WELCOME TO JAVAPOLIS

Project Grizzly:

Jean-Francois Arcand

Senior Staff Engineer
Sun Microsystems

Agenda

- Introduction
- What is Grizzly 1.0
- Introduction to Grizzly.Next
  - Framework
  - HTTP
  - Asynchronous Request Processing
    - > Comet
- Performance
- Q&A

What is Grizzly 1.0

- Grizzly 1.0 is a TCP framework that uses lower level Java NIO primitives, and provides high-performance APIs for socket communications.
- Grizzly 1.0 was originally part of GlassFish and SJSAS. It was mainly used to build a HTTP Web Server, replacing Tomcat's Coyote Connector and Sun WebServer 6.1.
- Developed in 2004, it became more and more used as a WebServer outside GlassFish. It also started getting extended to support other Sun's products (OpenESB, WSIT stack, Jersey, Phobos, etc.)

What is Grizzly 1.0 (Cont.)

- 1.0 evolved to a generic framework but was polluted with HTTP concepts and GlassFish specific interfaces
  - The main class, SelectorThread, contains several reference to http like file caching, request monitoring, etc.
- Several classes needed to be extended in order to use the framework
  - JettySelectorThread extends SelectorThread
  - SSLSelectorThread extends SelectorThread
- The Framework mixed ‘extension’ and ‘implementation’.

What is Grizzly 1.0 (Cont.)

- Still, Grizzly 1.0 was still a good choice for any TCP/HTTP based protocol. Several implementation successfully extended the framework:
  - 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
- Hence it was **extremely important** to avoid breaking 1.0 implementation.

Project Grizzly and its Extensions

PHP

Cometd

GlassFish V3

Phobos

JRuby

Jersey

Comet

GlassFish v2

Adapter

Synchronous Processing

Asynchronous Processing

HTTP

RealTime/ Sailfin

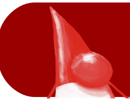
Jetty

Pogo.com (EA)

Grizzly Framework

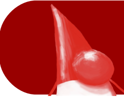
Introduction to Grizzly.Next

- Grizzly.Next and up is a tentative to fix 1.0 limitations.
- The goals are:
  - Remove all dependencies to HTTP and/or GlassFish
  - 1.0 Applications must still works with 1.5
  - Support all tricks and tips learned during development of 1.0 (performance, NIO workaround, etc.)
  - **Keep it simple!!**



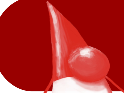
Controller

- Main entry point when using the Grizzly Framework. A Controller is composed of
  - Handlers
    - SelectorHandler
    - SelectionKeyHandler
    - InstanceHandler
  - ProtocolChain
  - Pipeline.
- All of those components are configurable by client using the Grizzly Framework.



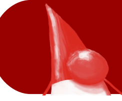
Example 1 - TCP

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



Example – 1 TCP (Cont.)

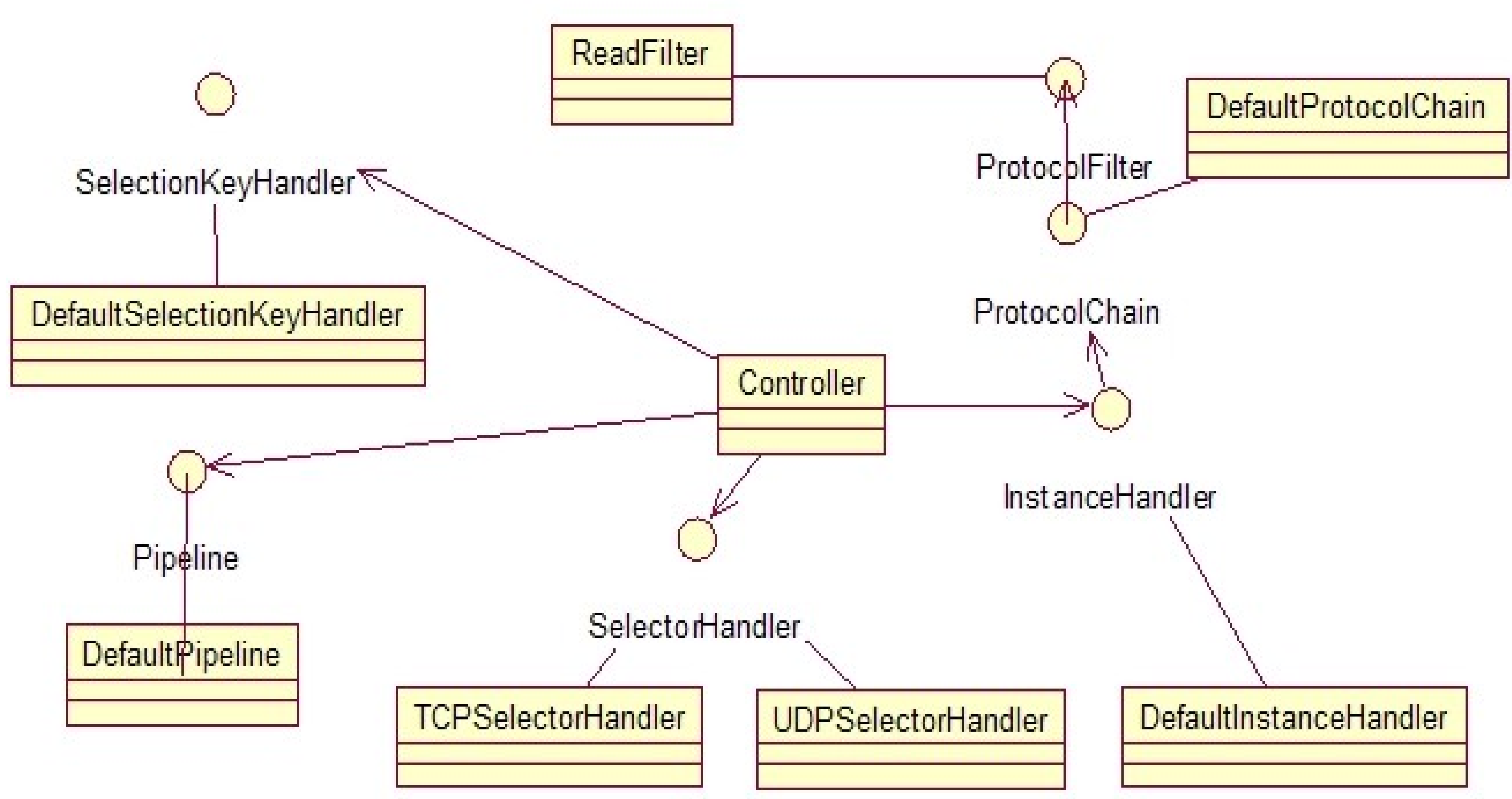
```
Controller con = new Controller();
con.setInstanceHandler(new DefaultInstanceHandler(){
  public ProtocolChain poll() {
       ProtocolChain protocolChain = protocolChains.poll();
       if (protocolChain == null){
           protocolChain = new DefaultProtocolChain();
           protocolChain.addFilter(new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



Example – 2 UDP

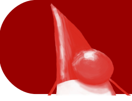
```
Controller con = new Controller();
con.setInstanceHandler(new DefaultInstanceHandler(){
  public ProtocolChain poll() {
       ProtocolChain protocolChain = protocolChains.poll();
       if (protocolChain == null){
           protocolChain = new DefaultProtocolChain();
           protocolChain.addFilter(new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

Main Concepts – 1.5 Class Diagram



InstanceHandler

- An InstanceHandler is where one or several ProtocolChain are created and cached.
- An InstanceHandler decide if a stateless or statefull ProtocolChain needs to be created.

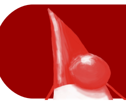


Pipeline

- An interface used as a wrapper around any kind of thread pool.

ProtocolChain

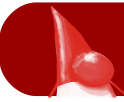
- A ProtocolChain implement the "Chain of Responsibility" pattern (for more info, take a look at the classic "Gang of Four" design patterns book).
- Towards that end, the Chain API models a computation as a series of "protocol filter" that can be combined into a "protocol chain".



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 can be assembled into a ProtocolChain, which allows them to either complete the required processing or delegate further processing to the next ProtocolFilter in the ProtocolChain.

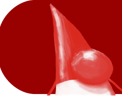
Architecture – HTTP layer



Grizzly HTTP layer

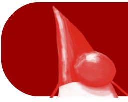
- Lightweight HTTP 1.0/1.1 based server
- Extremely easy to embed.
- Small footprint.
- Performance is extremely good
- **Good performance apply to both Synchronous processing and Asynchronous Processing**

Architecture – Adapter



Grizzly HTTP layer

- Easy to embedded. Only have to interact with one object: SelectorThread
- Write an implementation of com.sun.grizzly.tcp.Adapter class.
- The Adapter is the glue code between the HTTP layer and the program that embed Grizzly.
- Easy to inject your own ProtocolFilter to manipulate the HTTP Request



Architecture - Adapter

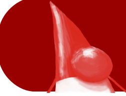
Main entry point for most of HTTP based server

- Most Grizzly 1.0 implementation write their own com.sun.grizzly.tcp.Adapter implementation.
  - Project Phobos in Netbeans
  - Jersey
  - JRuby on Grizzly
- Simple Interface

```
public void service(Request req,Response res);
```

- Request contains all HTTP information like:
  - Method: GET/POST/TRACE
  - Headers: content-length, content-type, etc.
- Works at the bytes level.

Source: Please add the source of your data here



Example – StaticResourceAdapter

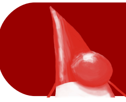
```
public void service(Request req, final Response res)  {
   MessageBytes mb = req.requestURI();
   ByteChunk requestURI = mb.getByteChunk();
   String uri = req.requestURI().toString();
    ....
    res.setStatus(200);
    res.setContentType(ct);
    res.sendHeaders();
    ....
    res.doWrite(chunk);
    res.finish()
}
```

Example

```
SelectorThread selectorThread = new SelectorThread();
selectorThread.setPort(port);

selectorThread.setAdapter(
                    new StaticResourcesAdapter());
// OR
selectorThread.setAdapter(
                    new GlassFishAdapter());

selectorThread.setWebAppRootPath(folder);
selectorThread.listen();
```



Architecture - Adapter

- But this approach is problematic if you need to embedded more than one http based implementation because you needs one adapter per implementation
  - > One for PHP
  - > One for Comet
  - > One for JRuby on Rail
- They cannot listen to the same http port!
- Adapter notes cannot be shared.
- Solution: GlassFish V3 project!

Source: Please add the source of your data here

Architecture – GlassFish V3

Tomcat 6

Phobos

PHP

JRuby

Comet

SWDP Rest

GlassFish V3

Synchronous Processing

Asynchronous Processing

HTTP

RealTime/ Sailfin

Jetty

Pogo.com (EA)

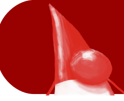
Grizzly Framework

Advantages

- Same performance
- Same port, different context
- Adapter Notes management (caching)
- ThreadLocal storage management
- Common administration : deploy, undeploy...
- Container loading/unloading
- Adapter boilerplate reduced
- Intra-adapter communication

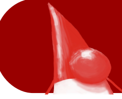
Application adapter

- In GlassFish V3, each application can register its adapter.
- Adapter have context root
- Requests are dispatched based on the registered context roots
- Registration/Unregistration of Adapter instances is automatically handled by the runtime
- GlassFish has no knowledge of the target container type, Adapter is the interface



Asynchronous Request Processing

- Allow for “parking” a request; a type of “continuation” at the request processing level
- The goal is to be able to build, on top of Grizzly, a scalable ARP implementation that doesn't hold one thread per connection, and achieve as closer as possible the performance of synchronous request processing (SRP).
- Ex: OpenESB (ex: SeeBeyond) HTTP BC Component.



Example – Asynchronous Request Processing

```
SelectorThread selectorThread = new SelectorThread();
selectorThread.setPort(port);
selectorThread.setWebAppRootPath(folder);
selectorThread.setAdapter(
                    new JRubyAdapter());

AsyncHandler asyncHandler = new DefaultAsyncHandler();
asyncHandler.addAsyncFilter(new CometAsyncFilter());
selectorThread.setAsyncHandler(asyncHandler);

selectorThread.listen();
```

What is Comet Request Processing (or Ajax Push)

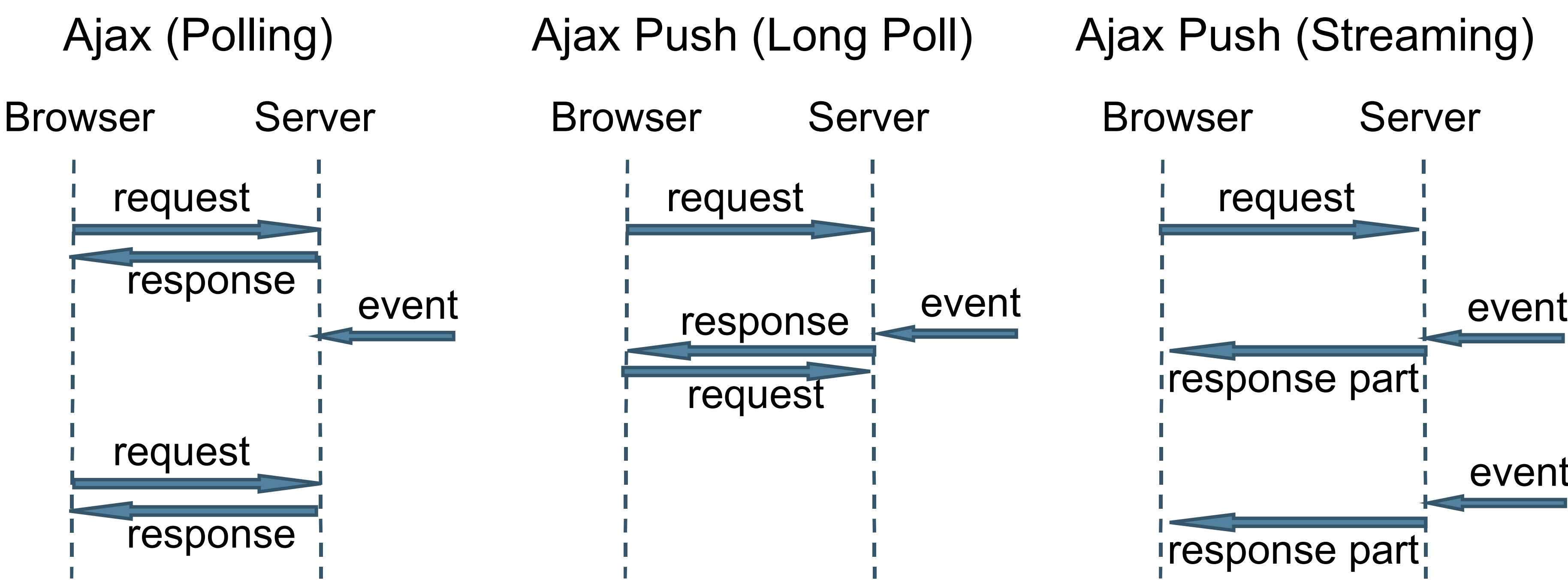
Comet is a programming technique that enables web servers to send data to the client without having any need for the client to request for it. It allows creation of event-driven web applications which are hosted in the browser.

What is Ajax Push (aka Comet)?

- Use it to create highly responsive, event driven applications in a browser
  - > Keep clients up-to-date with data arriving or changing on the server, without frequent polling
- Pros
  - > Lower latency, not dependent on polling frequency
  - > Server and network do not have to deal with frequent polling requests to check for updates
- Example Applications
  - > GMail and GTalk
  - > Meebo
  - > JotLive
  - > WebMc (OSS WebEX developped by ICESoft.com)
  - > 4homemedia.com (build on top of GlassFish's Comet)
  - > Many more …

How does “Push” to the browser work?

Standard Ajax compared to Ajax Push options



How does the “Push” to the browser works

- Deliver data over a previously opened connection
  - > Always “keep a connection open”; do not respond to the initiating request until event occurred
  - > Streaming is an option by sending response in multiple parts and not closing the connection in between

Technology Solutions

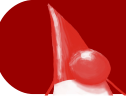
- Use new I/O (NIO) non-blocking sockets to avoid blocking a thread per connection (Cool this is exactly what Grizzly does!)
- Use technology that supports asynchronous request processing
  - Release the original request thread while waiting for an event
  - May process the event/response on another thread than the original request
- Advantages
  - Number of clients is primarily limited by the number of open sockets a platform can support
  - Could have all clients (e.g. 10’000) “waiting” without any threads processing or blocked

Grizzly Comet supports:

- Asynchronous Content Handlers
  - > Allow handling asynchronous read and write
- Suspendable Requests
  - > suspend/resume requests/response
- Container managed server Push
  - > push data from one connection to another

Grizzly Comet Framework details

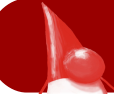
- Grizzly offer three solutions:
  - **Grizzly Comet**: Comet framework for Servlet/JSP deployed in GlassFish or Jetty. Support suspendable request, asynchronous content handler and container managed server push.
  - **Grizzlet**: simple POJO object deployed on top of Grizzly WebServer. Support suspendable request and container managed server push.
  - **Grizzly Continuation**: Simple API for suspending/resuming a connection. Support suspendable request.



Project Grizzly Performance

Comparing Project Grizzly to Apache MINA both running AsyncWeb

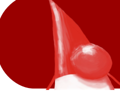
- What is MINA?
  - Apache MINA (Multipurpose Infrastructure for Network Applications) is a network application framework which helps users develop high performance and high scalability network applications easily.
- What is AsyncWeb?
  - AsyncWeb is a high-throughput, non blocking Java platform HTTP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



Project Grizzly Performance

How did we performance test?

- Client load generated via faban
  - faban.sunsource.net
- Two modes of measurement
  - Throughput
    - Limited # of clients
    - No Think Time
  - Scalability
    - Max # of clients with 90% response time metric
    - Think time



Project Grizzly versus Apache MINA both running AsyncWeb

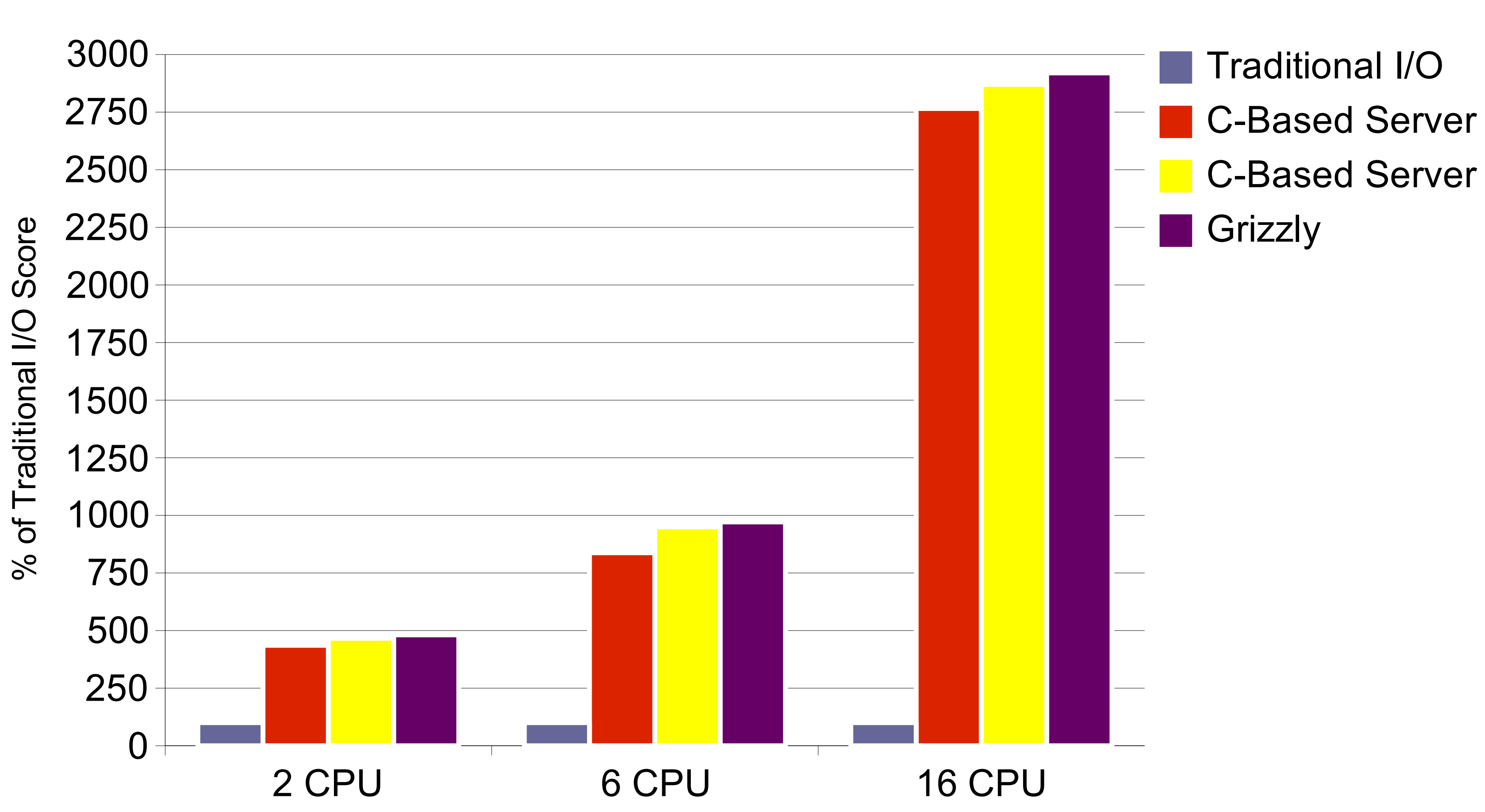
Higher is better, normalized to Grizzly score

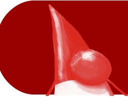
Source: Internal Benchmark Tests

Project Grizzly Performance

- Tested against a benchmark designed to:
  - Measure scalability, specifically to measure how many concurrent clients can be supported with:
    - Average client think time of 8 seconds
    - 90% response time within 3 seconds
    - Error rate < 0.1%

Project Grizzly HTTP vs other HTTP Servers





Summary

- Grizzly 1.0 migration to 1.5 is simple.
  - Ex: Grizzly 1.0 HTTP based application will works without any changes using module/http
- 1.5 make it simple to extends and isolate concept.
- Support out-of-the-box TCP, UDP, TLS via SelectorHandler implementation.
- Extremely simple to implement your own http extension.

Q&A

Project Grizzly

http://grizzly.dev.java.net

jeanfrancois.arcand@sun.com