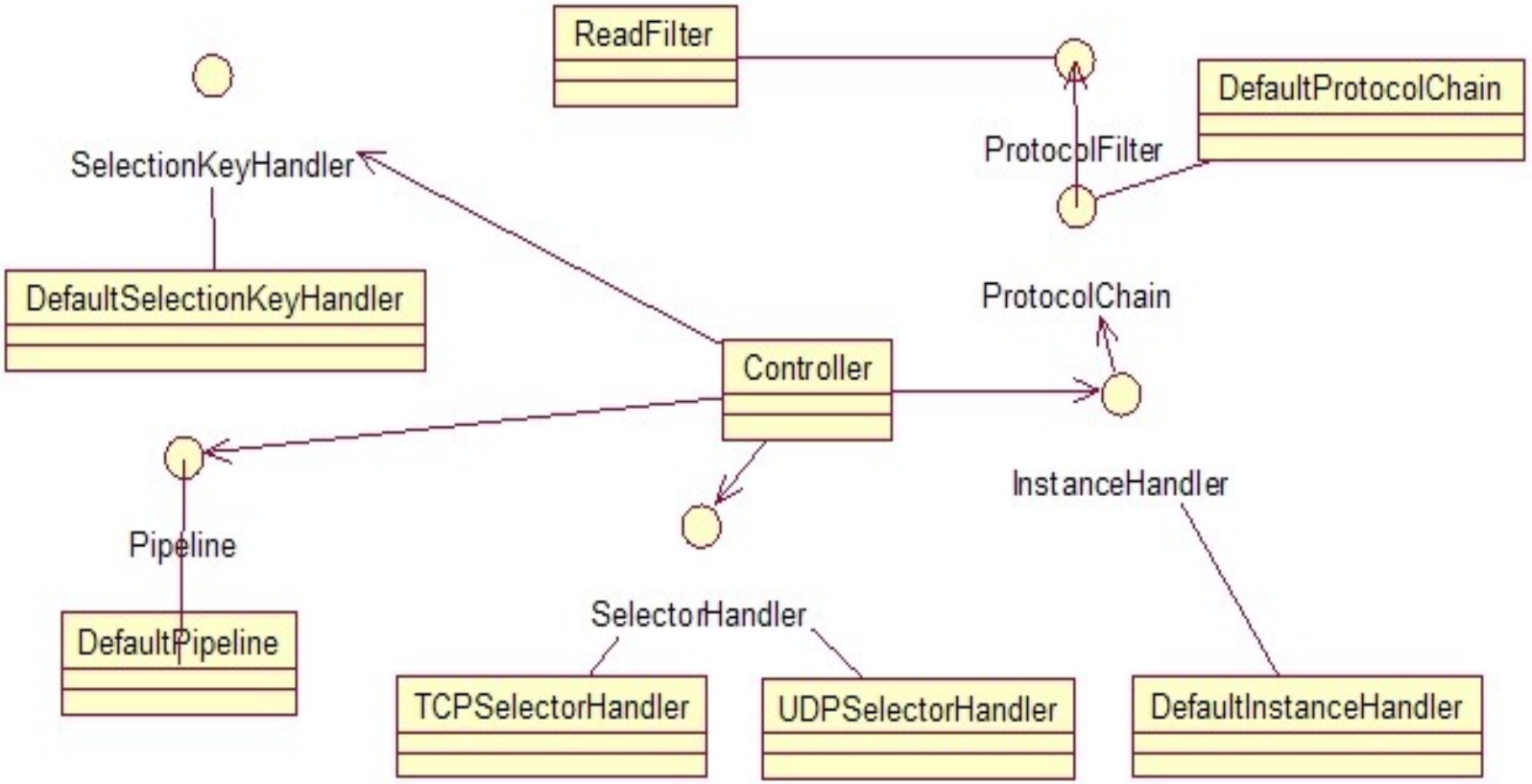
Grizzly 1.5 Architecture Overview

Jeanfrancois Arcand
Sun Microsystems

Agenda

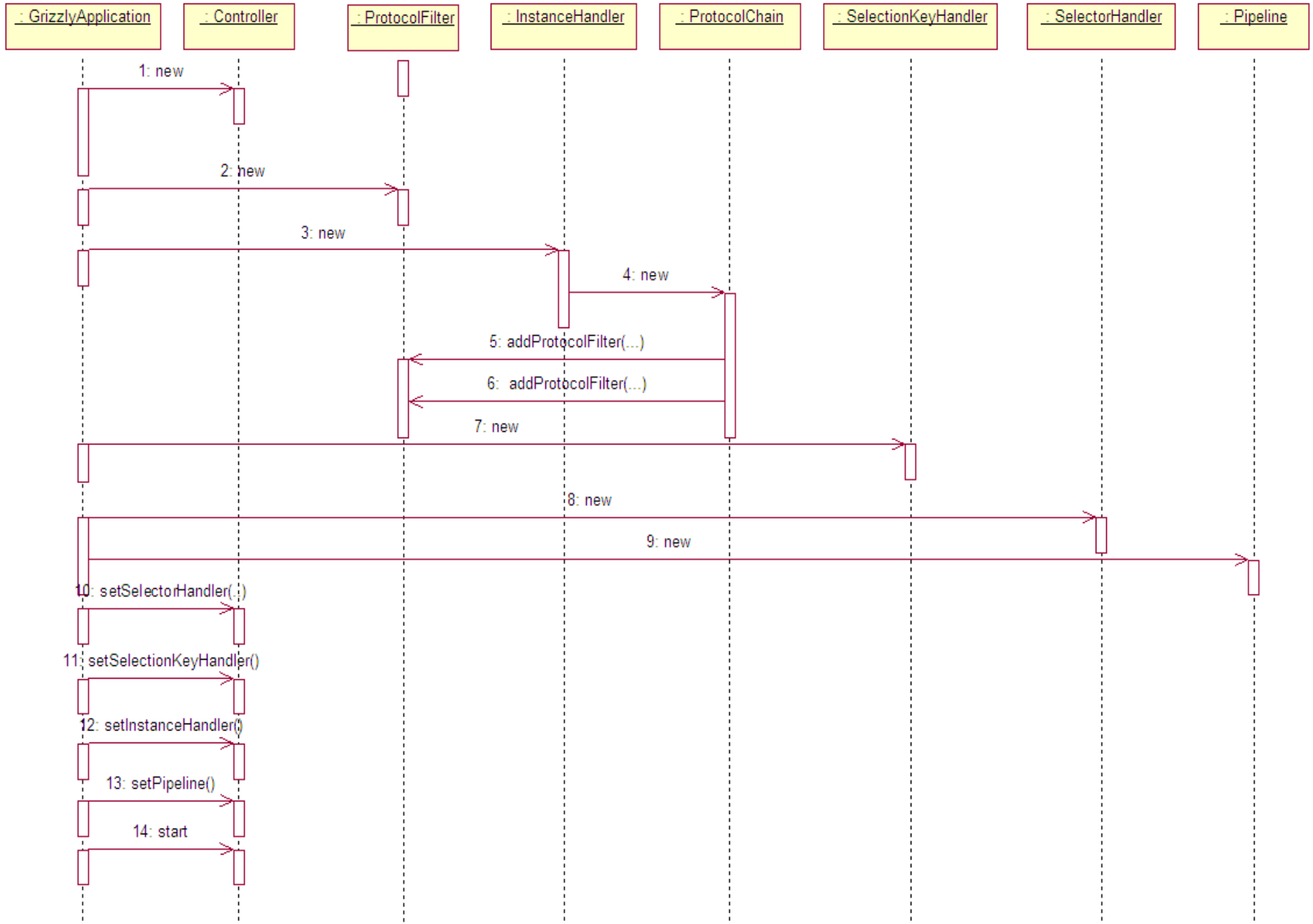
- Introduction
- Grizzly 1.5 UML Diagrams
  - Class Diagram
  - Sequence Diagrams
- Classes description and review
- Summary
- Q&A

Class Diagram



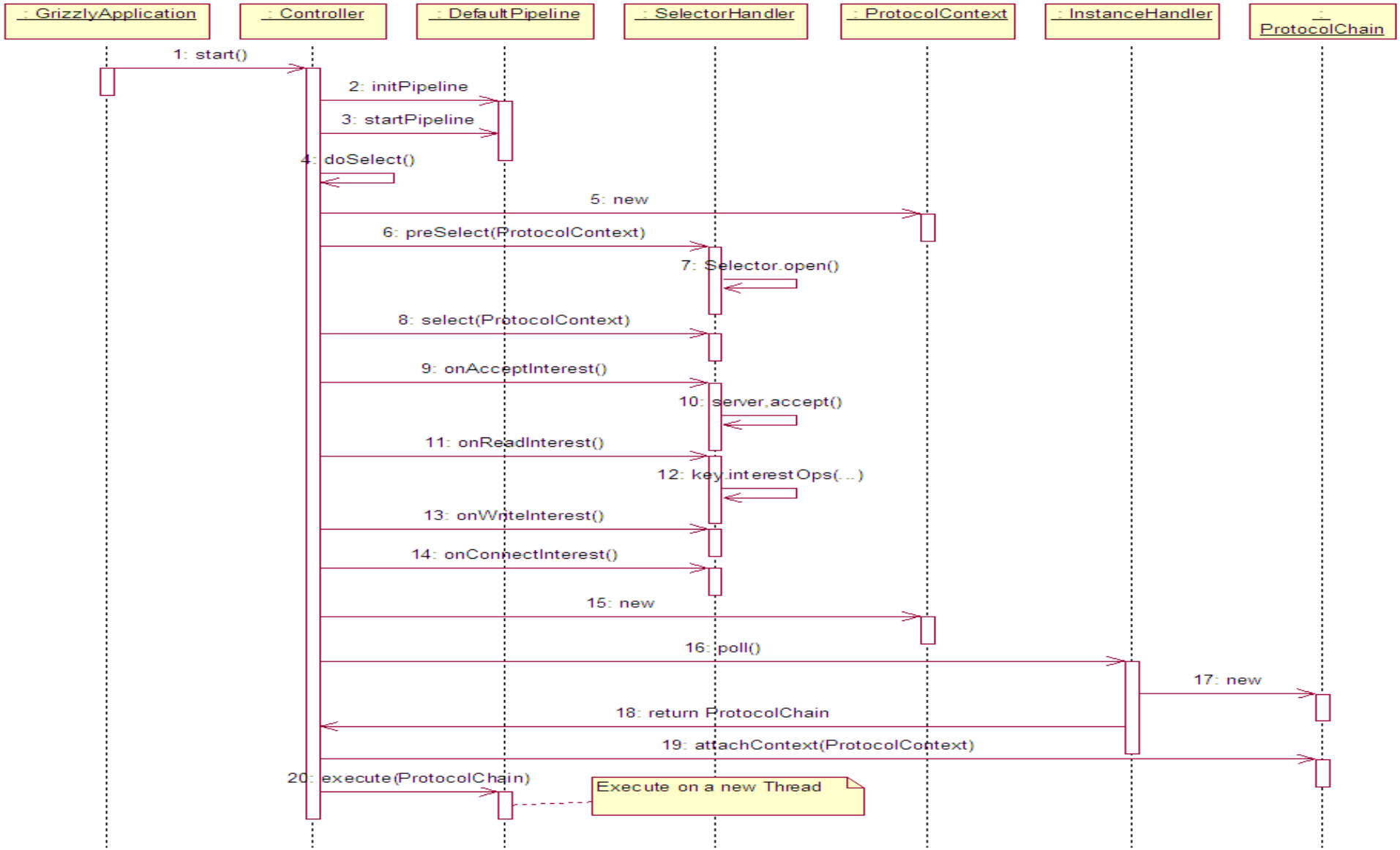
Creating a Controller

JGD



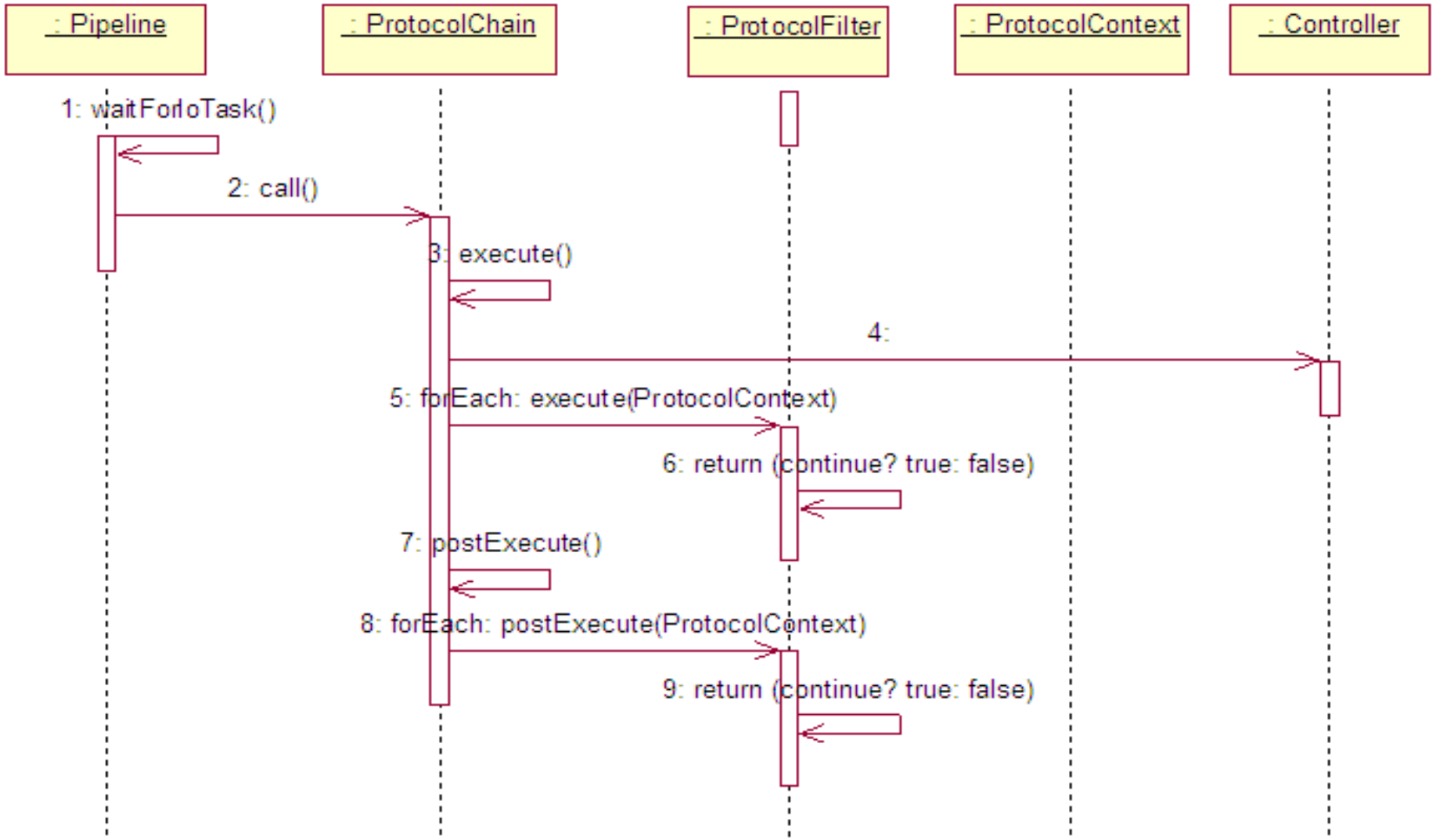
Request Handling

JGD



Worker Thread execution

JGD



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.

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.

SelectorHandler (Cont.)


```
/**
 * This method is garantee to always be called before operation
 * 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 garantee to always be called after operation
 * Selector.select().
 */
public void postSelect(Context controllerCtx)  throws IOException;
```

SelectionKeyHandler

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

SelectionKeyHandler (Cont.)


```
/**
 * 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 are created and cached.
- An InstanceHandler decide if a stateless or statefull ProtocolChain needs to be created.

InstanceHandler (Cont.)


```
/**
 * 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.

Pipeline (Cont.)


```
/**
 * Add an <code>E</code> to be processed by this
<code>Pipeline</code>
 */
public void execute(E task) throws PipelineFullException;
/**
 * Return a <code>E</code> object available in the pipeline.
 */
public E waitForIoTask() ;
```

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".

ProtocolChain (Cont.)

- The API for ProtocolFilter consists of a two methods (execute() and postExecute) which is passed a "protocol context" parameter containing the dynamic state of the computation, and whose return value is a boolean that determines whether or not processing for the current chain has been completed (false), or whether processing should be delegated to the next ProtocolFilter in the chain (true).
- 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.)


```
/**
 * Add a <code>ProtocolFilter</code> to the list.
<code>ProtocolFilter</code>
 * 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

- 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.
- ProtocolFilter implementations should be designed in a thread-safe manner, suitable for inclusion in multiple ProtocolChains that might be processed by different threads simultaneously.

ProtocolFilter (Cont.)

- In general, this implies that ProtocolFilter classes should not maintain state information in instance variables.
- Instead, state information should be maintained via suitable modifications to the attributes of the ProtocolContext that is passed to the execute() and postExecute() methods.

ProtocolFilter (Cont.)


```
/**
 * 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 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 UDPReadFilter());
          protocolChain.addFilter(new ParserFilter());
      }
      return protocolChain;
   }
   con.setSelectorHandler(new UDPSelectorHandler());
});
```

Q&A

Grizzly 1.5 Architecture Overview