The Java ICAP Project User Guide

1.0.0.RC2

Preface

1. ICAP and the lack of java implementations

ICAP (Internet Content Adaptation Protocol) was designed to encapsulate HTTP requests and responses that they can be sent to a ICAP capable server which can either modify or notice the HTTP messages. With this in mind the Protocol is made for HTTP based entry server and proxy servers like Squid to off load message processing.

As seen above ICAP will never be as popular like HTTP. But since HTTP is so popular and the demand for HTTP message processing in Security entities such as entry servers or reverse proxy servers is getting bigger, ICAP has become a protocol that is relevant and important.

The initiator for this project of building a JBoss Netty ICAP codec was the simple fact that there are roughly two ICAP implementations in Java, the problem is that both implementations are bound within a complete server infrastructure. In other words, no JEE environment is supported and therefore is the enterprise usage of ICAP as a server almost not possible. When building such a server there are always demands such as "load balancing", "administration via web-services", "jmx", "gui". These cannot be simply meet by the current implementations.

2. Goals

- Full Java ICAP protocol implementation according to RFC3507.
- JEE and Servlet container implementation capabilities.
- Server and Client support.

3. JBoss Netty (http://www.jboss.org/netty)

In order to achieve the above goals we have decided to implement the ICAP protocol on top of JBoss Netty as a codec. The advantage of doing this is that Netty is already providing a HTTP codec which can be integrated into the ICAP protocol codec. The encapsulated HTTP request and response objects are therefore the Netty HTTP request and response implementations.

Netty can be used as a stand alone server or client and you can integrate it into a servlet container or JEE environment. NIO and OIO sockets and streams are supported out of the box and the entire socket api complexity is abstracted. The performance of Netty is outstanding and is therefore in combination with our ICAP codec a real alternative to c or c++ based solutions.

Release Notes

In this chapter we list all release note in a chronological order. Newest to Oldest.

1.1. 1.0.0.RC2

- Introduced specified date setter and getter methods in the `IcapHeaders` class.
- renamed the getBody() method to getBodyType() in `IcapMessage` in order to have a more specific name. The old getBody() was ambiguous.
- fixed possible NBE in the simple icap server / client example.

1.2. 1.0.0.RC1

- Initial Release Candidate containing all basic funtionality for the ICAP protocol codec.

Usage

The basic usage is identical to all JBoss Netty codecs. You have different Handlers available for message encoding, decoding. In addition to that we also provide handlers that allow the user to abstract from protocol details like the chunked transfer of all message bodies. We recommend to read up on *`JBoss Netty`* in order to successfully use this codec.

2.1. Available Handlers

You have a set of Handlers available that will encode and decode ICAP messages. These are the basic handlers which are required in order to use the ICAP codec. Since ICAP encapsulated HTTP bodies are always chunked you can use the aggregation and separation handlers to abstract from this protocol overhead. alongside the source code are examples that suggest how to use the provided handlers. The examples can be found in the package: `ch.mimo.netty.example.icap.*` All examples show how to create a server and client handler pipeline. It is recommended to construct client and server pipelines according to these examples.

The simple example shows how to best use the provided handlers. It implicitly uses the aggregation and separation handlers which take care of all message body transfer details. The below depicted pipelines are client and server pipelines:

Note that at the end of each Pipeline is either a Server or Client handler that represents the so called Hub of the pipeline. This handler is responsible to receive messages process them and send the response back into the pipeline.

2.2. Client handler pipeline

A straight forward client handler pipeline consists of:

- *`IcapRequestEncoder`* The Encoder will take a *`IcapRequest`* or *`IcapChunk`* and transfer them into a ASCII string that represents the ICAP protocol.
- *`IcapResponseDecoder`* The Decoder produces an ICAP protocol ASCII string as input and creates *`IcapRequest`* and *`IcapChunk`* instances.

```
@Override
 public ChannelPipeline getPipeline() throws Exception {
  ChannelPipeline pipeline = pipeline();
  pipeline.addLast("encoder",new IcapRequestEncoder());
  pipeline.addLast("decoder",new IcapResponseDecoder());
  pipeline.addLast("handler",new IcapClientHandler());
  return pipeline;
 }
```

You also have the possibility to abstract from the tedious message body chunk handling and add two additional handlers which will take care of the chunked message body. It is important to understand that in this scenario (if you plan to send a message body) you have to attach the message body to the respective HTTP request or response.

- `IcapChunkSeparator` This handler is useful when you create an `IcapRequest` instance containing HTTP request/response and a body encapsulated with one of the http messages. The responsibility of this handler is then to extract the message body and send it as chunks to the receiving side.
- `IcapChunkAggregator` This handler is useful when you don't want to handle individual `IcapChunk` instances but rather receive a combined `IcapResponse`

```
@Override
 public ChannelPipeline getPipeline() throws Exception {
  ChannelPipeline pipeline = pipeline();
  pipeline.addLast("encoder",new IcapRequestEncoder());
  pipeline.addLast("chunkSeparator",new IcapChunkSeparator(4096));
  pipeline.addLast("decoder",new IcapResponseDecoder());
  pipeline.addLast("chunkAggregator",new IcapChunkAggregator(4096));
  pipeline.addLast("handler",new IcapClientHandler());
  return pipeline;
 }
```

2.3. Server handler pipeline

A straight forward Server pipeline consists of:

- `IcapRequestDecoder` The Decoder produces an ICAP protocol ASCII string as input and creates `IcapRequest` and `IcapChunk` instances.
- `IcapResponseEncoder` The Encoder will take a `IcapResponse` or `IcapChunk` and transfer them into a ASCII string that represents the ICAP protocol.

```
@Override
   public ChannelPipeline getPipeline() throws Exception {
  ChannelPipeline pipeline = pipeline();
  pipeline.addLast("decoder",new IcapRequestDecoder());
  pipeline.addLast("encoder",new IcapResponseEncoder());
  pipeline.addLast("handler",new IcapServerHandler());
  return pipeline;
   }
```

Analog to the Client example you also have the possibility to abstract from the tedious message body chunk handling and add two additional handlers which will take care of the chunked message body. It is important to understand that in this scenario (if you plan to send a message body) you have to attach the message body to the respective HTTP request or response.

- `IcapChunkSeparator` This handler is useful when you create an `IcapRequest` instance containing HTTP request/response and a body encapsulated with one of the HTTP messages. The responsibility of this handler is then to extract the message body and send it as chunks to the receiving side.
- `IcapChunkAggregator` This handler is useful when you don't want to handle individual `IcapChunk` instances but rather receive a combined `IcapResponse`

```
@Override
  public ChannelPipeline getPipeline() throws Exception {
 ChannelPipeline pipeline = pipeline();
 pipeline.addLast("decoder",new IcapRequestDecoder());
 pipeline.addLast("chunkAggregator",new IcapChunkAggregator(4096));
 pipeline.addLast("encoder",new IcapResponseEncoder());
 pipeline.addLast("chunkSeparator",new IcapChunkSeparator(4096));
 pipeline.addLast("handler",new IcapServerHandler());
 return pipeline;
  }
```

2.4. Preview

The ICAP protocol supports a special scenario whereby the client is enabled to send a pre-defined portion of the HTTP message body to the server and wait for further instructions. There server can the ask for more data by responding with `100 Continue` which tells the client to send the rest of the message body, or with `204 No Content` (If the client allows this!) and the client knows ok no modification was made on the request. The Server is also capable of sending a normal `200 OK` message back to the client. This circumstance complicates the protocol handling for a server or client. We have therefore made a simple example that can be found under `ch.mimo.netty.example.icap.preview.*` which will help to understand the inner workings of such a preview request for client and server.

In order to detect whether a `IcapRequest` or `IcapChunk` belongs to a preview message we have added getters on each object that indicate if a request or chunk participates on a preview interaction.

```
/**
  * @return whether this message is a preview of the actual message.
  */
  boolean isPreviewMessage();
```

```
/**
  * @return boolean true if this chunk is preview.
  */
 boolean isPreviewChunk();
```

The best way to handler preview requests is configuring a *`IcapChunkAggregator`* into your pipeline. This will guarantee that once you have sent the `100 Continue` response you will get the same *`IcapRequest`* instance containing the rest of the body that was not sent in the preview mode.

2.5. Abstraction handlers and the NIO threading paradigm

You cannot use the NIO (non blocking) threading paradigm togehther with the two abstaction handlers *`IcapChunkAggregator`* and *`IcapChunkSeparator`*. Both handlers are statefull and consider to be run by the same thread for one request only. As soon you will switch to NIO message content will be exchanged between requests. In other words the two handlers are not threadsafe. The roadmap contains the enablement of both handlers to be NIO capable.