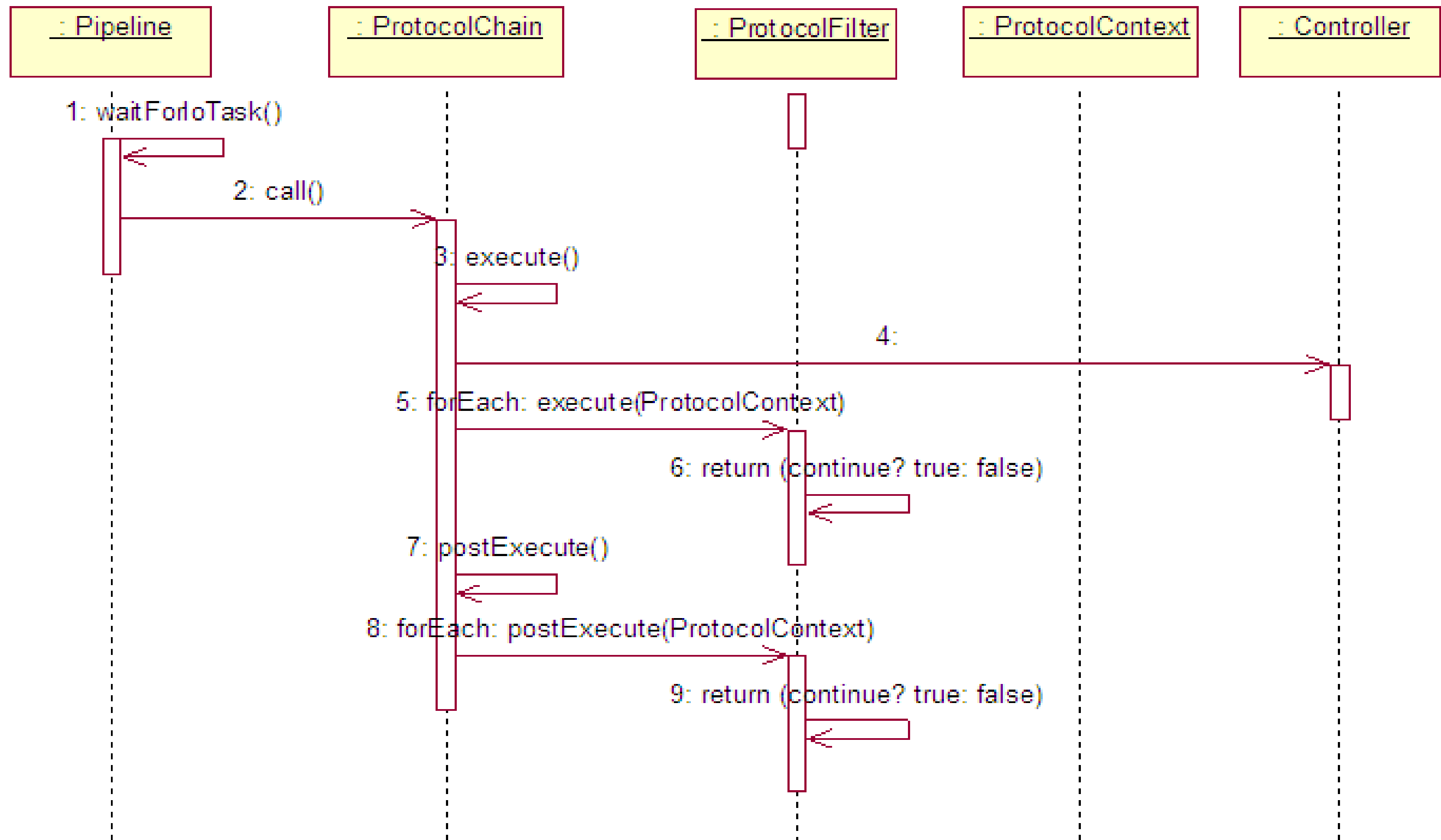
JGD



Request Handling



Worker Thread execution



Example – 2 UDP

JGD

```
Controller con = new Controller();
con.setInstanceHandler(new DefaultInstanceHandler(){
  public ProtocolChain poll() {
      ProtocolChain protocolChain = protocolChains.poll();
      if (protocolChain == null){
          protocolChain = new DefaultProtocolChain();
          protocolChain.addFilter(new UDPReadFilter());
          protocolChain.addFilter(new ParserFilter());
   }
     return protocolChain;
  }
});
con.setSelectorHandler(new UDPSelectorHandler());
```

Agenda

- Introduction
- What is Project Grizzly
- Grizzly Performance
- History of Grizzly
- Grizzly Architecture
- Who is using Grizzly
- Where to find more information

Who is using Grizzly

- Jetty
- Alaska – SeeBeyond (Open ESB)
- Tango – Microsoft Web Services Interoperability
- JRuby on Grizzly
- PHP on Grizzly
- Project Phobos in NetBeans
- GlassFish v2 (Http Server), v3 micro kernel, Port Unification
- Comet / Cometd
- Ning (Grizzly Asynchronous Request Processing)

Who is looking at / investigating Grizzly

- Sun Java System Message Queue
- Sun JDK ORB
- GlassFish ORB
- Derby / Java DB
- Major enterprise players and middleware companies
- ...and many more (subscribe to our mailing list to learn who!!!)

Agenda

- Introduction
- What is Project Grizzly
- History of Grizzly
- Grizzly Architecture
- Grizzly Performance
- Who is using Grizzly
- Where to find more information

Call to Action

- Download GlassFish and experience the fastest web container on the planet
- Project GlassFish home page http://glassfish.dev.java.net
- Download Grizzly source code https://grizzly.dev.java.net
- Join Project Grizzly and be added to Project Grizzly mailing lists

Where to find more information

- Project Grizzly home page
  https://grizzly.dev.java.net
- Jeanfrancois Arcand's blog
  http://weblogs.java.net/blog/jfarcand
- Charlie Hunt's blog
  http://blogs.sun.com/CharlieBrown
- Project Grizzly mailing lists,
  dev@grizzly.dev.java.net and/or
  users@dev.grizzly.java.net
- Join us at JavaOne 2007 for a session and a BOF on Grizzly!

jeanfrancois.arcand@sun.com

charlie.hunt@sun.com