Inspection Document

Version 1.0

Alberto Mario Pirovano Alessandro Vetere

January 5, 2016

POLITECNICO
MILANO 1863

Contents

1 Assigned Classes **3**
1.1 Class Name and Pattern Explanation . . . 3
1.2 Class Code . . . 3

2 Functional Role of Assigned Classes **18**
2.1 ORB Middleware Actors Overview . . . 18
2.2 IIOPSSLSocketFactory functionalities . . . 18
2.3 References . . . 20

3 Issues Found **22**
3.1 Code Inspection Checklist . . . 22
3.1.1 Naming Conventions . . . 22
3.1.2 Indention . . . 22
3.1.3 Braces . . . 22
3.1.4 File Organization . . . 23
3.1.5 Wrapping Lines . . . 23
3.1.6 Comments . . . 23
3.1.7 Java Source Files . . . 23
3.1.8 Package and Import Statements . . . 24
3.1.9 Class and Interface Declarations . . . 24
3.1.10 Initialization and Declarations . . . 24
3.1.11 Method Calls . . . 25
3.1.12 Arrays . . . 25
3.1.13 Object Comparison . . . 25
3.1.14 Output Format . . . 25
3.1.15 Computation, Comparisons and Assignments . . . 26
3.1.16 Exceptions . . . 26
3.1.17 Flow of Control . . . 26
3.1.18 Files . . . 26
3.2 Class Issues . . . 27
3.2.1 Naming Conventions . . . 27
3.2.2 Comments . . . 28
3.2.3 Java Source Files . . . 29
3.2.4 Class and Interface Declarations . . . 30
3.2.5 Initialization and Declarations . . . 30
3.3 Method Issues . . . 30
3.3.1 Issues in createServerSocket . . . 31
3.3.1.1 Indention . . . 31
3.3.1.2 File Organization . . . 32
3.3.1.3 Wrapping Lines . . . 32
3.3.1.4 Comments . . . 32
3.3.1.5 Initialization and Declarations . . . 33
3.3.1.6 Computation, Comparisons and Assignments . . . 33
3.3.2 Issues in createSocket . . . 33

3.3.2.1 Indention 34
3.3.2.2 File Organization 35
3.3.2.3 Wrapping Lines 35
3.3.2.4 Comments 35
3.3.2.5 Initialization and Declarations 35
3.3.2.6 Output Format 36
3.3.2.7 Computation, Comparisons and Assignments . . 36
3.3.2.8 Exceptions 36
3.3.3 Issues in createSSLServerSocket 37
3.3.3.1 Naming Conventions 38
3.3.3.2 Indention 39
3.3.3.3 File Organization 39
3.3.3.4 Wrapping Lines 39
3.3.3.5 Comments 40
3.3.3.6 Initialization and Declarations 40
3.3.3.7 Output Format 41
3.3.3.8 Exceptions 41
3.3.4 Issues in createSSLSocket 41
3.3.4.1 Naming Conventions 42
3.3.4.2 Indention 43
3.3.4.3 File Organization 43
3.3.4.4 Wrapping Lines 43
3.3.4.5 Comments 43
3.3.4.6 Initialization and Declarations 43
3.3.4.7 Output Format 44
3.3.4.8 Exceptions 44

4 Other Problems **45**
4.1 Nagle's Algorithm Disabling 45
4.2 Secure Socket Creation 45
4.3 Secure Server Socket Creation 45
4.4 Class Context Dependency 45
4.5 No Generics Used 46

5 Appendix **47**
5.1 Tools Used 47
5.2 Hours Of Work 47
5.3 Revision History 47

1 Assigned Classes

All the code presented in this document is taken from the **revision 64219** of **GlassFish 4.1.1**.

1.1 Class Name and Pattern Explanation

The class we have been assigned is named **IIOPSSLSocketFactory**. It is included in the **org.glassfish.enterprise.iiop.impl** package. **IIOPSSLSocketFactory** implements the **ORBSocketFactory** interface, which is part of the **CORBA** (Common Object Request Broker Architecture) standard declined to **Java Enterprise Edition**. The **ORBSocketFactory** interface is included in the **com.sun.corba.ee.spi.transport** package, and is an interface that abstracts some parts of the **ORB** (Object Request Broker) middleware related to sockets creation. As the name suggests, the **ORBSocketFactory** interface is a **Factory** of **Sockets** for the **ORB** middleware. The **IIOP** (Internet Inter-ORB Protocol, a concrete protocol) is an implementation of the **GIOP** (General Inter-ORB Protocol, an abstract protocol) that **ORB**s use to communicate over the Internet, and provides a mapping between **GIOP** messages and the **TCP/IP** layer. Our class is therefore a **Factory** of **SSLSockets** for **IIOP**, and is an **Implementation** of the relevant part of the **IIOP** included in the **Enterprise** facilities of the **GlassFish** server.

1.2 Class Code

For reader's convenience, the whole content of the **IIOPSSLSocketFactory** Java class source file is reported below.

```
/*
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.
 *
 * Copyright (c) 1997-2012 Oracle and/or its affiliates. All rights
    reserved.
 *
 * The contents of this file are subject to the terms of either the GNU
 * General Public License Version 2 only ("GPL") or the Common
    Development
 * and Distribution License("CDDL") (collectively, the "License"). You
 * may not use this file except in compliance with the License. You can
 * obtain a copy of the License at
 * https://glassfish.dev.java.net/public/CDDL+GPL_1_1.html
 * or packager/legal/LICENSE.txt. See the License for the specific
 * language governing permissions and limitations under the License.
 *
 * When distributing the software, include this License Header Notice in
    each
 * file and include the License file at packager/legal/LICENSE.txt.
 *
```

```java
 * GPL Classpath Exception:
 * Oracle designates this particular file as subject to the "Classpath"
 * exception as provided by Oracle in the GPL Version 2 section of the
   License
 * file that accompanied this code.
 *
 * Modifications:
 * If applicable, add the following below the License Header, with the
   fields
 * enclosed by brackets [] replaced by your own identifying information:
 * "Portions Copyright [year] [name of copyright owner]"
 *
 * Contributor(s):
 * If you wish your version of this file to be governed by only the CDDL
   or
 * only the GPL Version 2, indicate your decision by adding
   "[Contributor]
 * elects to include this software in this distribution under the [CDDL
   or GPL
 * Version 2] license." If you don't indicate a single choice of
   license, a
 * recipient has the option to distribute your version of this file under
 * either the CDDL, the GPL Version 2 or to extend the choice of license
   to
 * its licensees as provided above. However, if you add GPL Version 2
   code
 * and therefore, elected the GPL Version 2 license, then the option
   applies
 * only if the new code is made subject to such option by the copyright
 * holder.
 */

package org.glassfish.enterprise.iiop.impl;

import com.sun.corba.ee.impl.misc.ORBUtility;
import com.sun.corba.ee.spi.transport.Acceptor;
import java.util.Hashtable;
import java.util.Map;
import java.util.logging.Logger;
import javax.net.ssl.SSLContext;
import com.sun.corba.ee.spi.orb.ORB;
import com.sun.corba.ee.spi.misc.ORBConstants;
import com.sun.corba.ee.spi.transport.ORBSocketFactory;
import com.sun.enterprise.config.serverbeans.Config;
import org.glassfish.orb.admin.config.IiopListener;
import org.glassfish.orb.admin.config.IiopService;
import org.glassfish.grizzly.config.dom.Ssl;
import com.sun.logging.LogDomains;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.List;
import java.util.StringTokenizer;
import java.util.logging.Level;
import javax.net.ssl.KeyManager;
import javax.net.ssl.SSLServerSocket;
import javax.net.ssl.SSLServerSocketFactory;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
import org.glassfish.api.admin.ProcessEnvironment;
import org.glassfish.api.admin.ProcessEnvironment.ProcessType;
import org.glassfish.internal.api.Globals;
import org.glassfish.security.common.CipherInfo;
import org.glassfish.enterprise.iiop.api.IIOPSSLUtil;
import com.sun.enterprise.security.integration.AppClientSSL;
import org.glassfish.api.admin.ServerEnvironment;


/**
 * This is socket factory used to create either plain sockets or SSL
 * sockets based on the target's policies and the client policies.
 * @author Vivek Nagar
 * @author Shing Wai Chan
 */
public class IIOPSSLSocketFactory implements ORBSocketFactory
{
    private static final Logger _logger = LogDomains.getLogger(
        IIOPSSLSocketFactory.class, LogDomains.CORBA_LOGGER);

    private static final String TLS = "TLS";
    private static final String SSL3 = "SSLv3";
    private static final String SSL2 = "SSLv2";
    private static final String SSL = "SSL";
    private static final String SSL_MUTUALAUTH = "SSL_MUTUALAUTH";
    private static final String PERSISTENT_SSL = "PERSISTENT_SSL";

    private static final int BACKLOG = 50;

    //private static SecureRandom sr = null;

    /* this is stored for the Server side of SSL Connections.
     * Note: There will be only a port per iiop listener and a
         corresponding
     * ctx for that port
     */
    /*
     * @todo provide an interface to the admin, so that whenever a
         iiop-listener
     * is added / removed, we modify the hashtable,
     */
    private Map portToSSLInfo = new Hashtable();
    /* this is stored for the client side of SSL Connections.
     * Note: There will be only 1 ctx for the client side, as we will
         reuse the
     * ctx for all SSL connections
     */
    private SSLInfo clientSslInfo = null;

    private ORB orb;

    /**
     * Constructs an <code>IIOPSSLSocketFactory</code>
     */
    public IIOPSSLSocketFactory() {
        try {

            ProcessEnvironment penv = null;
            ProcessType processType = null;
            boolean notServerOrACC = Globals.getDefaultHabitat() == null
                ? true : false;
            if (!notServerOrACC) {
                penv = Globals.get(ProcessEnvironment.class);
                processType = penv.getProcessType();
            }
            //if (Switch.getSwitch().getContainerType() ==
                Switch.EJBWEB_CONTAINER) {
            if((processType != null) && (processType.isServer())) {
                //this is the EJB container
                        Config conf =
                            Globals.getDefaultHabitat().getService(Config.class,
                        ServerEnvironment.DEFAULT_INSTANCE_NAME);
                IiopService iiopBean
                    =conf.getExtensionByType(IiopService.class);
                List<IiopListener> iiopListeners =
                    iiopBean.getIiopListener();
                for (IiopListener listener : iiopListeners) {
                    Ssl ssl = listener.getSsl();
                    SSLInfo sslInfo = null;
                    boolean securityEnabled =
                        Boolean.valueOf(listener.getSecurityEnabled());

                    if (securityEnabled) {
                        if (ssl != null) {
                            boolean ssl2Enabled =
                                Boolean.valueOf(ssl.getSsl2Enabled());
                            boolean tlsEnabled =
                                Boolean.valueOf(ssl.getTlsEnabled());
                            boolean ssl3Enabled =
                                Boolean.valueOf(ssl.getSsl3Enabled());
                            sslInfo = init(ssl.getCertNickname(),
                                    ssl2Enabled, ssl.getSsl2Ciphers(),
                                    ssl3Enabled, ssl.getSsl3TlsCiphers(),
                                    tlsEnabled);
                        } else {
                            sslInfo = getDefaultSslInfo();
                        }
                        portToSSLInfo.put(
                            new Integer(listener.getPort()), sslInfo);
                    }
                }

                if (iiopBean.getSslClientConfig() != null &&
                        /*iiopBean.getSslClientConfig().isEnabled()*/
                        iiopBean.getSslClientConfig().getSsl() != null) {
                    Ssl outboundSsl =
                        iiopBean.getSslClientConfig().getSsl();
                    if (outboundSsl != null) {
                        boolean ssl2Enabled =
                            Boolean.valueOf(outboundSsl.getSsl2Enabled());
                        boolean ssl3Enabled =
                            Boolean.valueOf(outboundSsl.getSsl3Enabled());
                        boolean tlsEnabled =
                            Boolean.valueOf(outboundSsl.getTlsEnabled());
                        clientSslInfo = init(outboundSsl.getCertNickname(),
                            ssl2Enabled,
                            outboundSsl.getSsl2Ciphers(),
                            ssl3Enabled,
                            outboundSsl.getSsl3TlsCiphers(),
                            tlsEnabled);
                    }
                }
                if (clientSslInfo == null) {
                    clientSslInfo = getDefaultSslInfo();
                }
            } else {
                if ((processType != null) && (processType ==
                    ProcessType.ACC)) {
                    IIOPSSLUtil sslUtil =
                        Globals.getDefaultHabitat().getService(IIOPSSLUtil.class);
                    AppClientSSL clientSsl =
                        (AppClientSSL)sslUtil.getAppClientSSL();
                    if (clientSsl != null) {
                        clientSslInfo = init(clientSsl.getCertNickname(),
                                clientSsl.getSsl2Enabled(),
                                    clientSsl.getSsl2Ciphers(),
                                clientSsl.getSsl3Enabled(),
                                    clientSsl.getSsl3TlsCiphers(),
                                clientSsl.getTlsEnabled());
                    } else { // include case keystore, truststore jvm
                        option

                        clientSslInfo = getDefaultSslInfo();
                    }
                } else {
                    clientSslInfo = getDefaultSslInfo();
                }
            }
        } catch (Exception e) {
            _logger.log(Level.SEVERE,"iiop.init_exception",e);
            throw new IllegalStateException(e);
        }
    }

    /**
     * Return a default SSLInfo object.
     */
    private SSLInfo getDefaultSslInfo() throws Exception {
       return init(null, false, null, true, null, true);
    }

    /**
     * serveralias/clientalias cannot be set at the same time.
     * this method encapsulates the common code for both the client side
         and
     * server side to create a SSLContext
     * it is called once for each serveralias and once for each
         clientalias
     */
    private SSLInfo init(String alias,
            boolean ssl2Enabled, String ssl2Ciphers,
            boolean ssl3Enabled, String ssl3TlsCiphers,
            boolean tlsEnabled) throws Exception {

        String protocol;
        if (tlsEnabled) {
            protocol = TLS;
        } else if (ssl3Enabled) {
            protocol = SSL3;
        } else if (ssl2Enabled) {
            protocol = SSL2;
        } else { // default
            protocol = "SSL";
        }

        String[] ssl3TlsCipherArr = null;
        if (tlsEnabled || ssl3Enabled) {
            ssl3TlsCipherArr = getEnabledCipherSuites(ssl3TlsCiphers,
                    false, ssl3Enabled, tlsEnabled);
        }

        String[] ssl2CipherArr = null;
        if (ssl2Enabled) {
            ssl2CipherArr = getEnabledCipherSuites(ssl2Ciphers,
                    true, false, false);
        }

        SSLContext ctx = SSLContext.getInstance(protocol);
        if (Globals.getDefaultHabitat() != null) {
            IIOPSSLUtil sslUtil =
                Globals.getDefaultHabitat().getService(IIOPSSLUtil.class);
            KeyManager[] mgrs = sslUtil.getKeyManagers(alias);
            ctx.init(mgrs, sslUtil.getTrustManagers(),
                sslUtil.getInitializedSecureRandom());
        } else {
            //do nothing
            //ctx.init(mgrs, sslUtil.getTrustManagers(),
                sslUtil.getInitializedSecureRandom());
        }

        return new SSLInfo(ctx, ssl3TlsCipherArr, ssl2CipherArr);
    }

    //----- implements com.sun.corba.ee.spi.transport.ORBSocketFactory
        -----

    public void setORB(ORB orb) {
        this.orb = orb;
    }

    /**
     * Create a server socket on the specified InetSocketAddress based
         on the
     * type of the server socket (SSL, SSL_MUTUALAUTH, PERSISTENT_SSL or
         CLEAR_TEXT).
     * @param type type of socket to create.
     * @param inetSocketAddress the InetSocketAddress
     * @return the server socket on the specified InetSocketAddress
     * @exception IOException if an I/O error occurs during server socket
     * creation
     */
    public ServerSocket createServerSocket(String type,
            InetSocketAddress inetSocketAddress) throws IOException {

   if (_logger.isLoggable(Level.FINE)) {
       _logger.log(Level.FINE, "Creating server socket for type =" + type
               + " inetSocketAddress =" + inetSocketAddress);
   }

   if(type.equals(SSL_MUTUALAUTH) || type.equals(SSL) ||
      type.equals(PERSISTENT_SSL)) {
       return createSSLServerSocket(type, inetSocketAddress);
   } else {
           ServerSocket serverSocket = null;
           if (orb.getORBData().acceptorSocketType().equals(
                   ORBConstants.SOCKETCHANNEL)) {
               ServerSocketChannel serverSocketChannel =
                      ServerSocketChannel.open();
               serverSocket = serverSocketChannel.socket();
           } else {
               serverSocket = new ServerSocket();
           }

       serverSocket.bind(inetSocketAddress);
       return serverSocket;

   }
    }

    /**
     * Create a client socket for the specified InetSocketAddress.
         Creates an SSL
     * socket if the type specified is SSL or SSL_MUTUALAUTH.
     * @param type
     * @param inetSocketAddress
     * @return the socket.
     */
    public Socket createSocket(String type, InetSocketAddress
        inetSocketAddress)
           throws IOException {

   try {
       String host = inetSocketAddress.getHostName();
       int port = inetSocketAddress.getPort();
       if (_logger.isLoggable(Level.FINE)) {
      _logger.log(Level.FINE, "createSocket(" + type + ", " + host + ",
          " +port + ")");
       }
       if (type.equals(SSL) || type.equals(SSL_MUTUALAUTH)) {
      return createSSLSocket(host, port);
       } else {
               Socket socket = null;
      if (_logger.isLoggable(Level.FINE)) {
          _logger.log(Level.FINE, "Creating CLEAR_TEXT socket for:"
              +port);
      }

              if (orb.getORBData().connectionSocketType().equals(
                      ORBConstants.SOCKETCHANNEL)) {
             SocketChannel socketChannel =
                  ORBUtility.openSocketChannel(inetSocketAddress);
             socket = socketChannel.socket();
         } else {
                  socket = new Socket(inetSocketAddress.getHostName(),
                      inetSocketAddress.getPort());
         }

              // Disable Nagle's algorithm (i.e. always send
                   immediately).
      socket.setTcpNoDelay(true);
              return socket;
       }
   } catch ( Exception ex ) {
       if(_logger.isLoggable(Level.FINE)) {
      _logger.log(Level.FINE,"Exception creating socket",ex);
       }
       throw new RuntimeException(ex);
   }
    }

    public void setAcceptedSocketOptions(Acceptor acceptor,
            ServerSocket serverSocket, Socket socket) {
   if (_logger.isLoggable(Level.FINE)) {
       _logger.log(Level.FINE, "setAcceptedSocketOptions: " + acceptor
                  + " " + serverSocket + " " + socket);
   }
        // Disable Nagle's algorithm (i.e., always send immediately).
        try {
        socket.setTcpNoDelay(true);
        } catch (SocketException ex) {
            throw new RuntimeException(ex);
        }
    }

    //----- END implements
         com.sun.corba.ee.spi.transport.ORBSocketFactory -----

    /**
     * Create an SSL server socket at the specified InetSocketAddress.
         If the type
     * is SSL_MUTUALAUTH then SSL client authentication is requested.
     */
    private ServerSocket createSSLServerSocket(String type,
            InetSocketAddress inetSocketAddress) throws IOException {

```

```
        if (inetSocketAddress == null) {
            throw new IOException(getFormatMessage(
                "iiop.invalid_sslserverport",
                new Object[] { null }));
        }
        int port = inetSocketAddress.getPort();
        Integer iport = Integer.valueOf(port);
        SSLInfo sslInfo = (SSLInfo)portToSSLInfo.get(iport);
        if (sslInfo == null) {
            throw new IOException(getFormatMessage(
                "iiop.invalid_sslserverport",
                new Object[] { iport }));
        }
        SSLServerSocketFactory ssf =
            sslInfo.getContext().getServerSocketFactory();
        String[] ssl3TlsCiphers = sslInfo.getSsl3TlsCiphers();
        String[] ssl2Ciphers = sslInfo.getSsl2Ciphers();
        String[] ciphers = null;
        if (ssl3TlsCiphers != null || ssl2Ciphers != null) {
            String[] socketCiphers = ssf.getDefaultCipherSuites();
            ciphers = mergeCiphers(socketCiphers, ssl3TlsCiphers,
                ssl2Ciphers);
        }

    String cs[] = null;

    if(_logger.isLoggable(Level.FINE)) {
        cs = ssf.getSupportedCipherSuites();
        for(int i=0; i < cs.length; ++i) {
       _logger.log(Level.FINE,"Cipher Suite: " + cs[i]);
        }
    }
    ServerSocket ss = null;
        try{
            // bugfix for 6349541
            // specify the ip address to bind to, 50 is the default used
            // by the ssf implementation when only the port is specified
            ss = ssf.createServerSocket(port, BACKLOG,
                inetSocketAddress.getAddress());
            if (ciphers != null) {
                ((SSLServerSocket)ss).setEnabledCipherSuites(ciphers);
            }
        } catch(IOException e) {
            _logger.log(Level.SEVERE, "iiop.createsocket_exception",
                new Object[] { type, String.valueOf(port) });
            _logger.log(Level.SEVERE, "", e);
            throw e;
        }

    try {
        if(type.equals(SSL_MUTUALAUTH)) {
       _logger.log(Level.FINE,"Setting Mutual auth");
       ((SSLServerSocket)ss).setNeedClientAuth(true);
        }
    } catch(Exception e) {
        _logger.log(Level.SEVERE,"iiop.cipher_exception",e);
        throw new IOException(e.getMessage());
    }
    if(_logger.isLoggable(Level.FINE)) {
        _logger.log(Level.FINE,"Created server socket:" + ss);
    }
    return ss;
     }

     /**
      * Create an SSL socket at the specified host and port.
      * @param host
      * @param port
      * @return the socket.
      */
     private Socket createSSLSocket(String host, int port)
        throws IOException {
        SSLSocket socket = null;
    SSLSocketFactory factory = null;
        try{
            // get socketfactory+sanity check
            // clientSslInfo is never null
            factory = clientSslInfo.getContext().getSocketFactory();

            if(_logger.isLoggable(Level.FINE)) {
                  _logger.log(Level.FINE,"Creating SSL Socket for host:"
                      + host +" port:" + port);
            }
            String[] ssl3TlsCiphers = clientSslInfo.getSsl3TlsCiphers();
            String[] ssl2Ciphers = clientSslInfo.getSsl2Ciphers();
            String[] clientCiphers = null;
            if (ssl3TlsCiphers != null || ssl2Ciphers != null) {
                String[] socketCiphers = factory.getDefaultCipherSuites();
                clientCiphers = mergeCiphers(socketCiphers,
                    ssl3TlsCiphers, ssl2Ciphers);
            }

            socket = (SSLSocket)factory.createSocket(host, port);
            if (clientCiphers != null) {
                socket.setEnabledCipherSuites(clientCiphers);
            }
        }catch(Exception e) {
            if(_logger.isLoggable(Level.FINE)) {
                _logger.log(Level.FINE, "iiop.createsocket_exception",
                new Object[] { host, String.valueOf(port) });
```

```
                _logger.log(Level.FINE, "", e);
            }
            IOException e2 = new IOException(
            "Error opening SSL socket to host="+host+" port="+port);
            e2.initCause(e);
            throw e2;
        }
        return socket;
    }

    /**
     * This API return an array of String listing the enabled cipher
         suites.
     * Input is the cipherSuiteStr from xml which a space separated list
     * ciphers with a prefix '+' indicating enabled, '-' indicating
         disabled.
     * If no cipher is enabled, then it returns an empty array.
     * If no cipher is specified, then all are enabled and it returns
         null.
     * @param cipherSuiteStr cipherSuiteStr from xml
     * @param ssl2Enabled
     * @param ssl3Enabled
     * @param tlsEnabled
     * @return an array of enabled Ciphers
     */
    private String[] getEnabledCipherSuites(String cipherSuiteStr,
            boolean ssl2Enabled, boolean ssl3Enabled, boolean tlsEnabled)
                {
        String[] cipherArr = null;
        if (cipherSuiteStr != null && cipherSuiteStr.length() > 0) {
            ArrayList cipherList = new ArrayList();
            StringTokenizer tokens = new StringTokenizer(cipherSuiteStr,
                ",");
            while (tokens.hasMoreTokens()) {
                String cipherAction = tokens.nextToken();
                if (cipherAction.startsWith("+")) {
                    String cipher = cipherAction.substring(1);
                    CipherInfo cipherInfo =
                        CipherInfo.getCipherInfo(cipher);
                    if (cipherInfo != null &&
                            isValidProtocolCipher(cipherInfo, ssl2Enabled,
                                ssl3Enabled, tlsEnabled)) {
                        cipherList.add(cipherInfo.getCipherName());
                    } else {
                        throw new IllegalStateException(getFormatMessage(
                            "iiop.unknown_cipher",
                            new Object[] { cipher }));
                    }
                } else if (cipherAction.startsWith("-")) {
                    String cipher = cipherAction.substring(1);
                    CipherInfo cipherInfo =
                        CipherInfo.getCipherInfo(cipher);
                    if (cipherInfo == null ||
                            !isValidProtocolCipher(cipherInfo, ssl2Enabled,
                                ssl3Enabled, tlsEnabled)) {
                        throw new IllegalStateException(getFormatMessage(
                            "iiop.unknown_cipher",
                            new Object[] { cipher }));
                    }
                } else if (cipherAction.trim().length() > 0) {
                    throw new IllegalStateException(getFormatMessage(
                        "iiop.invalid_cipheraction",
                        new Object[] { cipherAction }));
                }
            }

            cipherArr = (String[])cipherList.toArray(
                    new String[cipherList.size()]);
        }
        return cipherArr;
    }

    /**
     * Return an array of merged ciphers.
     * @param enableCiphers ciphers enabled by socket factory
     * @param ssl3TlsCiphers
     * @param ssl2Ciphers
     */
    private String[] mergeCiphers(String[] enableCiphers,
            String[] ssl3TlsCiphers, String[] ssl2Ciphers) {

        if (ssl3TlsCiphers == null && ssl2Ciphers == null) {
            return null;
        }

        int eSize = (enableCiphers != null)? enableCiphers.length : 0;

        if (_logger.isLoggable(Level.FINE)) {
            StringBuffer buf = new StringBuffer("Default socket ciphers:
                ");
            for (int i = 0; i < eSize; i++) {
                buf.append(enableCiphers[i] + ", ");
            }
            _logger.log(Level.FINE, buf.toString());
        }

        ArrayList cList = new ArrayList();
        if (ssl3TlsCiphers != null) {
            for (int i = 0; i < ssl3TlsCiphers.length; i++) {
                cList.add(ssl3TlsCiphers[i]);
```

```java
            }
        } else {
            for (int i = 0; i < eSize; i++) {
                String cipher = enableCiphers[i];
                CipherInfo cInfo = CipherInfo.getCipherInfo(cipher);
                if (cInfo != null && (cInfo.isTLS() || cInfo.isSSL3())) {
                    cList.add(cipher);
                }
            }
        }

        if (ssl2Ciphers != null) {
            for (int i = 0; i < ssl2Ciphers.length; i++) {
                cList.add(ssl2Ciphers[i]);
            }
        } else {
            for (int i = 0; i < eSize; i++) {
                String cipher = enableCiphers[i];
                CipherInfo cInfo = CipherInfo.getCipherInfo(cipher);
                if (cInfo != null && cInfo.isSSL2()) {
                    cList.add(cipher);
                }
            }
        }

        if (_logger.isLoggable(Level.FINE)) {
            _logger.log(Level.FINE, "Merged socket ciphers: " + cList);
        }

        return (String[])cList.toArray(new String[cList.size()]);
    }

    /**
     * Check whether given cipherInfo belongs to given protocol.
     * @param cipherInfo
     * @param ssl2Enabled
     * @param ssl3Enabled
     * @param tlsEnabled
     */
    private boolean isValidProtocolCipher(CipherInfo cipherInfo,
            boolean ssl2Enabled, boolean ssl3Enabled, boolean tlsEnabled)
                {
        return (tlsEnabled && cipherInfo.isTLS() ||
                ssl3Enabled && cipherInfo.isSSL3() ||
                ssl2Enabled && cipherInfo.isSSL2());
    }

    /**
     * This API get the format string from resource bundle of _logger.
     * @param key the key of the message
     * @param params the parameter array of Object
     * @return the format String for _logger
     */
    private String getFormatMessage(String key, Object[] params) {
        return MessageFormat.format(
            _logger.getResourceBundle().getString(key), params);
    }

    class SSLInfo {
        private SSLContext ctx;
        private String[] ssl3TlsCiphers = null;
        private String[] ssl2Ciphers = null;

        SSLInfo(SSLContext ctx, String[] ssl3TlsCiphers, String[]
             ssl2Ciphers) {
            this.ctx = ctx;
            this.ssl3TlsCiphers = ssl3TlsCiphers;
            this.ssl2Ciphers = ssl2Ciphers;
        }

        SSLContext getContext() {
            return ctx;
        }

        String[] getSsl3TlsCiphers() {
            return ssl3TlsCiphers;
        }

        String[] getSsl2Ciphers() {
            return ssl2Ciphers;
        }
    }
}
```

2 Functional Role of Assigned Classes

Starting from the considerations made in the previous section, the **IIOPSSLSocketFactory** functional role is further analysed.

2.1 ORB Middleware Actors Overview

First of all, an overview of the **ORB** middleware is given, because it is the component that uses the **IIOP** protocol to communicate over the Internet. The **O**bject **R**equest **B**roker allows method calls to be made from one computer to another via network, and it provides that for each remote method call there are two main actors exchanging informations:

- **Client:** It requests a method call to an object which interface is exposed by the **Server** and is known to the **Client**. The **Client** has the capability of sending some parameters to the Server for executing the given method call and the capability of receiving back the return value of the called method, if any.
- **Server:** It exposes the interfaces of the objects that can be called by the various **Clients** allowed to make remote method calls. Through those interfaces, the **Clients** can make remote method calls, passing objects as parameters if necessary, and receiving a return value, if any.

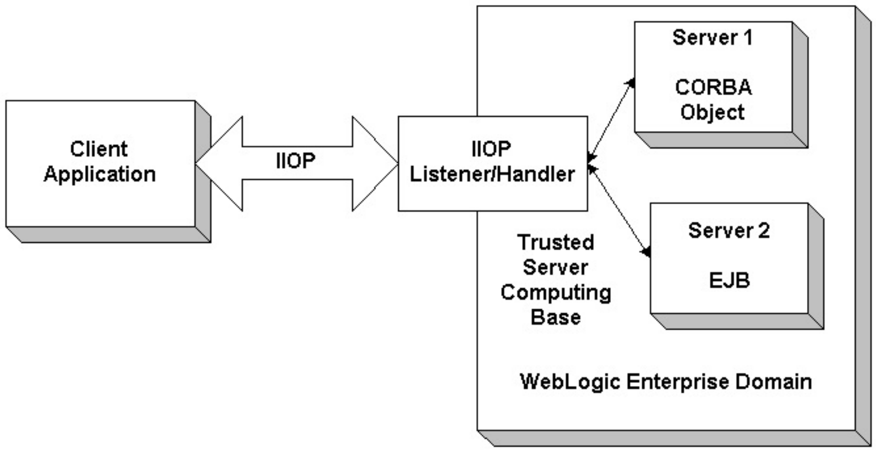

Figure 1: Overview of CORBA Architecture

2.2 IIOPSSLSocketFactory functionalities

In order to give each actor the capability of performing the actions provided by the middleware using the **IIOP** protocol, the **IIOPSSLSocketFactory** main functionalities are the following ones.

- **Socket Creation:** This functionality allows the creation of a **Socket** with some specific characteristics.
  - **Plain Socket:** Plain Text Socket (**java.net.Socket**) with *Nagle's algorithm disabled.* It is created using **java.nio.channels.SocketChannel**, if so is specified in the **ORB** object, whose reference is passed at runtime to a **IIOPSSLSocketFactory** object using a setter.
  - **Secure Socket:** Encrypted Socket (**javax.net.ssl.SSLSocket**) that uses **Secure Socket Layer** or **Transport Secure Layer**. Its characteristics are defined during the creation of a **IIOPSSLSocketFactory** object by *obtaining data from global variables* (which seems to be a bad behaviour) and storing those data into a specific private attribute of type **IIOPSSLSocketFactory.SSLInfo**. This private attribute is never modified after **IIOPSSLSocketFactory** object creation and thus it can be considered as final, although it is not declared final nor immutable. In particular, a secure socket built by **IIOPSSLSocketFactory** can use either one of the following cryptographic protocols for data encryption:
    - **SSL1**
    - **SSL2**
    - **SSL3**
    - **TLS**
- **Server Socket Creation:** This functionality allows the creation of a **Server Socket**.
  - **Plain Server Socket:** A Server Socket (**java.net.ServerSocket**) that accepts incoming **Plain Socket** connections from **Clients**. If the **ORB** object set into the given **IIOPSSLSocketFactory** object is configured accordingly, the **Server Socket** is created using **java.nio.ServerSocketChannel**.
  - **Secure Server Socket:** A Secure Server Socket (**javax.net.ssl.SSLSocket**) that accepts incoming **Secure Socket** connections on a certain **Port** of a given **IP** address. The **SSLInfo** object necessary to have the informations about how to build the **Secure Server Socket** are contained into an **IIOPSSLSocketFactory** attribute of type **java.util.Map** that associates a given **TCP Port** to the relevant **SSLInfo** object. This **java.util.Map** is *initialized from global variables* (which seems a bad habit again) at **IIOPSSLSocketFactory** object creation time and stores the association of every **IIOP Listener Port** to the relevant **IIOP Listener configuration**. An **IIOP listener**, using **Server Sockets**, accepts incoming connections from the remote **Clients** of **Enterprise Beans** and from other **CORBA** (Common Object Request Broker Architecture) based **Clients**.

The entire class behaviour depends on the type of process in which context the **IIOPSSLSocketFactory** object is built (A extremely bad modus operandi). There can exists two different types of processes in which this class could be used, inferring by the source code available:

- **EJB container:** The **EJB container** is the interface between **enterprise beans**, which provide the business logic in a **Java EE application**, and the **Java EE server**. The **EJB container** runs on the **Java EE server** and manages the execution of an **application's enterprise beans**.
- **Application Client Container:** The **Application Client Container** is the interface between **Java EE application clients** (special Java SE applications that use Java EE server components) and the **Java EE server**. The **application client container** runs on the **client machine** and is the gateway between the **client application** and the **Java EE server components** that the client uses.

The reader who would get a more comprehensive overview on this topic is suggested to go through this Oracle documentation. So, again using inference (the class documentation is not enough detailed to get a complete knowledge about these facts) on what has been discovered so far, it can be concluded that the class functional role is fundamentally different in the case of running into a **EJB Container** than on an **Application Client Container**. To recap, the functional role is either one of the following two:

- **EJB Container Functional Role:** The **Server Socket Creation** functionality is used to give a **IIOP Listener** the capability of accepting incoming **Plain Text Socket** and **Secure Socket** connections, in order to receive **Remote Method Calls** through **IIOP**. On the other side, the **Socket Creation** capability is exploited when the **EJB Container** needs to make **Remote Method Calls** using **IIOP** to another remote **EJB Container**.
- **Application Client Container Functional Role:** The **Socket Creation** functionality is used to connect to a remote **IIOP Listener** that is running into an **EJB Container** in order to deliver a **Remote Method Call**, and receive the **Return Value**, if any.

2.3 References

To study the functional role of **IIOPSSLSocketFactory** and further topics, some references have been consulted:

- Wikipedia, the free encyclopedia
- Expectations, Outcomes, and Challenges Of Modern Code Review by Alberto Bacchelli and Christian Bird

- Object Management Group (OMG) website
- Java Platform, Standard Edition 7 API Specification
- Java RMI over IIOP
- GrepCode for class and related classes source code and documentation
- GlassFish Server Administration Guide: Administering the Object Request Broker (ORB)
- SSL/TLS Strong Encryption: An Introduction
- Java EE Servers and Containers
- What is CORBA

3 Issues Found

In this section are reported all the coding choices that do not meet the **Code Inspection Checklist** given.

3.1 Code Inspection Checklist

3.1.1 Naming Conventions

- **Checklist[1]:** All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.
- **Checklist[2]:** If one-character variables are used, they are used only for temporary throwaway variables, such as those used in for loops.
- **Checklist[3]:** Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: class Raster; class ImageSprite;
- **Checklist[4]:** Interface names should be capitalized like classes.
- **Checklist[5]:** Method names should be verbs, with the first letter of each addition word capitalized. Examples: getBackground(); computeTemperature().
- **Checklist[6]:** Class variables, also called attributes, are mixed case, but might begin with an underscore (_) followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized. Examples: _windowHeight, timeSeriesData.
- **Checklist[7]:** Constants are declared using all uppercase with words separated by an underscore. Examples: MIN_WIDTH; MAX_HEIGHT;

3.1.2 Indention

- **Checklist[8]:** Three or four spaces are used for indentation and done so consistently
- **Checklist[9]:** No tabs are used to indent

3.1.3 Braces

- **Checklist[10]:** Consistent bracing style is used, either the preferred Allman style (first brace goes underneath the opening block) or the Kernighan and Ritchie style (first brace is on the same line of the instruction that opens the new block).
- **Checklist[11]:** All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces. Example: Avoid this:

```
        if (condition)
           doThis();
```

Instead do this:

```
        if (condition)
        {
           doThis();
        }
```

3.1.4 File Organization

- **Checklist[12]:** Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).
- **Checklist[13]:** Where practical, line length does not exceed 80 characters.
- **Checklist[14]:** When line length must exceed 80 characters, it does NOT exceed 120 characters.

3.1.5 Wrapping Lines

- **Checklist[15]:** Line break occurs after a comma or an operator.
- **Checklist[16]:** Higher-level breaks are used.
- **Checklist[17]:** A new statement is aligned with the beginning of the expression at the same level as the previous line.

3.1.6 Comments

- **Checklist[18]:** Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.
- **Checklist[19]:** Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

3.1.7 Java Source Files

- **Checklist[20]:** Each Java source file contains a single public class or interface.
- **Checklist[21]:** The public class is the first class or interface in the file.

- **Checklist[22]:** Check that the external program interfaces are implemented consistently with what is described in the javadoc.
- **Checklist[23]:** Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

3.1.8 Package and Import Statements

- **Checklist[24]:** If any package statements are needed, they should be the first non-comment statements. Import statements follow.

3.1.9 Class and Interface Declarations

- **Checklist[25]:** The class or interface declarations shall be in the following order:

    1. class/interface documentation comment
    2. class or interface statement
    3. class/interface implementation comment, if necessary
    4. class (static) variables
        (a) first public class variables
        (b) next protected class variables
        (c) next package level (no access modifier)
        (d) last private class variables
    5. instance variables
        (a) first public instance variables
        (b) next protected instance variables
        (c) next package level (no access modifier)
        (d) last private instance variables
    6. constructors
    7. methods

- **Checklist[26]:** Methods are grouped by functionality rather than by scope or accessibility.
- **Checklist[27]:** Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.

3.1.10 Initialization and Declarations

- **Checklist[28]:** Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected)
- **Checklist[29]:** Check that variables are declared in the proper scope

- **Checklist[30]:** Check that constructors are called when a new object is desired
- **Checklist[31]:** Check that all object references are initialized before use
- **Checklist[32]:** Variables are initialized where they are declared, unless dependent upon a computation
- **Checklist[33]:** Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces and). The exception is a variable can be declared in a for loop.

3.1.11 Method Calls

- **Checklist[34]:** Check that parameters are presented in the correct order
- **Checklist[35]:** Check that the correct method is being called, or should it be a different method with a similar name
- **Checklist[36]:** Check that method returned values are used properly

3.1.12 Arrays

- **Checklist[37]:** Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index)
- **Checklist[38]:** Check that all array (or other collection) indexes have been prevented from going out-of-bounds
- **Checklist[39]:** Check that constructors are called when a new array item is desired

3.1.13 Object Comparison

- **Checklist[40]:** Check that all objects (including Strings) are compared with ”equals” and not with ”==”

3.1.14 Output Format

- **Checklist[41]:** Check that displayed output is free of spelling and grammatical errors
- **Checklist[42]:** Check that error messages are comprehensive and provide guidance as to how to correct the problem
- **Checklist[43]:** Check that the output is formatted correctly in terms of line stepping and spacing

3.1.15 Computation, Comparisons and Assignments

- **Checklist[44]:** Check that the implementation avoids brutish programming: (see Brutish Programming)
- **Checklist[45]:** Check order of computation/evaluation, operator precedence and parenthesizing
- **Checklist[46]:** Check the liberal use of parenthesis is used to avoid operator precedence problems.
- **Checklist[47]:** Check that all denominators of a division are prevented from being zero
- **Checklist[48]:** Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding
- **Checklist[49]:** Check that the comparison and Boolean operators are correct
- **Checklist[50]:** Check throw-catch expressions, and check that the error condition is actually legitimate
- **Checklist[51]:** Check that the code is free of any implicit type conversions

3.1.16 Exceptions

- **Checklist[52]:** Check that the relevant exceptions are caught
- **Checklist[53]:** Check that the appropriate action are taken for each catch block

3.1.17 Flow of Control

- **Checklist[54]:** In a switch statement, check that all cases are addressed by break or return
- **Checklist[55]:** Check that all switch statements have a default branch
- **Checklist[56]:** Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions

3.1.18 Files

- **Checklist[57]:** Check that all files are properly declared and opened
- **Checklist[58]:** Check that all files are closed properly, even in the case of an error
- **Checklist[59]:** Check that EOF conditions are detected and handled correctly

- **Checklist[60]:** Check that all file exceptions are caught and dealt with accordingly

3.2 Class Issues

In this subsection are listed the issues related to the whole class and not only to a specific method.

3.2.1 Naming Conventions

- **Checklist[1]:**
  - The class has the capability of creating also **Plain Text Sockets** and **Plain Text Server Sockets**, even if the name **IIOPSSLSocket-Factory** clearly underlines that the class has to be a **Factory** of **Secure Sockets** and **Server Sockets**. In order to give the architecture the capability of creating **Plain Text Sockets** and **Plain Text Server Sockets**, a separate class should have been designed.
  - The method

```
private SSLInfo init(String alias,
        boolean ssl2Enabled, String ssl2Ciphers,
        boolean ssl3Enabled, String ssl3TlsCiphers,
        boolean tlsEnabled) throws Exception {
```

    has a name which is not meaningful at all. It's code could in fact be moved into the **SSLInfo constructor**.
  - The method

```
private SSLInfo getDefaultSslInfo() throws Exception {
    return init(null, false, null, true, null, true);
}
```

    has a name which is not really meaningful. It could have been omitted and substituted by a **SSLInfo constant** inside either the **IIOPSSLSocketFactory** class or the **SSLInfo** inner class.
  - The method

```
private boolean isValidProtocolCipher(CipherInfo
    cipherInfo,
        boolean ssl2Enabled, boolean ssl3Enabled, boolean
            tlsEnabled) {
      return (tlsEnabled && cipherInfo.isTLS() ||
              ssl3Enabled && cipherInfo.isSSL3() ||
              ssl2Enabled && cipherInfo.isSSL2());
}
```

has a name that is not really meaningful, and its functionality is in fact disjointed by the one of **Socket and Server Socket Creation**.

  – The method

```
    private String[] mergeCiphers(String[] enableCiphers,
            String[] ssl3TlsCiphers, String[] ssl2Ciphers) {
```

has a name that is not really meaningful, and its functionality is in fact disjointed by the one of **Socket and Server Socket Creation**.

  – The method

```
    public void setAcceptedSocketOptions(Acceptor acceptor,
            ServerSocket serverSocket, Socket socket) {
   if (_logger.isLoggable(Level.FINE)) {
       _logger.log(Level.FINE, "setAcceptedSocketOptions: " +
           acceptor
                  + " " + serverSocket + " " + socket);
   }
       // Disable Nagle's algorithm (i.e., always send
           immediately).
       try {
       socket.setTcpNoDelay(true);
       } catch (SocketException ex) {
          throw new RuntimeException(ex);
       }
    }
```

is simply terrible. No meaningful name, useless parameters and fuzzy functionality.

- **Checklist[7]:** The constant **_logger** follows the naming convention of normal attributes, even if it is a constant.

```
    private static final Logger _logger = LogDomains.getLogger(
        IIOPSSLSocketFactory.class, LogDomains.CORBA_LOGGER);
```

3.2.2 Comments

- **Checklist[18]:** The only adequate comment in the whole class is at line 337.

```
336                 // Disable Nagle's algorithm (i.e. always send
                        immediately).
337     socket.setTcpNoDelay(true);
```

The class is not adequately commented at all.

- **Checklist[19]:** The commented out code at lines 103, 135 and 254 is left alone without any additional hint.

```
103    //private static SecureRandom sr = null;
```

```
134            }
135            //if (Switch.getSwitch().getContainerType() ==
                   Switch.EJBWEB_CONTAINER) {
136            if((processType != null) && (processType.isServer())) {
```

```
253            //do nothing
254            //ctx.init(mgrs, sslUtil.getTrustManagers(),
                   sslUtil.getInitializedSecureRandom());
255        }
```

3.2.3 Java Source Files

- **Checklist[20]:** The main class contains an internal class named **SSLInfo**:

```
class SSLInfo {
    private SSLContext ctx;
    private String[] ssl3TlsCiphers = null;
    private String[] ssl2Ciphers = null;

    SSLInfo(SSLContext ctx, String[] ssl3TlsCiphers, String[]
         ssl2Ciphers) {
        this.ctx = ctx;
        this.ssl3TlsCiphers = ssl3TlsCiphers;
        this.ssl2Ciphers = ssl2Ciphers;
    }

    SSLContext getContext() {
        return ctx;
    }

    String[] getSsl3TlsCiphers() {
        return ssl3TlsCiphers;
    }

    String[] getSsl2Ciphers() {
        return ssl2Ciphers;
    }
```

- **Checklist[23]:** The provided *Javadoc* is not complete and is not an help in understanding the class behaviour.

3.2.4 Class and Interface Declarations

- **Checklist[26]:** The methods returning a *SSLInfo Object*, for example *getDefaultSslInfo()* and *init()*, should be declared in the *SSLInfo* inner class but here they are implemented in the main class. This is an hint for improving the readability of the code and for avoiding the continuous scrolling of the class code

```
209     private SSLInfo getDefaultSslInfo() throws Exception {
```

```
219     private SSLInfo init(String alias,
220             boolean ssl2Enabled, String ssl2Ciphers,
221             boolean ssl3Enabled, String ssl3TlsCiphers,
222             boolean tlsEnabled) throws Exception {
```

- **Checklist[27]:** The constructor *IIOPSSLSocketFactory()* is 75 lines of code long and it is not easily comprehensible. Furthermore it is full of *if - else* structures with no meaningful conditions. A such long method worsens the readability and the instant comprehension of the method's role in the code. In addition it would be better to refactor and separate the atomic parts of code in additional methods.

3.2.5 Initialization and Declarations

- **Checklist[28]:**
  - **Lines 94 to 97:** The private variables *TLS, SSLv3, SSLv2, SSL* are used as simple string instead of creating an enumeration for better show the logic bond between each one and the other ones.

```
94     private static final String TLS = "TLS";
95     private static final String SSL3 = "SSLv3";
96     private static final String SSL2 = "SSLv2";
97     private static final String SSL = "SSL";
```

- **Checklist[32]:**
  - **Line 118:** The *clientSslInfo Object* is initialized with *null* value even if it is useless.

```
118     private SSLInfo clientSslInfo = null;
```

3.3 Method Issues

The checklist has also been checked analysing the assigned source code file method by method.

3.3.1 Issues in createServerSocket

```
266  /**
267   * Create a server socket on the specified InetSocketAddress based
            on the
268   * type of the server socket (SSL, SSL_MUTUALAUTH, PERSISTENT_SSL or
            CLEAR_TEXT).
269   * @param type type of socket to create.
270   * @param inetSocketAddress the InetSocketAddress
271   * @return the server socket on the specified InetSocketAddress
272   * @exception IOException if an I/O error occurs during server socket
273   * creation
274   */
275  public ServerSocket createServerSocket(String type,
276          InetSocketAddress inetSocketAddress) throws IOException {
277
278 if (_logger.isLoggable(Level.FINE)) {
279     _logger.log(Level.FINE, "Creating server socket for type =" + type
280              + " inetSocketAddress =" + inetSocketAddress);
281 }
282
283 if(type.equals(SSL_MUTUALAUTH) || type.equals(SSL) ||
284    type.equals(PERSISTENT_SSL)) {
285     return createSSLServerSocket(type, inetSocketAddress);
286 } else {
287          ServerSocket serverSocket = null;
288          if (orb.getORBData().acceptorSocketType().equals(
289                  ORBConstants.SOCKETCHANNEL)) {
290              ServerSocketChannel serverSocketChannel =
291                      ServerSocketChannel.open();
292              serverSocket = serverSocketChannel.socket();
293          } else {
294              serverSocket = new ServerSocket();
295          }
296
297     serverSocket.bind(inetSocketAddress);
298     return serverSocket;
299
300 }
301  }
```

The following problems have been found in this method.

3.3.1.1 Indention

- **Checklist[8]:**
  - **Lines 277 to 300:** The whole method lacks a level of indentation.
  - **Lines 287 to 295:** Have an extra level of indentation.

- **Checklist**[9]**:**
  - **Lines 278, 281, 283, 286, 300:** Are indented using one tab.
  - **Lines 279, 285, 297, 298:** Are indented using one tab and four spaces.
  - **Line 284:** Is indented using two tabs.

3.3.1.2 File Organization

- **Checklist**[12]**:** Lines 277, 289, 296 and 299 are blank without a clear reason.

3.3.1.3 Wrapping Lines

- **Checklist**[15]**:**
  - **Line 279:** Is broken before an operator.
  - **Line 288:** Is broken at an open parenthesis.
- **Checklist**[16]**:**
  - **Line 288:** A lower-level break occurs.
- **Checklist**[17]**:**
  - **Lines 280:** Is aligned with an extra level of indentation.

3.3.1.4 Comments

- **Checklist**[18]**:** The provided *JavaDoc* is too short and not really explicative. It does not completely explain the method functionalities.

```
266     /**
267      * Create a server socket on the specified InetSocketAddress
            based on the
268      * type of the server socket (SSL, SSL_MUTUALAUTH,
            PERSISTENT_SSL or CLEAR_TEXT).
269      * @param type type of socket to create.
270      * @param inetSocketAddress the InetSocketAddress
271      * @return the server socket on the specified InetSocketAddress
272      * @exception IOException if an I/O error occurs during server
            socket
273      * creation
274      */
```

3.3.1.5 Initialization and Declarations

- **Checklist**[**28**]: For the parameter **type**, it would have been better to use an enumeration instead of a **String**.
- **Checklist**[**32**]: At line 285, the **serverSocket** local variable is initialized to **null** without reason.

3.3.1.6 Computation, Comparisons and Assignments

- **Checklist**[**44**]:
  - **Lines 283 to 284:** The **if** condition is not explicit and requires inference to be fully understood. In addition it is error prone. The whole problem should have been faced using an **enumeration** instead of **String** constants.

```
283    if(type.equals(SSL_MUTUALAUTH) || type.equals(SSL) ||
284       type.equals(PERSISTENT_SSL)) {
```

  - **Line 287:** The initialization to **null** could have been omitted.

```
287             ServerSocket serverSocket = null;
```

  - **Lines 288 to 293:** The local variable **serverSocketChannel** is useless, and the **if** condition could have been wrapped in a boolean private method for better readability.

```
288             if (orb.getORBData().acceptorSocketType().equals(
289                     ORBConstants.SOCKETCHANNEL)) {
290                 ServerSocketChannel serverSocketChannel =
291                         ServerSocketChannel.open();
292                 serverSocket = serverSocketChannel.socket();
293             } else {
```

3.3.2 Issues in createSocket

```
    /**
     * Create a client socket for the specified InetSocketAddress. Creates an SSL
     * socket if the type specified is SSL or SSL_MUTUALAUTH.
     * @param type
     * @param inetSocketAddress
     * @return the socket.
     */
    public Socket createSocket(String type, InetSocketAddress inetSocketAddress)
```

```
311         throws IOException {
312
313 try {
314     String host = inetSocketAddress.getHostName();
315     int port = inetSocketAddress.getPort();
316     if (_logger.isLoggable(Level.FINE)) {
317    _logger.log(Level.FINE, "createSocket(" + type + ", " + host + ",
           " +port + ")");
318     }
319     if (type.equals(SSL) || type.equals(SSL_MUTUALAUTH)) {
320    return createSSLSocket(host, port);
321     } else {
322             Socket socket = null;
323    if (_logger.isLoggable(Level.FINE)) {
324       _logger.log(Level.FINE, "Creating CLEAR_TEXT socket for:"
              +port);
325    }
326
327             if (orb.getORBData().connectionSocketType().equals(
328                    ORBConstants.SOCKETCHANNEL)) {
329            SocketChannel socketChannel =
                   ORBUtility.openSocketChannel(inetSocketAddress);
330            socket = socketChannel.socket();
331        } else {
332                socket = new Socket(inetSocketAddress.getHostName(),
333                    inetSocketAddress.getPort());
334        }
335
336             // Disable Nagle's algorithm (i.e. always send
                    immediately).
337    socket.setTcpNoDelay(true);
338             return socket;
339     }
340 } catch ( Exception ex ) {
341     if(_logger.isLoggable(Level.FINE)) {
342    _logger.log(Level.FINE,"Exception creating socket",ex);
343     }
344     throw new RuntimeException(ex);
345 }
346  }
```

The following problems have been found in this method.

3.3.2.1 Indention

- **Checklist[8]:**
  - **Lines 313 to 345:** The whole method is not indented correctly.
  - **Lines 317, 320, 342:** Lack an extra level of indentation, over the one mentioned above.

- **Checklist[9]:** Excluding lines 311, 322, 326, 327, 328, 332, 333, 335, 336, 338 and 346, tabs are always used to indent, in conjugation with four spaces.

3.3.2.2 File Organization

- **Checklist[12]:**
  - **Lines 312, 326, 335:** Are blank without reason.
- **Checklist[13]:**
  - **Line 304:** Is 82 characters long.
  - **Line 310:** Is 81 characters long.
  - **Line 317:** Is 90 characters long.
  - **Line 329:** Is 95 characters long.

3.3.2.3 Wrapping Lines

- **Checklist[15]:**
  - **Line 327, 328:** A break occurs after an open parenthesis.
- **Checklist[16]:**
  - **Lines 327, 328:** Low-level break is used.
- **Checklist[17]:**
  - **Line 311:** Has an extra level of indentation.

3.3.2.4 Comments

- **Checklist[18]:** The whole method is not commented enough.

3.3.2.5 Initialization and Declarations

- **Checklist[28]:** At line 401 the variable **ss** should have been declared of type **SSLServerSocket** instead of plain **ServerSocket**.
- **Checklist[29]:** At lines 332, 332 the local variables **host** and **port** could have been used instead for better code readability.

```
314         String host = inetSocketAddress.getHostName();
315         int port = inetSocketAddress.getPort();
```

```
332                     socket = new
                             Socket(inetSocketAddress.getHostName(),
333                         inetSocketAddress.getPort());
```

- **Checklist[32]:** Line 322, the **socket** variable is initialized to **null**, but that value is immediately overwritten and therefore the initialization is useless.

3.3.2.6 Output Format

- **Checklist[42]:** The error message

```
342          _logger.log(Level.FINE,"Exception creating socket",ex);
```

  is not explaining anything about the error that has occurred.

3.3.2.7 Computation, Comparisons and Assignments

- **Checklist[44]:**
  - **Line 319:** The **if** condition

    ```
    319          if (type.equals(SSL) || type.equals(SSL_MUTUALAUTH)) {
    ```

    is not clear enough, invoking a dedicate boolean method and using enumeration could have delivered better results.
  - **Lines 327, 328:** The **if** condition

    ```
    327                  if
                             (orb.getORBData().connectionSocketType().equals(
    328                         ORBConstants.SOCKETCHANNEL)) {
    ```

    is not clear enough, invoking a dedicate boolean method and using enumeration could have delivered better results.
  - **Line 329:** The local variable **socketChannel**

    ```
    329                 SocketChannel socketChannel =
                            ORBUtility.openSocketChannel(inetSocketAddress);
    330                 socket = socketChannel.socket();
    331             } else {
    ```

    is useless.

3.3.2.8 Exceptions

- **Checklist[52]:**
  - **Lines 340 to 345:** The **catch** block

    ```
    340    } catch ( Exception ex ) {
    341        if(_logger.isLoggable(Level.FINE)) {
    342       _logger.log(Level.FINE,"Exception creating socket",ex);
    ```

```
343         }
344         throw new RuntimeException(ex);
345     }
```

is actually catching a generic **Exception** instead of the generated ones.

- **Checklist[53]:**
  - **Lines 340 to 345:** The **catch** block mentioned above is only outputting a generic log and re-throwing a generic **RuntimeException**, built using the caught one.

3.3.3 Issues in createSSLServerSocket

```
364     /**
365      * Create an SSL server socket at the specified InetSocketAddress.
            If the type
366      * is SSL_MUTUALAUTH then SSL client authentication is requested.
367      */
368     private ServerSocket createSSLServerSocket(String type,
369             InetSocketAddress inetSocketAddress) throws IOException {
370 
371         if (inetSocketAddress == null) {
372             throw new IOException(getFormatMessage(
373                 "iiop.invalid_sslserverport",
374                 new Object[] { null }));
375         }
376         int port = inetSocketAddress.getPort();
377         Integer iport = Integer.valueOf(port);
378         SSLInfo sslInfo = (SSLInfo)portToSSLInfo.get(iport);
379         if (sslInfo == null) {
380             throw new IOException(getFormatMessage(
381                 "iiop.invalid_sslserverport",
382                 new Object[] { iport }));
383         }
384         SSLServerSocketFactory ssf =
                sslInfo.getContext().getServerSocketFactory();
385         String[] ssl3TlsCiphers = sslInfo.getSsl3TlsCiphers();
386         String[] ssl2Ciphers = sslInfo.getSsl2Ciphers();
387         String[] ciphers = null;
388         if (ssl3TlsCiphers != null || ssl2Ciphers != null) {
389             String[] socketCiphers = ssf.getDefaultCipherSuites();
390             ciphers = mergeCiphers(socketCiphers, ssl3TlsCiphers,
                    ssl2Ciphers);
391         }
392 
393    String cs[] = null;
394 
```

```
    if(_logger.isLoggable(Level.FINE)) {
        cs = ssf.getSupportedCipherSuites();
        for(int i=0; i < cs.length; ++i) {
       _logger.log(Level.FINE,"Cipher Suite: " + cs[i]);
        }
    }
    ServerSocket ss = null;
         try{
            // bugfix for 6349541
            // specify the ip address to bind to, 50 is the default used
            // by the ssf implementation when only the port is specified
            ss = ssf.createServerSocket(port, BACKLOG,
                inetSocketAddress.getAddress());
            if (ciphers != null) {
                ((SSLServerSocket)ss).setEnabledCipherSuites(ciphers);
            }
         } catch(IOException e) {
            _logger.log(Level.SEVERE, "iiop.createsocket_exception",
                new Object[] { type, String.valueOf(port) });
            _logger.log(Level.SEVERE, "", e);
            throw e;
         }

    try {
        if(type.equals(SSL_MUTUALAUTH)) {
       _logger.log(Level.FINE,"Setting Mutual auth");
       ((SSLServerSocket)ss).setNeedClientAuth(true);
        }
    } catch(Exception e) {
        _logger.log(Level.SEVERE,"iiop.cipher_exception",e);
        throw new IOException(e.getMessage());
    }
    if(_logger.isLoggable(Level.FINE)) {
        _logger.log(Level.FINE,"Created server socket:" + ss);
    }
    return ss;
     }
```

The following problems have been found in this method.

3.3.3.1 Naming Conventions

- **Checklist[1]:**
  - **Lines 376, 377:** The difference between *port* and *iport* should be more highlighted through the naming choices.

```
376        int port = inetSocketAddress.getPort();
377        Integer iport = Integer.valueOf(port);
```

- **Line 384:** The variable name *ssf* is not really meaningful.

```
384         SSLServerSocketFactory ssf =
                sslInfo.getContext().getServerSocketFactory();
```

- **Line 393:** The variable name *cs* is not really meaningful.

```
393   String cs[] = null;
```

- **Line 401:** The variable name *ss* is not really meaningful.

```
401   ServerSocket ss = null;
```

3.3.3.2 Indention

- **Checklist[8]:**
  - **Lines 368 to 430:** The whole method is not indented correctly.
  - **Lines 398, 419, 420:** Lack an extra level of indentation, over the one mentioned above.
- **Checklist[9]:**
  - **Lines 396, 397, 399, 418, 421, 423, 424, 427:** These lines are indented using four spaces and one tab. This approach is neither consistent with the (wrong) style adopted in the whole method.

3.3.3.3 File Organization

- **Checklist[12]:**
  - **Lines 392, 394, 416:** Are blank without reason.
- **Checklist[13]:**
  - **Line 368:** Is 113 characters long.
  - **Line 372:** Is 92 characters long.
  - **Line 380:** Is 93 characters long.
  - **Line 411:** Is 101 characters long.

3.3.3.4 Wrapping Lines

- **Checklist[15]:**
  - **Lines 372 and 380:** The line break occurs after an open rounded bracket.

```
372            throw new IOException(getFormatMessage(
373                "iiop.invalid_sslserverport",
374                new Object[] { null }));
```

```
380            throw new IOException(getFormatMessage(
381                "iiop.invalid_sslserverport",
382                new Object[] { iport }));
```

3.3.3.5 Comments

- **Checklist[18]:** The provided *JavaDoc* is too short and not really explicative. It does not completely explain the method functionalities.

```
364    /**
365     * Create an SSL server socket at the specified
           InetSocketAddress. If the type
366     * is SSL_MUTUALAUTH then SSL client authentication is
           requested.
367     */
```

3.3.3.6 Initialization and Declarations

- **Checklist[31]:**
  - **Lines 393 and 393:** The variable *cs* is initialized to *null* even if is useless.

```
393      String cs[] = null;
```

  - **Lines 401 and 401:** The variable *ss* is initialized to *null* even if is useless.

```
401      ServerSocket ss = null;
```

- **Checklist[33]:**
  - **Lines 393 and 401:** The *cs* and *ss* variables are declared at the middle of the method code. They have to be declared at the beginning of it.

```
393      String cs[] = null;
```

```
401      ServerSocket ss = null;
```

3.3.3.7 Output Format

- **Checklist**[42]: In the *catch blocks* the caught exceptions are not explained to the user, they are only printed out.
  - **Line 411:**

```
411             _logger.log(Level.SEVERE,
                    "iiop.createsocket_exception",
412                 new Object[] { type, String.valueOf(port) });
413             _logger.log(Level.SEVERE, "", e);
```

  - **Line 423:**

```
423         _logger.log(Level.SEVERE,"iiop.cipher_exception",e);
```

3.3.3.8 Exceptions

- **Checklist**[52]:
  - **Lines 422 to 425:** The **catch** block

```
422     } catch(Exception e) {
423         _logger.log(Level.SEVERE,"iiop.cipher_exception",e);
424         throw new IOException(e.getMessage());
425     }
```

    is actually catching a generic **Exception** instead of the generated ones.

- **Checklist**[53]:
  - The problem in the above *code-block* is that, for every generated exception, this block throws an **IOException**, even if the caught exception is not an **IO** one.

3.3.4 Issues in createSSLSocket

```
432     /**
433      * Create an SSL socket at the specified host and port.
434      * @param host
435      * @param port
436      * @return the socket.
437      */
438     private Socket createSSLSocket(String host, int port)
439         throws IOException {
440         SSLSocket socket = null;
441   SSLSocketFactory factory = null;
```

```java
        try{
            // get socketfactory+sanity check
            // clientSslInfo is never null
            factory = clientSslInfo.getContext().getSocketFactory();

            if(_logger.isLoggable(Level.FINE)) {
                  _logger.log(Level.FINE,"Creating SSL Socket for host:"
                      + host +" port:" + port);
            }
            String[] ssl3TlsCiphers = clientSslInfo.getSsl3TlsCiphers();
            String[] ssl2Ciphers = clientSslInfo.getSsl2Ciphers();
            String[] clientCiphers = null;
            if (ssl3TlsCiphers != null || ssl2Ciphers != null) {
                String[] socketCiphers = factory.getDefaultCipherSuites();
                clientCiphers = mergeCiphers(socketCiphers,
                    ssl3TlsCiphers, ssl2Ciphers);
            }

            socket = (SSLSocket)factory.createSocket(host, port);
            if (clientCiphers != null) {
                socket.setEnabledCipherSuites(clientCiphers);
            }
        }catch(Exception e) {
            if(_logger.isLoggable(Level.FINE)) {
                _logger.log(Level.FINE, "iiop.createsocket_exception",
                new Object[] { host, String.valueOf(port) });
                _logger.log(Level.FINE, "", e);
            }
            IOException e2 = new IOException(
            "Error opening SSL socket to host="+host+" port="+port);
            e2.initCause(e);
            throw e2;
        }
        return socket;
    }
```

The following problems have been found in this method.

3.3.4.1 Naming Conventions

- **Checklist[1]:**
  - The following piece of code contains a variable named **e2** whose name is not meaningful.

```java
            IOException e2 = new IOException(
            "Error opening SSL socket to host="+host+"
                port="+port);
```

3.3.4.2 Indention

- **Checklist[8]:**
  - **Line 441, 469:** Lack a level of indentation.
- **Checklist[9]:**
  - **Line 441:** Is indented using a tab instead of four spaces.

3.3.4.3 File Organization

- **Checklist[12]:**
  - **Lines 446, 457:** Are blank without reason.
- **Checklist[13]:**
  - **Line 448:** Is 99 characters long.
  - **Line 455:** Is 90 characters long.

3.3.4.4 Wrapping Lines

- **Checklist[15]:**
  - **Line 468:** Is broken at an open parenthesis.
- **Checklist[17]:**
  - **Lines 465, 469:** Lack an extra level of indentation.

3.3.4.5 Comments

- **Checklist[18]:** Comments and JavaDoc provided in this method are completely useless.

3.3.4.6 Initialization and Declarations

- **Checklist[33]:**
  - **Lines 450, 451, 452, 468:** The local variables are not initialized at the beginning of their relevant blocks.

3.3.4.7 Output Format

- **Checklist[42]:** At lines from 462 to 472 the error message generated are a bit too general and not specific. They may not really help debugging the problem.

```
462         }catch(Exception e) {
463             if(_logger.isLoggable(Level.FINE)) {
464                 _logger.log(Level.FINE,
                        "iiop.createsocket_exception",
465                 new Object[] { host, String.valueOf(port) });
466                 _logger.log(Level.FINE, "", e);
467             }
468             IOException e2 = new IOException(
469             "Error opening SSL socket to host="+host+" port="+port);
470             e2.initCause(e);
471             throw e2;
472         }
```

3.3.4.8 Exceptions

- **Checklist[52]:** At lines 462 to 472 is caught a generic **Exception** instead of the generated ones.
- **Checklist[53]:** At lines 462 to 472 a **IOException** is created in the place of the generic **Exception** caught, and it is configured and re-thrown. This modus operandi destroys information about the error occurred in the first place, given that the logging is poor.

4 Other Problems

4.1 Nagle's Algorithm Disabling

By default, Nagle's algorithm is disabled for **all and only** the plain sockets built: this is done by setting the "TCP No Delay" property of the java.net.Socket to true, using the given setter. For encrypted sockets, Nagle's algorithm is not disabled and this could lead to severe performance issue. This is because Nagle's algorithm is essentially delaying the delivery of TCP Packets in order to avoid the delivery of several small packets (which obviously increases the overhead to data ratio), preferring less but bigger packets. Thus, a server response that is generated very fast could be delivered later to the client because of this policy. **For us, the missed disabling of Nagle's algorithm in secure sockets is a major bug.**

4.2 Secure Socket Creation

Encrypted Socket (**javax.net.ssl.SSLSocket**) characteristics are defined during the creation of a **IIOPSSLSocketFactory** object by *obtaining data from global variables* (which seems to be a bad behaviour) and storing those data into a specific private attribute of type **IIOPSSLSocketFactory.SSLInfo**. This means that the **Secure Socket Creation** depends on the surrounding context and not only on the parameters passed to the class methods.

4.3 Secure Server Socket Creation

The **SSLInfo** object necessary to have the informations about how to build the secure server socket are contained into an **IIOPSSLSocketFactory** attribute of type **java.util.Map** that associates a given **TCP port** to the relevant **SSLInfo** object. This **java.util.Map** is initialized from global variables (which seems again a bad habit) at **IIOPSSLSocketFactory** object creation time and stores the association of every **IIOP Listener** port to the relevant **IIOP Listener** configuration. Again, the usage of **public static methods** to obtain **global data** makes the functionality dependant on the surrounding context and not only to the parameters passed to the class methods.

4.4 Class Context Dependency

The entire class behaviour depends on the type of process in which context the **IIOPSSLSocketFactory** object is built. In fact, the **Secure Socket Creation** is both used in the **Application Client Container (as Client of a EJB Container)** and the **EJB Container (as Client of another EJB Container)**, but the **Secure Server Socket Creation** is used and can only be used (if contrary a **RuntimeException** is thrown) by the **EJB Container (as Server of Application Client Containers and EJB Containers)**. This is a **major mess in the software architecture**, the solution should have been designed in a completely different way, following the **OOP principles**.

4.5 No Generics Used

There's no usage of generics in the **java.util.Map** attribute **portToSSLInfo**.

```
    private Map portToSSLInfo = new Hashtable();
```

5 Appendix

5.1 Tools Used

1. **TeXstudio:** To write this LaTeX document.
2. **SVN:** To download GlassFish 4.1 source code.
3. **Notepad++, Editra:** To view the Java file source code.
4. **Eclipse Mars:** To view the Java project source code.
5. **SonarQube:** To analyse the Java project source code.
6. **SourceTree:** To guarantee team-work.

5.2 Hours Of Work

- **Alessandro:** 18
- **Alberto Mario:** 18

5.3 Revision History

Version Number	***Release Date***	***Changelog***
1.0	05/01/2016	Initial Release