Customizing the “Grizzly” NIO Framework

Jean-Francois Arcand, Staff Engineer
Sreeram Duvur, Senior Staff Engineer
Andreas Egloff, Staff Engineer
Sun Microsystems
http://www.sun.com

BOF-0520

Customizing the “Grizzly” NIO Framework

This presentation covers some new projects involving Grizzly NIO Framework, such as customizable request processing, Asynchronous Request Processing and Resource Consumption Management.

Agenda

Introduction

Grizzly Framework

Resource Consumption Management
  Features in GlassFish

OpenESB HTTP SOA features in SJSAS 9.0

Q&A

Agenda

Introduction

Grizzly Framework

Resource Consumption Management
  Features in Glassfish

OpenESB HTTP SOA features in SJSAS 9.0

Q&A

Introduction

- Grizzly is an NIO Framework used in several Sun product: SJSAS 8.2/9.0 PE, Tango, etc.
- Grizzly is not tied to HTTP and can be used as a generic NIO Framework.
- This presentation will give an overview of the Grizzly Framework, and the we will discuss two extensions, Application Resource Allocation and JBI.

Source: Please add the source of your data here

Agenda

Introduction

Grizzly Framework

Resource Consumption Management features in GlassFish

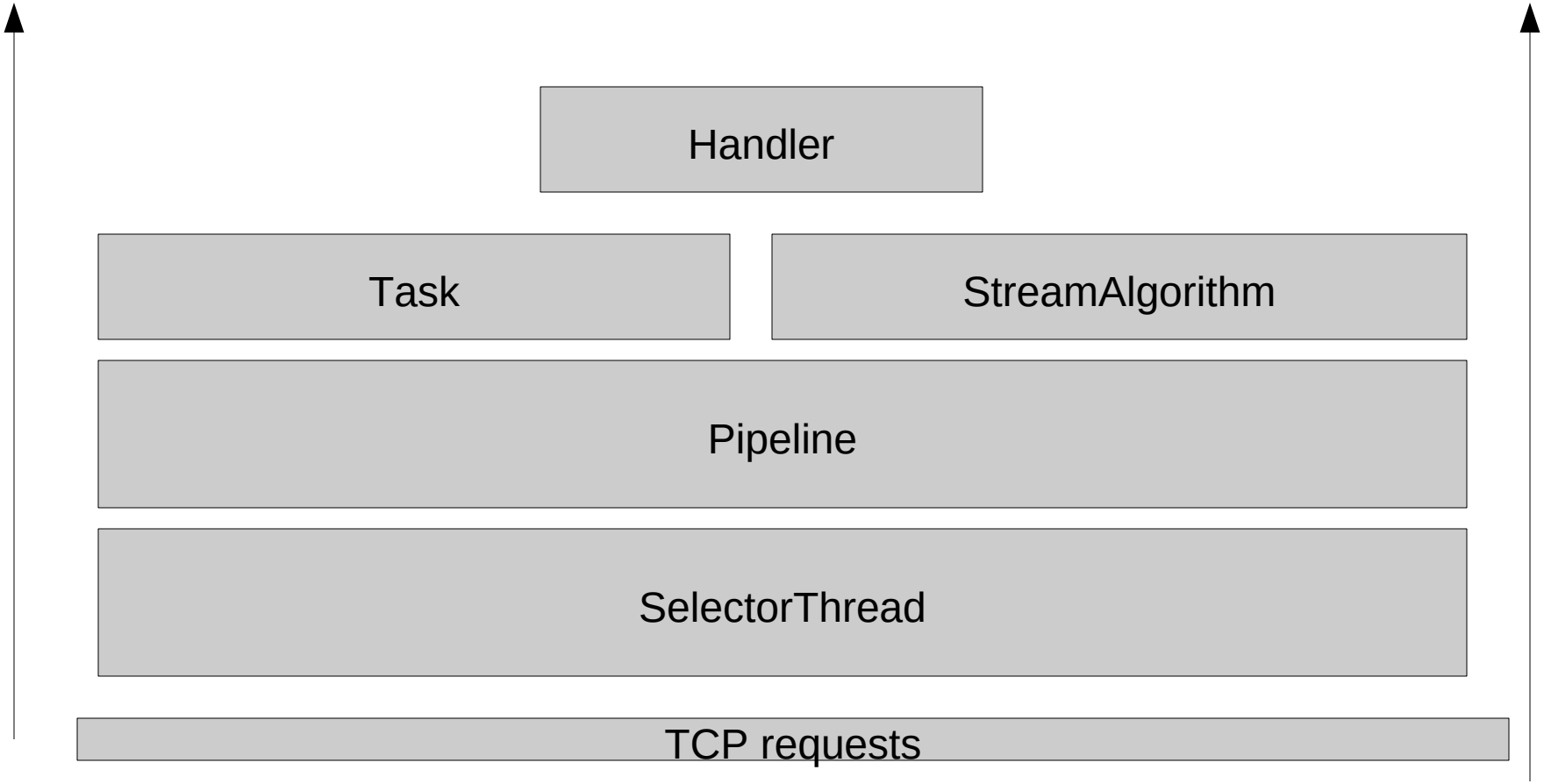
OpenESB HTTP SOA features in SJSAS 9.0

Q&A

Grizzly Framework

- What Is the Grizzly Framework?
  - Grizzly is a multi protocols (HTTP, UDP, etc.) framework that uses lower level Java NIO primitives, and provides high-performance APIs for socket communications.
- Grizzly can be extended in several places, from the management of low level byte buffer to thread management.
- Next couple of slides will describe where and how you can extend Grizzly.

Grizzly Main Components



Source: Please add the source of your data here

Definitions

- SelectorThread: java.nio.channels.Selector implementation.
- Pipeline: An execution queue. Most of the time implemented as a wrapper around a thread pool.
- Task: An execution token which handles the life cycle of the SocketChannel
- StreamAlgorithm: The strategy used to pull out bytes from SocketChannel (when and how)

SelectorThread

- Main entry point in Grizzly.
- Handles NIO low level events:
  - SelectionKey events: OP_ACCEPT, OP_READ
  - SelectionKey registration/de-registration with the java.nio.channels.Selector.
- Handles the allocation and life cycle of Pipelines, Tasks, StreamAlgorithms and Handlers.
- **Can be embedded in any products.**

Pipeline

- Responsible of the execution of a Task. The Pipeline can execute using the caller thread or create its own thread pool.
- Grizzly ships by default with three:
  - LinkedListPipeline: thread pool based on a linked list
  - ThreadPoolExecutorPipeline: thread pool based on java.util.concurrent.ThreadPoolExecutor
  - ExecutorServicePipeline: based on java.util.concurrent.Executors

StreamAlgorithm

- Implement the strategy of deciding:
  - The ByteBuffer type (direct, heap or view)
  - Deciding when we start/stop reading bytes from the SocketChannel.
  - The registration/de-registration on the SelectionKey with the main Selector (SelectorThread) or using a temporary Selector.
  - Ship with three implementation.

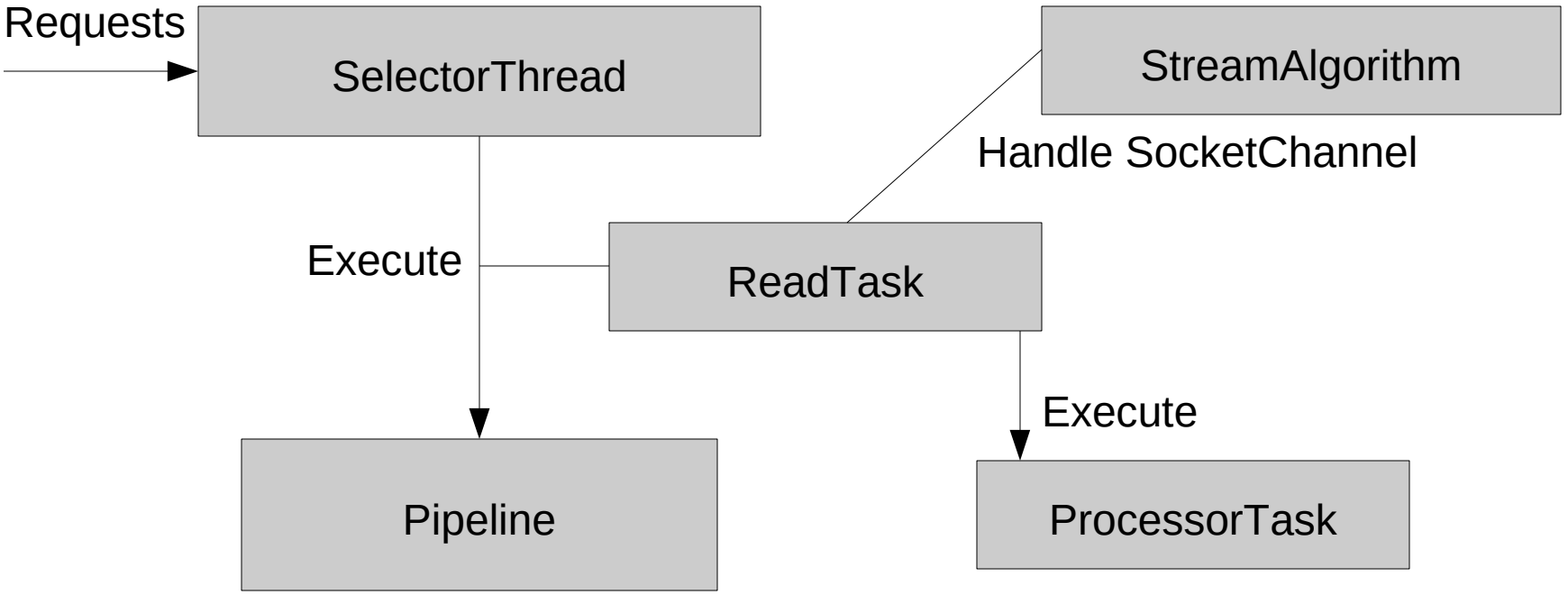
Tasks

- Execution token that can be executed by a thread pool (implement java.lang.Runnable).
- Configurable using via the StreamAlgorithm.
- The SelectorThread will handle the life cycle of Tasks.
- Usualy implement the request logic operations.

Tasks (Cont.)

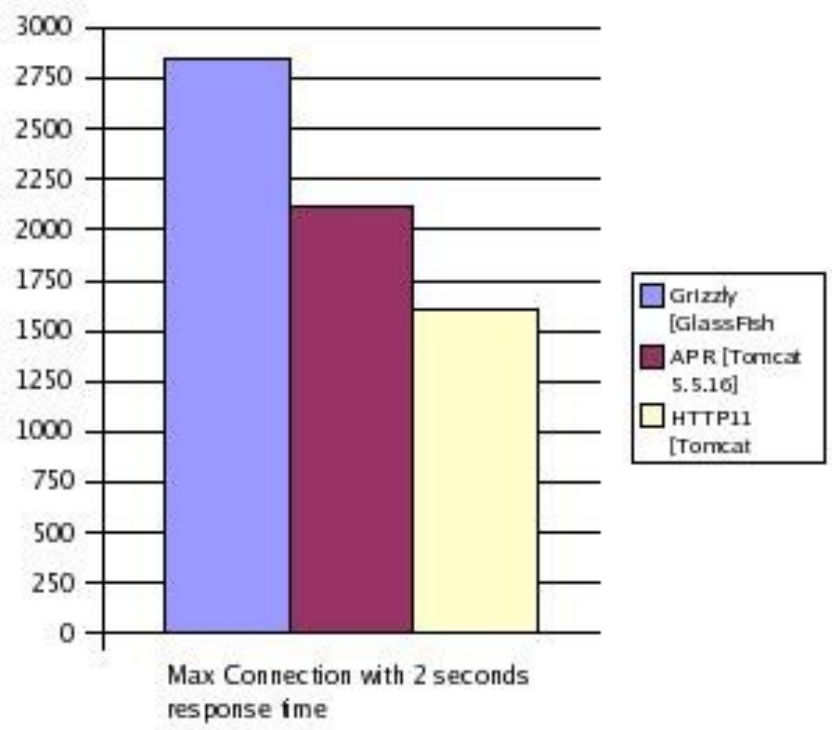
- Several implementations:
  - ReadTask: handle the SocketChannel operations and decide what to do next using its associated StreamAlgorithm
  - ProcessorTask: HTTP processing implementation. Parse the HTTP request header and body.
  - AsyncReadTask: Same as ReadTask but can handle asynchronous requests.

Example: Grizzly in GlassFish



Source: Please add the source of your data here

Grizzly makes GlassFish FAST!



- Benchmarked Grizzly vs Tomcat 5.5 with an application that contains complex Servlet, JSP and Database transaction.
- The benchmark measures the maximum number of users that the website can handle assuming that 90% of the responses must come back within 2 seconds and that the average think time of the users is 8 seconds.

Agenda

Introduction

Grizzly Framework

Resource Consumption Management Features in GlassFish

OpenESB HTTP SOA features in SJSAS 9.0

Q&A

Resource Management Problem

- What Operating systems have done for decades
  - Fair share or real-time class scheduling
  - Fair access to memory, disk and network resources
  - Isolate process failures, more
- In Enterprise Java Execution Environment
  - How to isolate component failures?
  - How to ensure fair access to Data sources?
  - How to execute all requests within a reasonable time?
  - How to isolate rogue applications (memory leaks, connection leaks, connection hogs,....)

Source: Please add the source of your data here

Some example requirements

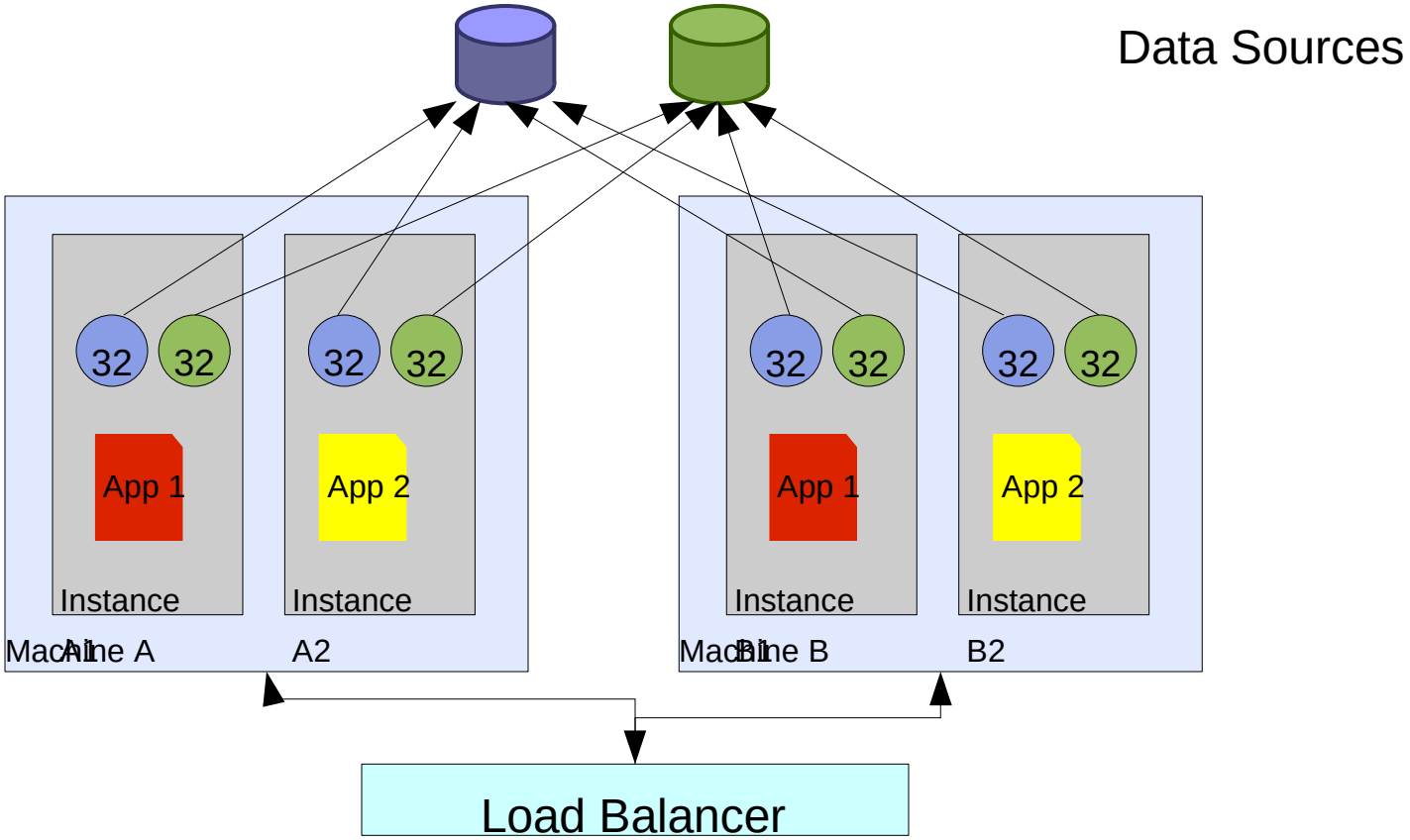
- Application does not consume more than 20% of CPU and 30% of memory
- All response times are under XX milliseconds
- Ensure that a service does not consume resources at the expense of others (ceilings)

Service Isolation is one answer

- Pros
  - Provides fine grained management control
  - Each service is individually tunable for service quality
  - Rogue applications do not hurt (no co-residents)
- Cons
  - Many containers to manage
  - Resource under-utilization is a bigger issue
- Many Administrators prefer consolidation
  - Choice may vary however, based on other criteria

Source: Please add the source of your data here

Typical Isolated Deployment Strategy



Can we do better?

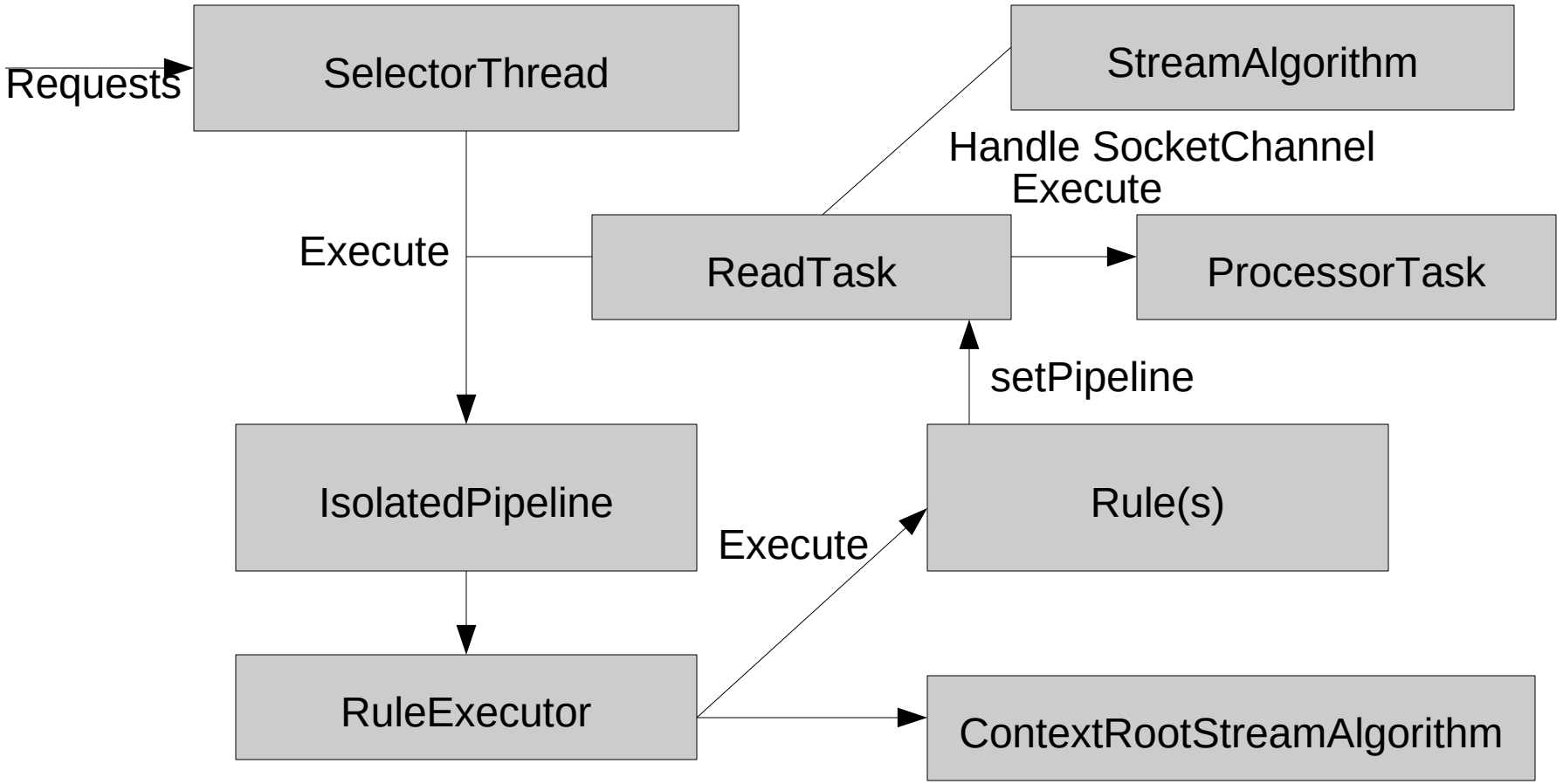
- App Server Containers have “perfect” information
  - Know the component being invoked
  - Know the identity of invoker
  - Know the resource being requested
- Questions of Policy
  - What ? Ceiling (max) and Quota (reserve)
  - When ?
    - Fine grained: Apply policy at every resource request
    - Coarse grained: Apply policy at request dispatch time

Source: Please add the source of your data here

Our project with Grizzly

- Test a coarse grained approach
- Implemented ***ceilings*** and ***quotas*** in Http Request handling path
  - ***Ceiling***: Never allow more than x% of request threads to be allocated for requests to App-1
    - Leave at least (100-x)% of “processing capacity” to other apps
    - Consume “< x%” of resources (like database connections)
  - ***Quota***: If we reserve x% of thread capacity to App-1
    - Better chance that threads that will respond quickly to App-1
    - May deprive or delay other co-resident applications

Example:



Source: Please add the source of your data here

Grizzly Details

- Added a new Pipeline:
  - Intercept the requests until the policy engine has determined the request execution conditions.
- Added the a new StreamAlgorithm:
  - Read the request lines (first 8k bytes) and try to determine the HTTP request context root.

Grizzly Details

- Added a new rule based Policy Engine
  - Rules can be added on the fly. The Rule is responsible for determining which policy apply to the request (context root). More than one rule can be applied to the same request.
  - **The last rule executed decide which Pipeline will be associated with the Task (hence the request).**

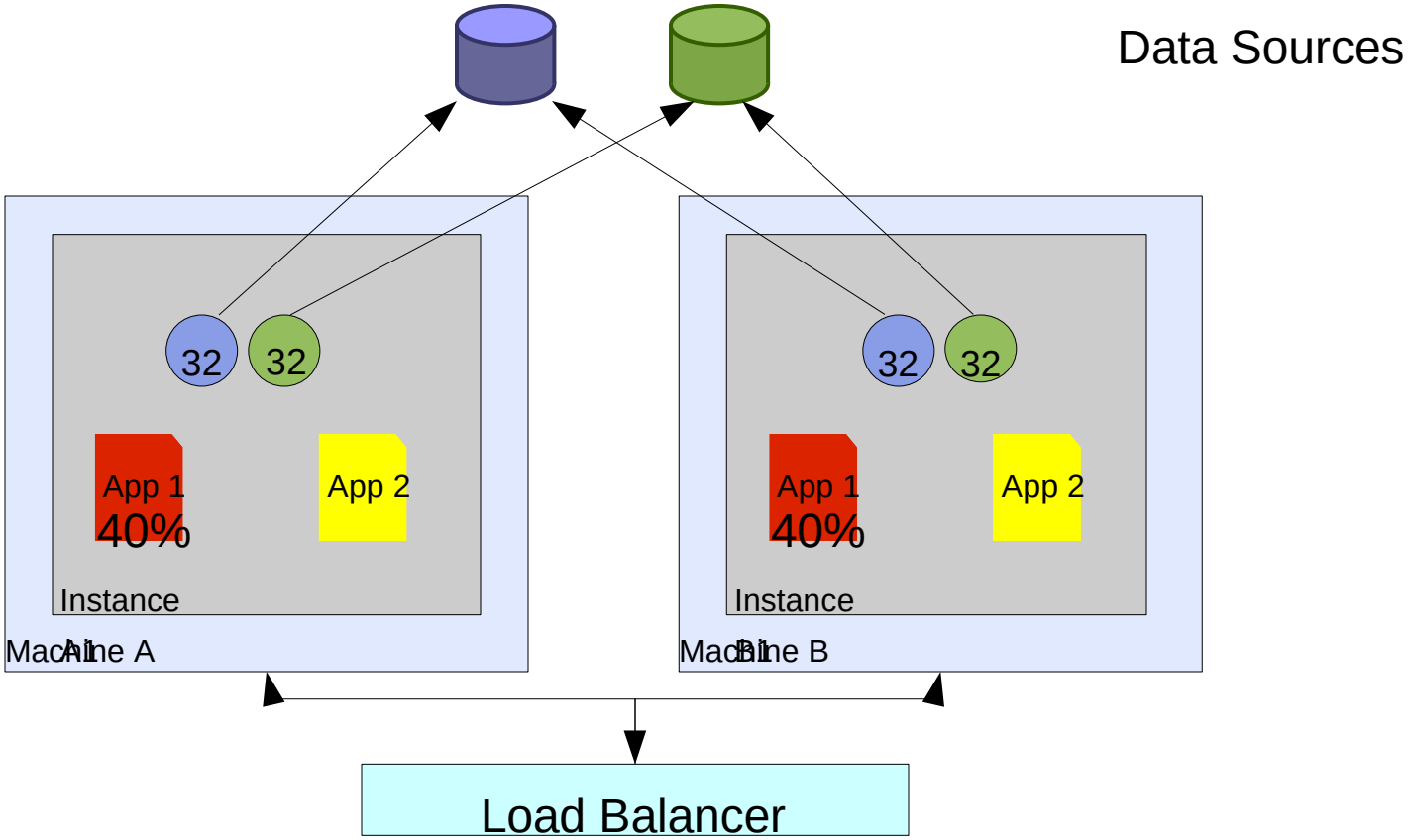
Configuration Example

```
<jvm-options>
      -Dcom.sun.enterprise.web.ara.rules=
       com.sun.enterprise.web.ara.rules.ThreadRatioRule
       com.sun.enterprise.web.ara.rules.HeapRatioRule

      -Dcom.sun.enterprise.web.ara.allocationPolicy=reserve

      -Dcom.sun.enterprise.web.ara.allocationRatio=appA|0.5,appB|0.4
</jvm-options>
```

Consolidated Deployment with ARA



Conclusions

- For similar application types
  - Thread ratios result in roughly proportional CPU utilization and roughly same ratio of resource usage as thread ratio
- For non-similar co-resident it is possible to get the ratios right, after tweaking
- BENEFIT: We allocate fewer resource connections than with perfect isolation
  - Do not have to provision each application for peak

Other Approaches

- At JVM Level
  - Isolates (JSR-121) and Sun MVM Project
  - Services execute as separate Isolates in one VM
  - Resource Management is orthogonal (JSR-284)
  - Not in mainstream JVMs, yet
- Leverage OS Capabilities
  - e.g: Solaris Zones and Resource Management
  - Specify CPU, Disk, Memory and Network quotas per *project* or *zone*
  - Not aware of application or identity

Source: Please add the source of your data here

Agenda

Introduction

Grizzly Framework

Resource Consumption Management

  Features in GlassFish

OpenESB HTTP SOA features in SJSAS 9.0

Q&A

A crash course in OpenESB and JBI

- What is JBI?
  - The JBI 1.0 (JSR 208) specification is an industry-wide initiative to create a standardized integration platform for Java and business applications
  - JBI addresses service-oriented architecture (SOA) needs in integration

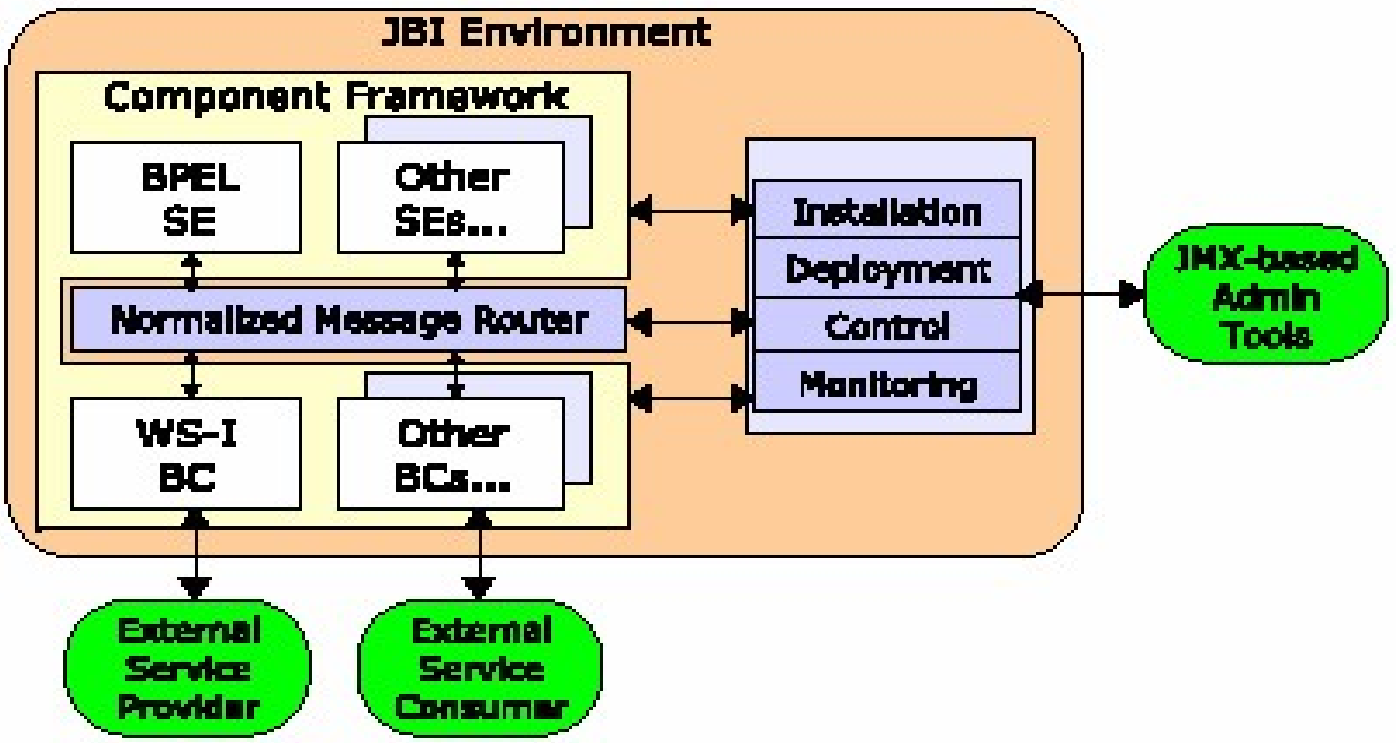
A crash course in OpenESB and JBI (Cont.)

- OpenESB is based on JBI
  - Extends the reference implementation (RI), adding additional binding components, service engines, tools, management and monitoring
    - Available in Java EE 5 SDK SOA Starter Kit Preview http://java.sun.com/javaee/downloads
    - Source code and further information available at http://java.sun.com/integration

JBI Basics

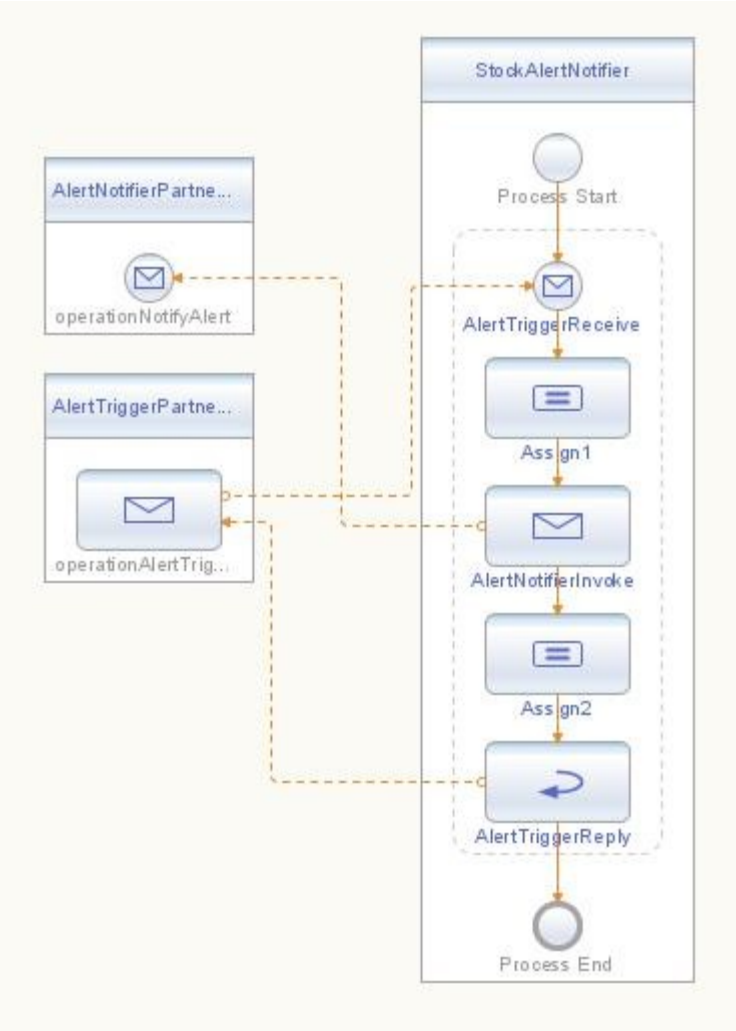
- Messaging based, plug-in architecture
  - Plug-in components into a standard infrastructure; allow them to discover and interoperate with each other
- Key components
  - Service Engines (SE) – pluggable business logic (e.g. BPEL SE, XSLT SE)
  - Binding Components (BC) – pluggable external connectivity (e.g. HTTP SOAP BC, JMS BC, File BC)
  - Normalized message router (NMR) – the messaging based bus through which the components communicate

JBI High Level Architecture



Source: JBI 1.0 (JSR 208) specification

Example Business Process (BP)



- Shows a request/reply web service implementation
- HTTP SOAP BC listens for and handles requests
- BPEL SE executes the BP logic

HTTP SOAP BC

- Has to efficiently support many 1000s of concurrent requests, including request/reply
- Support web services on ports that are not serviced by the application server
  - Makes use of the standard WSDL SOAP extensions to define external communication details
- Implements the WSDL 1.1 SOAP binding and SOAP 1.1
- Follows WS-I 1.0 conventions and adds additional support for non-conforming components

What are some of the challenges?

- Business Processes are user defined
  - Processing time for requests can be considerable and vary widely
- SOA involves orchestration of services
  - typically a business process might invoke several other services and might “wait” for replies (I/O) before it completes / replies itself
- JBI components are loosely coupled
  - Typically they do not share threads
  - They interact through an asynchronous message bus
- => We do **not** want to block a thread per request

Our Approach

- Use embedded Grizzly instances
  - Gives us NIO, non-blocking sockets
  - Comes with all the features such as efficient buffer handling, temporary selectors if the main selector gets overloaded etc.

Our Approach (Cont.)

- Use Grizzly asynchronous extensions
  - Matches up well with the JBI asynchronous message bus; when a reply is received from the NMR we “wake up” the response processing instead of blocking a thread until then
  - We have tested the HTTP SOAP BC with a single inbound thread to service all concurrent requests without an issue

Grizzly Asynchronous Extensions

- Allow for “parking” a request; a type of “continuation” at the request processing level
  - Enable via configuration or programmatically

```
SelectorThread selThread = protocolHandler.selectorThread();
DefaultAsyncHandler asyncHandler = new
com.sun.enterprise.web.connector.grizzly.async.DefaultAsyncHandler();
selThread.setAsyncHandler(asyncHandler);
selThread.setEnableAsyncExecution(true);
```

  - Implement and add an “async” filter

```
com.sun.enterprise.web.connector.grizzly.AsyncFilter filter = new
        com.sun.jbi.httpsoapbc.embedded.JBIGrizzlyAsyncFilter();
asyncHandler.addAsyncFilter(filter);
```

  - Once a reply is available continue response processing

```
AsyncHandler asyncHandler = task.getAsyncHandler();
asyncHandler.handle(asyncProcessorTask);
```

Using an Embedded Grizzly

- Currently Grizzly does not package a facade / helpers for embedded use; either use the glassfish implementations or provide your own
  - Create a connector

```
org.apache.catalina.Connector connector =
    GrizzlyEmbeddedWebContainer.createConnector(address, port, protocol);
BCCoyoteConnector bcCon = (BCCoyoteConnector) connector;
```

  - Create your own adapter implementation if desired

```
org.apache.coyote.Adapter adapter = new GrizzlyRequestProcessor(bcCon);
bcCon.setAdapter(adapter);
```

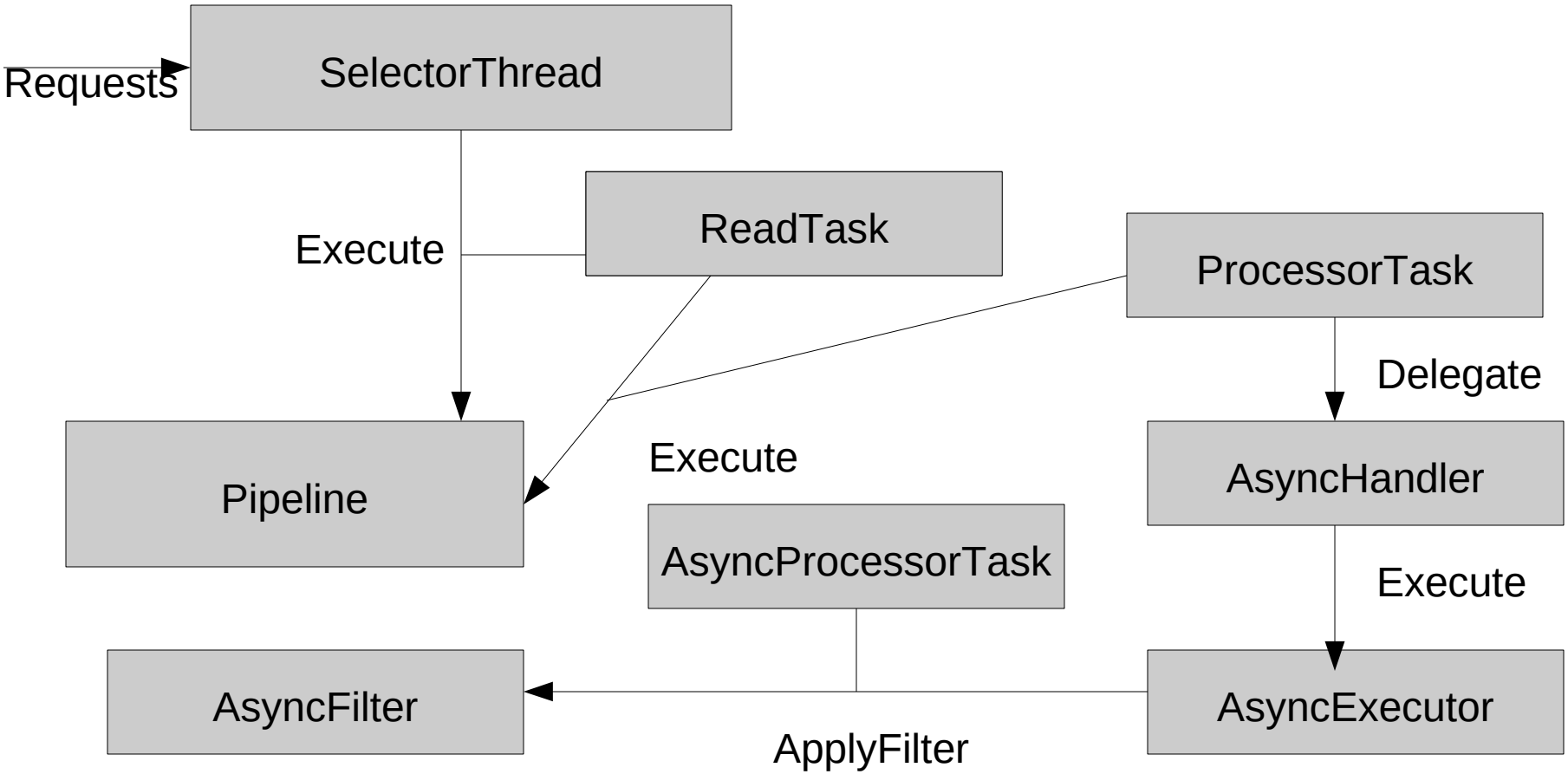
  - Set the protocol handler

```
protocolHandler = new
    com.sun.enterprise.web.connector.grizzly.GrizzlyHttpProtocol();
bcCon.setProtocolHandler(protocolHandler);
```

  - Configure and start the Grizzly connector

```
bcCon.setMaxReadWorkerThreads(3);
bcCon.start();
```

Example:



Source: Please add the source of your data here

Summary

Grizzly Everywhere!

- **Don't miss James Gosling keynote for another Grizzly extension!!**

What's Next

- Grizzly
  - An “embedded” grizzly controller class out of the box to simplify instantiation and configuration
  - Grizzly download out of GlassFish
- JAX-WS / Sun WS-* stack
  - Incorporate NIO non-blocking and allow asynchronous request processing (w/o blocking threads) client and server side

Additional Resources

- Binaries and Sources for OpenESB (including the HTTP SOAP BC) can be downloaded from https://open-esb.dev.java.net/public/downloads.html
- Grizzly available in GlassFish

  https://glassfish.dev.java.net
- Grizzly day to day news

  http://weblogs.java.net/blog/jfarcand/

Q&A

Customizing the “Grizzly” NIO Framework

Jean-Francois Arcand, Staff Engineer
Sreeram Duvur, Senior Staff Engineer
Andreas Egloff, Staff Engineer
Sun Microsystems
http://www.sun.com

BOF-0520