Compression and decompression of messages using WebSphere Message Broker V6

Tushita Jain
Shravan K Kudikala

November 14, 2007

This article shows you how use the Java Compute Node and one of the java.util.zip package to compress and decompress messages traversing through WebSphere Message Broker V6.

Introduction

IBM® WebSphere® Message Broker (hereafter called Message Broker) transforms and enriches in-flight information to provide a further level of intermediation between applications that use different message structures and formats. Through Message Broker, you can manipulate data using ESQL, Java™, and C. This article describes a case study to compress and decompress messages using WebSphere Message Broker. It includes a problem analysis, solution design, implementation of the solution, and an explanation of the example flows.

Description of problem and solution

While Message Broker can process large messages, they can consume significant bandwidth outside Message Broker and cause delays on the transport layer, diminishing the benefits from faster processing by Message Broker. You can mitigate this problem by compressing and decompressing messages before and after they are processed by message flows.

There are numerous ways to compress or decompress data by using C or Java and then adding them as plug-ins in Message Broker. This article uses Java Compute Nodes and the widely used tool GZip. You can specify a different algorithm and define that implementation to Message Broker. The algorithm to compress and decompress should be the same.

Implementing the solution

GZip is implemented using two Java classes:

- `java.util.zip.GZipOutputStream`
- `java.util.zip.GZipInputStream`

These classes work on bitstreams. In Message Broker, a BLOB message represents a bitstream. Therefore, if the input message is in a non-BLOB domain, it must be converted to a BLOB

message before compression and decompression. In our implementation, while compressing and decompressing a message, the headers are retained and only the message body is compressed and decompressed.

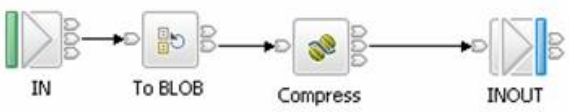
During compression, a `ByteArrayOutStream` object is attached to a `GZipOutputStream` to store the bytestream after compression. The write method accepts the uncompressed bytestream and writes to the `ByteArrayOutputStream` object attached to the `GZipOutputStream` object. Then the compressed bitstream is reattached to the message.

During decompression, the bitstream is extracted from the BLOB message and assigned to a `GZipInputStream` object. The `InflaterInputStreamread` method reads the unzipped bitstream into a buffer. The `write` method of `ByteArrayOutputStream` eventually writes the buffer to an output stream, and the converted message can then be attached to the body.

The solution uses two message flows to illustrate compression and decompression of a message. The domain used is XML, but you can choose the domain based on the incoming message format.

In the compression flow, the "To BLOB" `ResetContentDescriptor` node converts the incoming XML message to a BLOB message. The Java Compute Node `Compress` archives and compresses the message by first detaching the message body, compressing it using GZip, and then reattaching it to the message assembly before propagating it to the next node. Then the message is put to the output queue `INOUT`:

Figure 1



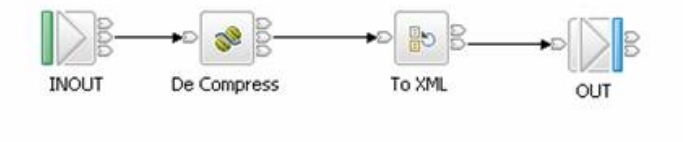
The corresponding code in the `Compress` node is:

```
/* This JCN allows to compress the message body.*/
public class Compress extends MbJavaComputeNode
{
  public void evaluate(MbMessageAssembly inAssembly) throws MbException
  {
    /* Get the output terminal */
    MbOutputTerminal Success = getOutputTerminal("out");
    /* Get the failure terminal */
    MbOutputTerminal Failure = getOutputTerminal("failure");
    /* Get the message tree from the input message assembly */
    MbMessage inMessage = inAssembly.getMessage();
    /* The input message is always read-only, so we need to copy/create a new message
       out of the incoming message to modify it. */
    MbMessage outMessage = new MbMessage(inMessage);
    MbElement msgBody;
    byte []msgByteStreamIn = null;    // ByteStream before compression
    byte []msgByteStreamOut = null;   // ByteStream after compression
    try
    {
      /* The last element of the message is always the messagebody. We compress only
         the message body and not the headers, if any */
```

```
      msgBody = outMessage.getRootElement().getLastChild();
      /* Message body is the last child of Root*/
      msgBody.detach();  //detach the message body
      // Returns bit stream representation of the element. This method causes the parser
      // associated with the element to serialize the element and all children. This
      // method can only be called on message body - the last child of the message root.
      msgByteStreamIn = msgBody.toBitstream("","","",0,0,0);
      ByteArrayOutputStream bytesOut = new ByteArrayOutputStream();
      /* Create the compressed message */
      GZIPOutputStream gzipOutputStream = new GZIPOutputStream(bytesOut);
      gzipOutputStream.write(msgByteStreamIn);
      gzipOutputStream.close();
      msgByteStreamOut = bytesOut.toByteArray();
      // After compression, the resultant bitstream is BLOB.
      // Next, it has to be reattached to the output message.
      msgBody = outMessage.getRootElement().createElementAsLastChildFromBitstream
          (msgByteStreamOut,MbBLOB.PARSER_NAME,"","","",0,0,0);
      bytesOut.close();
      // ----------------------------------------------------------------------------
      // Build the output message assembly before propagating the message.
      MbMessageAssembly outAssembly = new MbMessageAssembly(inAssembly,outMessage);
      // The following should only be changed if not propagating message to 'out' terminal
      Success.propagate(outAssembly);
    }
    catch(IOException e) {
      e.printStackTrace();
      }
    catch(MbException e) {
      throw e;
    /* The broker will ensure to send message to failure terminal if it is connected. */
    }
    finally {
      // clear the outMessage
      outMessage.clearMessage();
    }
  }
}
```

The decompression flow does the converse of the compression flow. It accepts the incoming BLOB message, detaches the message body, decompresses it, and attaches it again to the message assembly. Next, a `ResetContentDescriptor` node "To XML" converts the message back to an XML message to be put to the output queue `OUT`.

Figure 2



The corresponding code in the `Decompress` node is:

```
/* This JCN allows you to decompress the message body */
public class Decompress extends MbJavaComputeNode
{
  public void evaluate(MbMessageAssembly inAssembly) throws MbException
  {
    MbOutputTerminal Success = getOutputTerminal("out");
    MbOutputTerminal Failure = getOutputTerminal("failure");
```

```
    MbMessage inMessage = inAssembly.getMessage();
    MbMessage outMessage = new MbMessage(inMessage);
    // create new message
    MbMessageAssembly outAssembly = new MbMessageAssembly(inAssembly,outMessage);
    MbElement msgBody;
    byte []msgByteStreamIn;   // ByteStream before compression
    byte []msgByteStreamOut;  // ByteStream after compression
    InflaterInputStream infInputStream = null;
    try {
      // Message body is the last child of Root
      msgBody = outMessage.getRootElement().getLastChild();
      msgBody.detach();  //detach the message body
      // Returns the bit stream representation of the element. This method causes
      // the parser associated with the element to serialize the element and all
      // its children. This method can only be called on the message body, i.e.
      // the last child of the message root.
      msgByteStreamIn = msgBody.toBitstream("","","",0,0,0);
      infInputStream = new GZIPInputStream(new ByteArrayInputStream(msgByteStreamIn));
      ByteArrayOutputStream bytesOut = new ByteArrayOutputStream(msgByteStreamIn.length);
      byte[] tempBuffer = new byte[msgByteStreamIn.length];
      int numBytesRead = 0;
      while (numBytesRead != -1) {
      numBytesRead = infInputStream.read(tempBuffer, 0, msgByteStreamIn.length);
        if (numBytesRead != -1) {
          bytesOut.write(tempBuffer, 0, numBytesRead);
        }
      }
      msgByteStreamOut = bytesOut.toByteArray();
      infInputStream.close();
      bytesOut.close();
      // After decompression, the resultant bitstream should be converted to
      // BLOB since we do not have a PARSER class corresponding to MRM.
      // Next, it has to be reattached to the output message.
      msgBody = outMessage.getRootElement().createElementAsLastChildFromBitstream
          (msgByteStreamOut,MbBLOB.PARSER_NAME,"","","",0,0,0);
      Success.propagate(outAssembly);
    }
    catch (IOException e){
      e.printStackTrace();
    }
    catch (MbException e){
      throw e;
    }
    finally {
      // clear the outMessage
      outMessage.clearMessage();
    }
  }
}
```

Using the `ResetContentDescription` node causes the entire message body to be re-parsed according to the specified new domain., which can slow down performance.

Attached is the project Interchange zip file containing the four projects, `javaCompress, javaCompressJava, javaDecompress` and `javaDecompressJava`.

Downloadable resources

Description	Name	Size
Code sample	Projects.zip	8 KB

Related topics

- Compressing and decompressing data using Java
  This article from the Sun Developer Network describes the APIs that you can use to compress and decompress data from within your applications, with code samples to show how to use the java.util.zip package to compress and decompress data.
- Compress data streams in Java with GZIP and Zip
  This article from builderau.com shows you how to compress data streams with GZIP and Zip data formats.
- Java 2 Platform, Standard Edition, V1.4.2 API Specification
  Official API specification from Sun.
- WebSphere Message Broker product page
  Product descriptions, product news, training information, support information, and more.
- WebSphere Message Broker information center
  A single Eclipse-based Web portal to all WebSphere Message Broker V6 documentation, with conceptual, task, and reference information on installing, configuring, and using your WebSphere Message Broker environment.
- WebSphere Message Broker documentation library
  WebSphere Message Broker specifications and manuals.