Awake FILE v3.1 – Tutorial

November 16, 2015

Table of Contents

1 Fundamentals

Awake FILE is a secure Open Source framework that allows to program very easily file uploads/downloads and RPC through http. File transfers include powerful features like file chunking and automatic recovery mechanism.

Security has been taken into account from the design: server side allows
to specify strong security rules in order to protect the files and to
secure the RPC calls.

Awake FILE is licensed through the GNU Lesser General Public License (LGPL v2.1): you can use it for free and without any constraints in your open source projects and in your commercial applications.

The Awake FILE framework consists of:

- A Client Library.
- A Server Manager.
- User Configuration classes injected at runtime (start of server container).

The Client Library is installed on the client side - typically a PC or an Android device. The client application - typically a Desktop or Android application - accesses the remote files or java classes it through APIs. The execution of each Awake FILE command is conditioned by the rules defined in the User Configuration classes.

All communications between the PC and the Server are done using HTTP protocol on the standard 80 and 443 ports. (Communications may be secured using SSL/TLS).

This user guide covers:

- Configuration on the server side.
- Programming remote files access, RPC calls and file uploads/download on the client side.
- Awake FILE Internals.
- Advanced techniques.

1.1 Technical operating environment

Awake FILE is 100% written in Java, and functions identically under Android Microsoft Windows, Linux and all versions of UNIX supporting Java 6+ and Servlet 2.5.

The following environments are supported in this version:

JVM (Java Virtual Machine)	
Android	Dalvik 4.0.3+
Windows	Oracle Java SE 6, Java SE 7 and Java SE 8
UNIX/Linux	Oracle Java SE 6, Java SE 7 and Java SE 8 OpenJDK 6, OpenJDK 7, OpenJDK 8
OS X	Apple Java SE 6 for OS X. Oracle Java SE 7 for OS X 10.7.3+ Oracle Java SE 8 for OS X 10.8+

Servlet Containers
All Servlet containers that implement the Servlet 2.5+ specifications.

2 Installation

2.1 Installation files

Download and unzip `awake-file-3.1.x-bin.zip` or download and untar `awake-file-3.1.x-bin.tar.gz`.

2.2 Server installation

Add the jars of the `/lib-server` subdirectory to your Servlet container webapp library folder. (Typically in `/WEB-INF/lib)`.

Create a "Server" project and add the jars of the `/lib-server` subdirectory to your development `CLASSPATH`.

Maven users:

```
<groupId>org.awake-file</groupId>
<artifactId>awake-file-server</artifactId>
<version>3.1.x</version>
```

(`x` exact value is on download page).

Note: Awake FILE requires to define a Servlet in your `web.xml` configuration file. We will detail this in *3.2.3 Passing concrete Configurator classes*.

2.3 Client installation (including Android)

Create a "Client" project and add to your development `CLASSPATH` the path to the jars located in the `/lib-client` subdirectory of your installation folder.

Maven users:

```
<groupId>org.awake-file</groupId>
<artifactId>awake-file-client</artifactId>
<version>3.1.x</version>
```

(`x` exact value is on download page).

2.3.1 Android Project settings

Add the 3 following lines to your `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

2.3.2 Eclipse settings for Android project

1) Create a folder and copy `/lib-client` jars into the folder. (Not necessary to copy the jar into Android `libs.)`
2) Select all added jars, right click and “Build Path” ➔ “Add to Build Path”.
3) Make sure that all added jars are checked in “Java Build Path ➔ “Order and Export”.

Note: the source files archive `awake-file-3.1.x-src.zip` contains a ready to use Eclipse Android project: `awake-file-android-sample`

3 Configuration on the server side

The Configuration on the server side addresses:

- Client login and password verification.
- Basic Security settings.
- Defining the uploaded user files location on the server.
- RPC security settings.

3.1 The Server File Manager Servlet

All commands sent by the client side are received by the Server File Manager Servlet. The Manager Servlet then:

- Authenticates the client call.
- For file commands :
  - Executes the file command asked.
  - Sends the result of the statement back to the client side.
- For RPC commands :
  - If the RPC command matches the rules defined by the Awake Configurators (see below), the Java method command is executed. Otherwise, an Exception is sent back to the client
  - Sends the return value of the Java method to the client.

3.1.1 Configure the webapp web.xml

Add the Server File Manager Servlet to your web.xml:

```
<servlet>
	<servlet-name>ServerFileManager</servlet-name>
	<servlet-class>org.kawanfw.file.servlet.ServerFileManager</servlet-class>
	<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
	<servlet-name>ServerFileManager</servlet-name>
	<url-pattern>ServerFileManager</url-pattern>
</servlet-mapping>
```

3.2 Configurators fundamentals

All server configurations are done through Java classes called “Configurators” in our terminology. A Configurator is a user-developed Java class that implements one the Configurator Java interfaces. The Configurator instance is dynamically loaded by the Awake Server at bootstrap through DI (Dependency injection). The methods of the Configurator instance are then called internally by the Awake FILE Manager Servlet when necessary.

Awake FILE use 2 types of Configurators defined as Java interfaces:

- `CommonsConfigurator`.
- `FileConfigurator`.

Note that Awake FILE comes with Default Configurator classes:
it's not required to write your own Configurator
(in case you only want to test Awake FILE, etc.).
See the Quick Start.

3.2.1 CommonsConfigurator interface

This CommonsConfigurator interface allows to define basic access and security settings. A concrete implementation code will let you define:

- How to authenticate a remote client user
- How to secure the http session.
- The `java.util.logging.Logger` that Awake will use for logging.
- Advanced procedures in order to secure the service (IP Banning, session encryption).
- If needed for the authentication phase, how to get a `Connection` from the connection pool.

Awake FILE comes with a default `CommonsConfigurator`. implementation that may be extended: DefaultCommonsConfigurator.

3.2.2 FileConfigurator interface

The File Configurator interface lets you define the server settings for the files to upload/download and the for the client Java calls.

It allows to :

- Define the Awake FILE Server root directory.
- Define if each client username has his own root directory.
- Define a specific piece of Java code to analyse the method name and it's parameter values before allowing or not it's execution.
- Execute a specific piece of Java code if the method not allowed.

Note that Awake FILE comes with a default `FileConfigurator` implementation that is very liberal and should be extended: DefaultFileConfigurator

If no `FileConfigurator` is implemented, Awake loads and uses the `DefaultFileConfigurator` class.

3.2.3 Passing concrete Configurator classes

Your concrete implementations are passed to Awake as parameters of the `ServerFileManager` Servlet in your `web.xml` configuration file.

- The `CommonsConfiguratorClassName` parameter lets you define your `CommonsConfigurator` concrete implementation.
- The `FileConfiguratorClassName` parameter lets you define your `FileConfigurator` concrete implementation.

```
<servlet>
	<servlet-name>ServerFileManager</servlet-name>
	<servlet-class>org.kawanfw.file.servlet.ServerFileManager</servlet-class>

	<init-param>
		<param-name>CommonsConfiguratorClassName</param-name>
		<param-value>org.acme.config.MyCommonsConfigurator</param-value>
	</init-param>

	<init-param>
		<param-name>FileConfiguratorClassName</param-name>
		<param-value>org.acme.config.MyFileConfigurator</param-value>
	</init-param>
</servlet>

<servlet-mapping>
	<servlet-name>ServerFileManager</servlet-name>
	<url-pattern>ServerFileManager</url-pattern>
</servlet-mapping>
```

If you don't provide a parameter for a Configurator, Awake will use the corresponding Default Configurator.

3.3 Coding CommonsConfigurator

Now, let's code our own configuration methods in a concrete implementation of the CommonsConfigurator interface.

Create a class `MyCommonsConfigurator` that extends DefaultCommonsConfigurator. We will then implement our own methods.

3.3.1 Extracting a Connection from the pool

(This is not required and necessary only if you want to use SQL for the other methods of your `CommonsConfigurator` implementation).s

The default implementation `DefaultCommonsConfigurator.getConnection()` extract a `Connection` from a Tomcat JDBC Pool `<Resource>` configured in `context.xml`.

Download a Tomcat JDBC Pool configuration example: context.xml.

You may of course implement you own Connection Pool implementation by overloading `CommonsConfigurator.getConnection()`

3.3.2 Coding Basic Security settings

All the Basic Security Settings are optional. If a method is not implemented, the corresponding method of `DefaultCommonsConfigurator` will be called.

The basic security settings let you define through a concrete implementation of the `CommonsConfigurator` method:

- How to authenticate the remote (username, password) couple send by the client side.
- Whether the client must be in secured https.
- The list of banned usernames.
- The list of banned IPs.
- A secret value to reinforce the strength of the hash value used as authentication token.
- A password that will be used to decrypt the http request parameters sent by the client side.

We will explore some examples:

3.3.2.1 Login method: authenticating client username and password

You don't want your Awake FILE server be accessible to the whole world. Awake provides a mechanism that allows to check the username and password sent by the remote client program. This is done through the `login` method of `CommonsConfigurator` interface. If `login` returns true, access is granted.

The username and password should be checked against an applicative access mechanism, such as a LDAP directory, a login table in the database, etc.

The following example will check that the username and password passed by the client match an access list defined in a SQL table of your host database. Add the method in your `MyCommonsConfigurator` class:

```
    /**
     * Our own Acme Company authentication of remote client users. This methods
     * overrides the {@link DefaultCommonsConfigurator#login} method. <br>
     * The (username, password) values are checked against the user_login table.
     *
     * @param username
     *            the username sent by client side
     * @param password
     *            the user password sent by client side
     *
     * @return true if access is granted, else false
     */
    @Override
    public boolean login(String username, char[] password) throws IOException,
	    SQLException {
	Connection connection = null;

	try {
	    // Extract a Connection from our Pool
	    connection = super.getConnection();

	    // Compute the hash of the password
	    Sha1 sha1 = new Sha1();
	    String hashPassword = null;

	    try {
		hashPassword = sha1.getHexHash(new String(password).getBytes());
	    } catch (Exception e) {
		throw new IOException("Unexpected Sha1 failure", e);
	    }

	    // Check (username, password) existence in user_login table
	    String sql = "SELECT username FROM user_login "
		    + "WHERE username = ? AND hash_password = ?";
	    PreparedStatement prepStatement = connection.prepareStatement(sql);
	    prepStatement.setString(1, username);
	    prepStatement.setString(2, hashPassword);

	    ResultSet rs = prepStatement.executeQuery();

	    if (rs.next()) {
		// Yes! (username, password) are authenticated
		return true;
	    }

	    return false;
	} finally {
	    // Always free the Connection so that it is put
	    // back into the pool for another user
	    if (connection != null) {
		connection.close();
	    }
	}
    }
```

Note: the included SshAuthCommonsConfigurator is a concrete `CommonsConfigurator` that extends `DefaultCommonsConfigurator`.
It allows zero-code client `(usernname, password)` authentication using SSH.

3.3.2.2 forceSecureHttp method: define if the http request must be in SSL/TLS

It is better to encrypt with SSL/TLS all the http exchange between the client side and the server. However, this is not mandatory. If you do not need SSL/TLS encryption (for instance in an Intranet scenario), there is nothing to do : the exchange won't be encrypted.

If you want to force https encryption with SSL/TLS, just add the following method in your `MyCommonsConfigurator` class:

```
    /**
     * @return <code>true</code>.
     *
     *          Our client programs will be forced to use https in the url parm
     *          of the RemoteSession constructor
     */
    @Override
    public boolean forceSecureHttp() {
         return true;
    }
```

If the client tries to connect to the Awake FILE server using a "http" scheme in the URL instead of a https address, the server will send back to the client the order to update the scheme to "https". This will be done silently by the client side, prior authentication: the username & password will thus be sent encrypted with SSL. All subsequent server calls in the session will be in secure https.

3.3.2.3 getBannedUsernames method: define banned users

If you want to ban some users, just add a `Set<String> getBannedUsernames()` method in your `MyCommonsConfigurator` class.
Let's say we want to ban two users :

```
    /**
     * These usernames must not access our databases: user1 & user2 (In real
     * world case we would retrieve the usernames from a database or a file).
     *
     * @return the usernames that are banned from our server.
     */
    @Override
    public Set<String> getBannedUsernames() throws IOException, SQLException {
       Set<String> set = new HashSet<String>();
       set.add("user1");
       set.add("user2");
       return set;
    }
```

The `CommonsConfigurator` interface contains other methods that you should implement in your `MyCommonsConfigurator` class to strengthen the security of your Awake FILE configuration:

`getIPsBlacklist:`
Implement this method if you want to ban some IP.

`getIPsWhitelist:`
Implement this method if you want to authorize only a restricted list of IPs

`addSecretForAuthToken:`
Implement this method in order to reinforce the security of the authentication mechanisms. (See *5.1 Session security*)

`getEncryptionPassword:`
Implement this method to encrypt all http request parameters sent from the client to the server for security reason (obfuscation and transport encryption. See 0

Http session encryption.)

Please check the Javadoc of Commons Configurator for more information.

3.4 Coding FileConfigurator

The Awake FILE Server settings are coded in a concrete implementation of the FileConfigurator interface with the methods:

- `getServerRoot`.
- `useOneRootPerUsername`.
- `allowCallAfterAnalysis`.
- `runIfCallRefused`.

Note that FileConfigurator comes with a default implementation: DefaultFileConfigurator. If no `FileConfigurator` is implemented, Awake loads and uses the `DefaultFileConfigurator` class.

3.4.1 Defining the user files locations

3.4.1.1 Defining Awake FILE Server root directory

`getServerRoot` allows to define the root directory of the Awake FILE Server, aka where the users files will be stored when uploaded.

Code the root directory you want to use in your `FileConfigurator` implementation. Code `null` if you don't want to define a server root and you always want to upload the files exactly where the client side defines it.

3.4.1.2 Define if each client username has his own root directory

Code your choice in `useOneRootPerUsername` by returning `false` or `true`.

If `useOneRootPerUsername` returns `false`: the location of the file is independent of the username and depends only on `getServerRoot`.

If `useOneRootPerUsername` returns `true`: the name of the username will be used as root directory per user.

3.4.1.3 Example values of getServerRoot & useOneRootPerUsername.

Let's suppose a client is running Windows and that the server is Unix.
The client sends the following upload order :

```
RemoteSession remoteSession = new RemoteSession(url, username,
  password);

// OK: upload a file
remoteSession.upload(new File("c:\\myFile.txt"),"/mydir/myFile.txt");
```

The real location of the uploaded file will depend on `getServerRoot` & `useOneRootPerUsername` values.

Case 1 - `getServerRoot` returns `null`:

The file will be uploaded in `/mydir/myFile.txt`.
(The value of `useOneRootPerUsername` has no effect.)

Case 2 - `getServerRoot` returns a directory:

The file location will thus depends on `useOneRootPerUsername` return value:

case 2.a - `useOneRootPerUsername` returns `false`:

The file will be uploaded in:
`/server-root/mydir/myFile.txt`.

Where :
`server-root` is the directory returned by `getServerRoot`.

case 2.b - `useOneRootPerUsername` returns `true:`

The file will be uploaded at:
`/server-root/username/mydir/myFile.txt.`

Where :

- `server-root` is the directory returned by `getServerRoot.`
- `username` is the username passed in `new RemoteSession()` constructor on the client side.

3.4.2 RPC rules for calling Java methods from the client side

The RPC mechanism has been designed to be very simple to use but sill secure.

3.4.2.1 How to declare your server class as callable by the client side

Any server Java class method maybe called by the client side if the server class follows the following requirements:

- The class must implement the ClientCallable or ClientCallableNoAuth interface.

- The class must have a default visible constructor with no parameters. (It is the constructor that will be invoked by the Awake FILE Manager servlet).

`ClientCallable` and `ClientCallableNoAuth` are marker interfaces and have no method: they allow to indicate to Awake FILE Manager that the classes are callable from the client side.

The interface to implement depends on your authentication need:

- Implement `ClientCallable` if you require that users must be authenticated to use the class.
- Implement `ClientCallableNoAuth` if you want your class to be callable by any user, without authentication.

3.4.2.2 RPC Security check

FileConfigurator provides two methods to secure the RPC mechanism

- `allowCallAfterAnalysis`: allows to define a specific piece of Java code to analyze the method name and it's parameter values before allowing or not it's execution. If you want the execution to be disallowed, return false after the analysis. It will raise an Exception on the client side.

- `runIfCallRefused`: Executes a specific piece of Java code if the method not allowed (`runIfCallRefused` is only called if `allowCallAfterAnalysis` returns false).

Let's say we want to allow out client users to call our `AccountDeletor` server class :

```
public class AccountDeletor implements ClientCallable {

    /**
     * Constructor with no parameters (required by Awake RPC mechanism)
     */
    public AccountDeletor() {
    }

    /**
     * Deletes an account.
     *
     * @param username
     *            the username of the account to delete
     * @return true if the account is deleted
     */
    public boolean deleteAccount(String username) {

      // code to delete the account.
      // ...

      return true;
    }
}
```

We want to double-check that the username is the real one, and not another username (case an attacker would modify the client program call after reverse engineering and change the username parameter).

If an illegitimate username is detected, discard the username and log his IP as a banned IP.

Let's create a `MyFileConfigurator` that extend `DefaultFileConfigurator` and implement `allowExecuteAfterAnalysis:`

```
    /**
     * We will analyse in the example only the
     * org.acme.server.AccountDeletor.deleteAccount method
     *
     * @return true only if the username parameter of deleteAccount is identical
     *         to the username of the login method.
     */
    @Override
    public boolean allowCallAfterAnalysis(String username,
            Connection connection, String methodName, List<Object> params)
            throws IOException, SQLException {

        // We will check that the account to delete is the real one
        // aka, a user has not forged the call to delete another user!

        if (methodName.equals("org.acme.server.AccountDeletor.deleteAccount")) {

            String passedUsername = params.get(0).toString();

            if (username.equals(passedUsername)) {
                return true;
            } else {
                // Do not allow the call!
                return false;
            }

        } else {
            // We don' analyse other methods for our example.
            return true;
        }
    }
```

We can now implement `runIfExecuteDisallowed` that will ban our user:

```
    /**
     * The parent method will be called and the username will be added to the
     * BANNED_USERNAMES table.
     */
    @Override
    public void runIfCallRefused(String username, Connection connection,
            String ipAddress, String methodName, List<Object> params)
            throws IOException, SQLException {

        // 1) Call default implementation. It will log the event:
        super.runIfCallRefused(username, connection, ipAddress, methodName,
                params);

        // 2) Insert the username & it's IP into the banned usernames table
        String sqlOrder = "INSERT INTO BANNED_USERNAMES VALUES (?, ?)";

        PreparedStatement prepStatement = connection.prepareStatement(sqlOrder);
        prepStatement.setString(1, username);
        prepStatement.setString(2, ipAddress);

        prepStatement.executeUpdate();
        prepStatement.close();
    }
```

3.5 Testing you server configuration

After restarting you server, type the http address of the ServerFileManager Servlet in a browser:

```
http://www.yourhost.com/path-to-servlet/ServerFileManager
```

It will display your configuration and a status line that should display: OK & Running.

If not, the configuration errors are detailed in red for correction.

4 Client Side Programming

The client API is composed of 4 main classes:

Name	Role
`RemoteSession`	Main class for establishing an http session with a remote host and for executing from client side some basic operations
`RemoteFile`	Allows to execute `java.io.File` methods on remote files.
`RemoteInputStream`	Obtains input bytes from a remote file. Used to download a remote file.
`RemoteOutputStream`	An output stream for writing data to a remote file. Used to upload a file.

4.1 The RemoteSession class

RemoteSession is the main class for establishing an http session with a remote host.

The `RemoteSession` instance is passed to all other client API classes and is used by them for session identification and authentication.

`RemoteSession` is also used and for executing some basic client side operations:

- Get the Java version of the servlet container on the remote server.
- Call remote Java methods.
- Upload files by wrapping bytes copy from a `FileInputStream` to a `RemoteOutputStream`.
- Download files by wrapping bytes copy from a `RemoteInputStream` to a `FileOutputStream`.
- Returns with one call the length of a list of files located on the remote host.

4.1.1 Session creation & authentication

The `RemoteSession` constructor takes at least three parameters:

- `url`: the URL of the path to the Awake FILE Manager Servlet.
- `username`: the user username for authentication.
- `password`: the authentication password.

The `RemoteSession` is created only if the Awake FILE Manager has authenticated the user. The authentication is done by invoking the `login` method of the `CommonsConfigurator` instance.

Example of `RemoteSession` creation:

```
        // Define URL of the path to the ServerFileManager servlet
        String url = "https://www.acme.org/ServerFileManager";

        // The login info for strong authentication on server side
        String username = "myUsername";
        char[] password = { 'm', 'y', 'P', 'a', 's', 's', 'w', 'o', 'r', 'd' };

        // Establish a session with the remote server
        RemoteSession remoteSession = new RemoteSession(url, username, password);
```

4.1.2 Defining a proxy

Communication via an (authenticating) proxy server is done using a `java.net.Proxy` instance. If proxy requires authentication, pass the credentials via a `java.net.PasswordAuthentication` instance:

```
        Proxy proxy = new Proxy(Proxy.Type.HTTP, new InetSocketAddress(
                "proxyHostname", 8080));

        PasswordAuthentication passwordAuthentication = null;

        // If proxy require authentication:
        passwordAuthentication = new PasswordAuthentication("proxyUsername",
                "proxyPassword".toCharArray());

        RemoteSession remoteSession = new RemoteSession(url, username,
                password, proxy, passwordAuthentication);
        // Etc.
```

See *7.2 Defining a proxy with Java 7+* for possible troubleshooting.

4.1.3 Handling Exceptions thrown by RemoteSession constructors

`RemoteSession` constructors throw exclusively the following exceptions:

Table of Exceptions thrown by RemoteSession constructors

Exception	Signification
`java.net.UnknownHostException`	The URL is malformed.
InvalidLoginException	The (username, password) authentication failed on the remote Awake FILE Manager. (The implemented `CommonsConfigurator.login()` method returned `false`).
`java.net.UnknownHostException`	• There is no Internet Connection. • There is an error in http address in the URL parameter.
`java.net.ConnectException`	The Http Request returned a Http Status Code != OK (200). Use `RemoteSession.getHttpStatusCode()` to retrieve the Status Code.
`java.net.SocketException`	Network failure during transmission.
`java.lang.SecurityException`	The URL scheme is not https as requested by the remote Awake FILE Manager. (The implemented `CommonsConfigurator.forceSecureHttp()` method returned `true`).
RemoteException	An unexpected Exception has been thrown by the server. Signals an Awake product failure.
`java.io.IOException`	For all other IO / Network / System Error.

See the RemoteSessionExceptionDecoder.java class for a complete example of Exception handling.

4.1.4 RPC: Calling Java methods on the Awake FILE Server

Use `RemoteSession.call` to call a remote Java method:

- First parameter is the full name of the method to call with the notation : `packageName.className.methodName`.
  Example: `org.kawanfw.examples.Calculator.add`
- All following parameters are optional and are the parameters of the remote method.
- The result is always returned as a String.

The requirements for the Java class on the server are explained in 3.4.2.1 How to declare your server class as callable by the client side

Example of a `RemoteSession.call`:

```
        // OK: call the add(int a, int b) remote method that returns a + b:
        String result = remoteSession.call(
              "org.kawanfw.file.test.api.server.Calculator.add" ", 33, 44);
        System.out.println("Calculator Result: " + result);
```

Code of the remote `org.kawanfw.examples.Calculator` class:

```
package org.kawanfw.file.test.api.server;

import org.kawanfw.file.api.server.ClientCallable;

/**
 *
 * Simple calculator to be called from the client side.
 * Requires the client to be authenticated.
 */

public class Calculator implements ClientCallable {

    /**
     * Constructor
     */
    public Calculator() {

    }

    public int add(int a, int b) {
      return (a + b);
    }
```

4.1.5 Uploading & Downloading files

The two RemoteSession APIs allow to upload or download a file:

- RemoteSession.upload(File file, String pathname)
- RemoteSession.download(String pathname, File file)

As these APIs wrap `RemoteInputStream` and `RemoteOutputStream`, we will explain the internal mechanism of files transfer in their dedicated chapters.

4.1.5.1 Syntax of the remote pathname

The remote pathname strings must include the complete path to the file. Example :

```
remoteSession.upload(new File("c:\\myFile.txt"),
            "/home/mylogin/myFile.txt");
```

Note that the real location of the remote file `"/home/mylogin/myFile.txt"` will depend on your FileConfigurator implementation, as explained in 3.4.1 Fundamentals.

4.1.6 Handling Exceptions thrown by RemoteSession methods

Table of Exceptions thrown by RemoteSession methods

Exception	Signification
`java.net.UnknownHostException`	There is no more Internet Connection.
InvalidLoginException	The Connection has been closed.
`java.net.ConnectException`	The Http Request returned a Http Status Code != OK (200). Use `RemoteSession.getHttpStatusCode()` to retrieve the Status Code.
`java.net.SocketException`	Network failure during transmission.
`java.lang.SecurityException`	Happens when a call() is refused by the server FileConfigurator.allowCallAfterAnalysis concrete implementation.
RemoteException	An unexpected Exception has been thrown by the server. Signals an Awake product failure.
`IOException`	For all other IO / Network / System Error.

4.2 The RemoteFile class

Remote File is the main class for accessing remote files.

`RemoteFile` methods have the same names, signatures and behaviors as `java.io.File` methods:
a `RemoteFile` method is a `File` method that is executed on the remote host.

The few differences with `File` are:

- There is only one constructor that takes as parameter a `RemoteSession` and a `pathname`. The `pathname` is defined with a "/" separator on all platforms and must be absolute.
- `File.compareTo(File)` and `File.renameTo(File)` methods take a `RemoteFile` parameter in this class.
- File methods that return `File` instance(s) return `RemoteFile` instance(s) in this class.
- The `File.toURI()` and `File.toURL()` methods are meaningless in this context and are thus not ported.
- The static `File.listRoots()` and `File.createTempFile()` are not ported.
- The Java 7+ `File.toPath()` method is not ported as this Awake FILE version does not support remote `Path` objects.

Note that the real pathname used on host for File method execution depends on the Awake FILE configuration of `FileConfigurator.getServerRoot()` and `FileConfigurator.useOneRootPerUsername()`. This follow the same principle of FTP server mechanisms.

Example:

If `FileConfigurator.getServerRoot()` returns `/home/` and `FileConfigurator.useOneRootPerUsername()` returns `true` and client username is "`mike`":

```
        new RemoteFile(remoteSession, "/myfile.txt").exists();
```

will return the result of the execution on remote host of:

```
        new File("/home/mike/myfile.txt").exists();
```

This is described in detail at: *3.4.1 Defining the user files locations*

Note: the user rights are the rights of the servlet container when accessing remote files.

4.2.1 Using FilenameFilter and FileFilter

When using `FilenameFilter` and `FileFilter` filters, the filter implementation must follow these rules:

- The filter must implement `Serializable`.
- Thus, the filter class must already exist on the server side.
- When using anonymous inner class for `FilenameFilter` or `FileFilter`: it must be public static.

4.2.2 Handling Exceptions thrown by RemoteFile methods

`RemoteFile` methods throw corresponding `java.io.File` Exceptions plus the following Exceptions that are wrapped by a `RuntimeException` (except `java.lang.SecurityException` which is not wrapped).

Table of Exceptions thrown by RemoteFile **methods**

Exception	Signification
`java.net.UnknownHostException`	There is no more Internet Connection.
InvalidLoginException	The Connection has been closed.
`java.net.ConnectException`	The Http Request returned a Http Status Code != OK (200). Use `RemoteSession.getHttpStatusCode()` to retrieve the Status Code.
`java.net.SocketException`	Network failure during transmission.
`java.lang.SecurityException`	The remote `java.io.File` throwed a `java.lang.SecurityException`.
RemoteException	The remote `java.io.File` throwed an `Exception`.
`IOException`	For all other IO / Network / System Error.

4.2.3 Example of RemoteFile usage

```
// Define URL of the path to the ServerFileManager servlet
String url = "https://www.acme.org/ServerFileManager";

// The login info for strong authentication on server side
String username = "myUsername";
char[] password = { 'm', 'y', 'P', 'a', 's', 's', 'w', 'o', 'r', 'd' };

// Establish a session with the remote server
RemoteSession remoteSession = new RemoteSession(url, username, password);

// Create a new RemoteFile that maps a file on remote server
RemoteFile remoteFile = new RemoteFile(remoteSession, "/Koala.jpg");

// We can use all the familiar java.io.File methods on our RemoteFile
if (remoteFile.exists()) {
    System.out.println(remoteFile.getName() + " length  : "
          + remoteFile.length());
    System.out.println(remoteFile.getName() + " canWrite: "
          + remoteFile.canWrite());
}

// List files on our remote root directory
remoteFile = new RemoteFile(remoteSession, "/");

RemoteFile[] files = remoteFile.listFiles();
for (RemoteFile file : files) {
    System.out.println("Remote file: " + file);
}

// List all text files in out root directory
// using an Apache Commons IO 2.4 FileFiter
FileFilter fileFilter = new SuffixFileFilter(".txt");

files = remoteFile.listFiles(fileFilter);
for (RemoteFile file : files) {
    System.out.println("Remote text file: " + file);
}
```

4.3 The RemoteInputStream and RemoteOutputStream classes

A RemoteInputStream obtains input bytes from a remote `File`.
The remote file bytes are read with standards `InputStream` read methods and can thus be downloaded into a local file.

A RemoteOutputStream is an output stream for writing data to a remote `File`
It allows to create a remote file by writing bytes on the `RemoteOutputStream` with standards `OutputStream` write methods.

These classes are provided to :

1. Offer APIs with no learning curve.
2. Allow easy existing code migration, because they implement `InputStream` and `OutputStream`.
3. Allow easy use of progress bars for file uploads and downloads.

`RemoteInputStream` & `RemoteInputStream` implement automatic file chunking and recovery mechanism as described in *5.2 Data transport*.

They are in fact the underlying classes used by `RemoteSession.download` and `RemoteSession.upload`.

`RemoteSession.download(String pathname, File file)` wraps byte copy from a `RemoteInputStream` to a `FileOutputStream`.

Extract:

```
        InputStream in = null;
        OutputStream out = null;

        // (IOUtils is a general IO stream manipulation utilities
        // provided by Apache Commons IO)

        try {
            in = new RemoteInputStream(this, pathname);
            out = new BufferedOutputStream(new FileOutputStream(file));
            IOUtils.copy(in, out);
            // Cleaner to close in here so that no Exception is thrown in
            // finally clause
            in.close();
        } finally {
            IOUtils.closeQuietly(in);
            IOUtils.closeQuietly(out);
        }
```

`RemoteSession.upload(File file, String pathname)` wraps byte copy from a `FileInputStream` to a `RemoteOutputStream`.

Extract:

```
        InputStream in = null;
        OutputStream out = null;

        // (IOUtils is a general IO stream manipulation utilities
        // provided by Apache Commons IO)

        try {
            in = new BufferedInputStream(new FileInputStream(file));
            out = new RemoteOutputStream(this, pathname, file.length());
            IOUtils.copy(in, out);
            // Cleaner to close out here so that no Exception is thrown in
            // finally clause
            out.close();
        } finally {
            IOUtils.closeQuietly(in);
            IOUtils.closeQuietly(out);
        }
```

4.4 Using Progress Bars with file upload & download

Simply use `RemoteInputStream` or `RemoteOutputStream` to download or upload the files, and increment the Progress Bar in the read/write loop.

At the end of the read/write, set the Progress Bar progress indicator to 100:

Example:

Declare the global variables used by the Progress Monitor and in the read/write loop:

```
    /**
     * Progress between 0 and 100. Updated by doFileUpload() at each 1% input
     * stream read
     */
    private int progress = 0;

    /** Says to doFileUpload() code if transfer is cancelled */
    private boolean cancelled = false;
```

The `doFileUpload` method is called to upload a file:

```
    /**
     * Do the file upload.
     */
    private void doFileUpload() {
        try {

            // BEGIN MODIFY WITH YOUR VALUES
            String userHome = System.getProperty("user.home");

            String url = "http://localhost:8080/awake-file/ServerFileManager";
            String username = "username";
            char[] password = "password".toCharArray();

            File file = new File(userHome + File.separator + "image_1.jpg");
            String pathname = "/image_1_1.jpg"; // remote file path
            // END MODIFY WITH YOUR VALUES

            RemoteSession remoteSession = new RemoteSession(url, username,
                    password);

            long fileLength = file.length();
            InputStream in = null;
            OutputStream out = null;

            try {

                in = new BufferedInputStream(new FileInputStream(file));
                out = new RemoteOutputStream(remoteSession, pathname,
                        fileLength);

                int tempLen = 0;
                byte[] buffer = new byte[1024 * 4];
                int n = 0;

                while ((n = in.read(buffer)) != -1) {
                    tempLen += n;

                    // Test if user has cancelled the upload
                    if (cancelled)
                        throw new InterruptedException(
                                "Upload cancelled by User!");

                    // Add 1 to progress for each 1% upload
                    if (tempLen > fileLength / 100) {
                        tempLen = 0;
                        progress++;
                    }

                    out.write(buffer, 0, n);
                }

                out.close();

            } finally {
                // When finished, set to the maximum value to stop the
                // ProgressMonitor
                progress = 100;
                if (in != null)
                    in.close();
                if (out != null)
                    out.close();
            }

            remoteSession.logoff();

            System.out.println("File upload done.");
        } catch (Exception e) {

            if (e instanceof InterruptedException) {
                System.out.println(e.getMessage());
                return;
            }

            System.err.println("Exception thrown during Upload:");
            e.printStackTrace();
        }
    }
```

Assuming you want to display a progress indicator using `SwingWorker`, you would start as a `Thread` the previous code. To update the progress bar, the `SwingWorker.doInBackground()` method would be overridden :

```
    @Override
    public Void doInBackground() {
        // Reset values at each upload
        cancelled = false;
        progress = 0;
        setProgress(0);

        // progress is ++ at each
        // 1% file transfer in doFileUpload()
        while (progress < 100) {
            try {
                Thread.sleep(50);
            } catch (InterruptedException ignore) {
            }

            // Say to doFileUpload() that
            // user has cancelled the upload
            if (isCancelled()) {
                cancelled = true;
                break;
            }

            setProgress(Math.min(progress, 100));
        }

        return null;
    }
```

A complete example is available in FileTransferProgressMonitorDemo.java

5 Awake FILE internals

This chapter describes some technical and implementation aspects of Awake FILE.

5.1 Session security

After server authentication succeeds (through the CommonsConfigurator.login method), the Awake Server Manager builds an hexadecimal Authentication Token that is send back to the client and will be used by each following client call in order to authenticate the calls.

5.1.1 The default Authentication Token creation

The default mechanism to build an Authentication Token is coded in the method DefaultCommonsConfigurator.computeAuthToken.

The Authentication Token value built by `computeAuthToken` on server side is:

`SHA-1(username + secretValue)` first 20 hexadecimal characters.

where:
- `username`: the username of the client.
- `secret` : a secret value defined by the implementation of method `public String addSecretForAuthToken()` in your `CommonsConfigurator` implementation.

A each client call, the Authentication Token is sent to the Awake Server along the requests. The Awake Server Manager then verifies that Authentication Token verifies:

```
Authentication Token = SHA-1(username + secretValue) first 20
hexadecimal characters.
```

If the `Authentication Token` does not match, the session is discontinued.

This mechanism, which uses SHA-1 strong cryptography, makes impossible for an attacker to forge a session without a legitimate (username, password) couple.

5.1.2 Creating your own Authentication Token

If you want to create the Authentication Token with your own enforced security rules (adding random or secret values stored in database, etc.) : override the method `DefaultCommonsConfigurator.computeAuthToken` in your own `CommonsConfigurator` implementation.

5.2 Data transport

5.2.1 Transport format

Awake transfers the least possible meta-information :

- Request parameters are transported in UTF-8 format.
- JSON format is used for data & classes transport (using Google Gson library).

5.2.2 Content Streaming & memory management

All requests are streamed, especially for file uploads and downloads:

- Output requests (from client side) are streamed to avoid buffering any content body by streaming directly to the socket to the server.
- input response (for client side) are streamed efficiently read the response body by streaming directly from the socket to the server.

5.2.3 Chunked upload of large files

Large files are split in chunks that are uploaded in sequence when using RemoteOutputStream or `RemoteSession.upload()`. The default chunk length is 10Mb. You can change the default value with SessionParameters.setUploadChunkLength(long) before passing `SessionParameters` to `RemoteSession` class constructor.

SessionParameters usage is described in *6.3 Session parameters*

5.2.4 Chunked download of large files

Large files are split in chunks that are downloaded in sequence when using RemoteInputStream or `RemoteSession.download()`. The default chunk length is 10Mb. You can change the default value with SessionParameters.setDownloadChunkLength(long) before passing `SessionParameters` to `RemoteSession` class constructor.

5.2.5 File chunking and statefull/stateless architecture

Note that file chunking requires that all chunks be uploaded/downloaded to/from to the same web server. Thus, file chunking does not support true stateless architecture with multiple identical web servers. If you want to set a full stateless architecture with multiple identical web servers, you must disable file chunking.

This is done by setting a 0 upload and download chunk length value using:

- SessionParameters.setUploadChunkLength(long)
- SessionParameters.setDownloadChunkLength(long)

5.2.6 Large file upload & download recovery

Awake FILE supports recovery for large files upload and download.

In case of recoverable I/O or communication Exception, aka `SocketException`, the recall of the upload or download sequence will restart the transfer at the last chunk non completely transmitted.

The only condition is to recall the upload/download in the same JVM run (so recovery will not be supported if application is completely stopped and restarted.)

For example, when using default chunk length of 10Mb: if the upload of a 2Gb file is interrupted at 1,8Gb, only the remaining 200Mb will be resent when re-invoking `RemoteSession.upload()` or `RemoteOutputStream` sequence in the same JVM life cycle.

Chunks are stored in temporary directories. See *6.7 Managing temporary files* for more information.

5.2.7 Upload and download recovery simple code example

These simple examples show a minimalist way to recover from network failure. More sophisticated code would also trap IOExceptions and recover other unexpected I/O or security/system errors, etc.

```
// Uploads a big file. If a SocketException (including subclass
// ConnectException) is raised, then allow to continue upload starting
// from the last chunk that was not uploaded
boolean continueUpload = true;

while (continueUpload) {
    try {
      remoteSession.upload(bigFile, "/" + bigFile.getName());
      continueUpload = false;
    } catch (SocketException se) {

      String title = "Warning";
      String text = "File upload interrupted because of network error. "
            + "Do you want to retry and continue upload?";

      int result = JOptionPane.showConfirmDialog(null, text, title,
            JOptionPane.YES_NO_OPTION);
      if (result != JOptionPane.YES_OPTION) {
          continueUpload = false;
      }
    }
}
```

```
// Downloads a big file. If a SocketException (including subclass
// ConnectException) is raised, then allow to continue download starting
// from the last chunk that was not downloaded
boolean continueDownload = true;

while (continueDownload) {
    try {
      remoteSession.download(, "/" + bigFile.getName(), bigFile);
      continueDownload = false;
    } catch (SocketException se) {

      String title = "Warning";
      String text = "File download interrupted because of network error. "
            + "Do you want to retry and continue download?";

      int result = JOptionPane.showConfirmDialog(null, text, title,
            JOptionPane.YES_NO_OPTION);
      if (result != JOptionPane.YES_OPTION) {
          continueDownload = false;
      }
    }
}
```

5.3 Stateless session management

The Awake FILE Manager Servlet is stateless: no user or session info are stored on the server. This allows to configure Awake FILE with any Http load balancing & failover services.

As explained in *5.2.5 File chunking and statefull/stateless architecture*, stateless session management require that files are *not* chunked when uploaded.

6 Advanced techniques

6.1 Using multiple RemoteSession in Threads

You may use multiple different `RemoteSession` in your programs. And you may use them in background threads.

Please note that `RemoteSession` is not thread safe: Only one thread may access an `RemoteSession` instance at a time. Otherwise, results are unpredictable.

However, `RemoteSession` is cloneable : just clone your current `RemoteSession` to get a new one to use for simultaneous file operations:

```
        // Establish a session with the remote server
        RemoteSession remoteSession = new RemoteSession(url, username, password);

        // Establish a secondary RemoteSession for background thread:
        RemoteSession secondaryRemoteSession = remoteSession.clone();
```

6.2 Using RemoteSession to different Awake FILE Servers

You may use multiple `RemoteSession` that access different Awake FILE Servers in the same program.

There is nothing to set on the client side. Simply use different `url` parameters in your `RemoteSession` constructors.

For each `url` defined, there must be a corresponding Server File Manager Servlet on the server side:

```
        // The main Server File Manager
        String url = "https://www.acme.org/ServerFileManager";

        // The second Server File Manager
        String url2 = "https://www.acme.org/ServerFileManager2 ";

        // The login info for strong authentication on server side
        // (Assuming it's the same for the two Server File Managers)
        String username = "myUsername";
        char[] password = { 'm', 'y', 'P', 'a', 's', 's', 'w', 'o', 'r', 'd' };

        // Establish a session with the first remote server
        RemoteSession remoteSession = new RemoteSession(url, username,
               password);

        // Establish a session with the second remote server
        RemoteSession remoteSession2 = new RemoteSession(url2,
               username, password);
```

There is some configuration on the server side: a second Server File Manager Servlet must be defined in `web.xml` with the corresponding new Configurators classes passed as parameters to the new Servlet:

```
<servlet>
	<servlet-name>ServerFileManager2</servlet-name>
	<servlet-class>org.kawanfw.file.servlet.ServerFileManager</servlet-class>

	<init-param>
		<param-name>CommonsConfiguratorClassName</param-name>
		<param-value>org.acme.config.MyCommonsConfigurator2</param-value>
	</init-param>

	<init-param>
		<param-name>FileConfiguratorClassName</param-name>
		<param-value>org.acme.config.MyFileConfigurator2</param-value>
	</init-param>
</servlet>

<servlet-mapping>
	<servlet-name>ServerFileManager2</servlet-name>
	<url-pattern>ServerFileManager2</url-pattern>
</servlet-mapping>
```

6.3 Session parameters

SessionParameters allows also to define some settings for the Awake FILE session:

- Timeout value, in milliseconds, to be used when opening a communications link with the remote server. Defaults to 0 (no timeout).

- Read timeout, in milliseconds, that specifies the timeout when reading from remote Input stream. Defaults to 0 (no timeout).

- Password to use for encrypting all parameters request between client and remote host.

- Boolean to say if Clob upload/download using character stream or ASCII stream must be html encoded. Defaults to true.

- Boolean to say if http content must be compressed. Defaults to true.

- Download chunk length to be used by RemoteInputStream. Defaults to 10Mb. 0 means files are not chunked.

- Upload chunk length to be used by RemoteOutputStream Defaults to 10Mb. 0 means files are not chunked.

- Boolean to say if client sides allows HTTPS call with all SSL Certificates, including "invalid" or self-signed Certificates. Defaults to false.

- Maximum authorized length for a string for upload or download (in order to avoid OutOfMemoryException on client and server side.) Defaults to 2 Mb.

This example shows how to change some timeout default values:

```
String url = "https://www.acme.org/ServerFileManager";
String username = "myUsername";
char[] password = { 'm', 'y', 'P', 'a', 's', 's', 'w', 'o', 'r', 'd' };

SessionParameters sessionParameters = new SessionParameters();

// Sets the timeout until a connection is established to 10 seconds
sessionParameters.setConnectTimeout(10);

// Sets the read timeout to 60 seconds
sessionParameters.setReadTimeout(60);

// We will use no proxy
Proxy proxy = null;
PasswordAuthentication passwordAuthentication = null;

RemoteSession remoteSession = new RemoteSession(url, username,
      password, proxy, passwordAuthentication, sessionParameters);
// Etc.
```

6.4 Http session encryption

All values sent with the http request from the client side can be encrypted. They will be decrypted by the Server File Manager Servlet. The encryption algorithm is AES 256 bit.

You have to define a common symmetric password that will be used on the client side and the server side.

Client side

Password is set using SessionParameters:

```
        SessionParameters sessionParameters = new SessionParameters();

        char[] passwdHttp = { 'h', 't', 't', 'p', 'P', 'a', 's', 's', 'w', 'd' };
        sessionParameters
                .setEncryptionPassword(passwdHttp);

        // Establish the RemoteSession using the modified Http parameters:
        RemoteSession remoteSession = new RemoteSession(url, username,
                password, httpProxy, sessionParameters);

        // Etc.
```

Server Side

Add a `getEncryptionPassword` `method` to your CommonsConfigurator implementation :

```
    @Override
    public char[] getEncryptionPassword() {
        char[] passwdHttp = { 'h', 't', 't', 'p', 'P', 'a', 's', 's', 'w', 'd' };
        return passwdHttp;
    }
```

6.5 How to obfuscate your apps on the client side

Obfuscation of your client code may be necessary if your distribute your application on the Internet.

You may (and should) obfuscate all your Awake FILE code. This is supported and we have done it in several real projects.

The obfuscation rules are the same for Android Edition and Desktop Edition:

- Remove from the jar to obfuscate all the classes in packages that include `server` or `servlet` sub names.
- Apache Commons FileUpload `commons-fileupload-x.y.z.jar` and JSch `jsch-x.y.z.jar` should be removed.
- The third parties libraries compiled code should however remain in clear:
    - Apache Commons Lang & Commons IO.
    - JSON.simple.
    - Google Gson.

Note that by using http session encryption as described before, it would be impossible to an attacker to trap your Awake FILE code using probes in modified versions of the third parties libraries classes: all requests strings are encrypted before they are passed to them. So, always use http session encryption if you want to set a real strong obfuscation.

You may of course modify all Awake FILE and third parties source code if you want to obfuscate all code with your own techniques and completely hide and obfuscate the third party libraries. You may do it in compliance to the licenses of Awake (LGPL v2.1) and other third parties (Apache License Version 2.0.)

6.6 Anonymous Notifications to Kawan Servers

The Server File Manager Servlet will notify our remote Kawan servers that a user has succeeded his first login. This is done once during the web server JVM session per client user login, at first login. It is also done in a separated and secured thread: your Server File Manager Servlet will not be slowed down by the notification and no Exceptions will be thrown. Notification contains only anonymous data that are not reversible and thus cannot identify your server: hash value of your server ip address and user login count. There are no notifications for localhost or 127.0.0.1 server name.

Please note that the notification mechanism is important for us as software editor: it says if our software is used, and the average client users per installation. However, if you *really* don't want our remote Kawan servers to be notified by your server, just create the following file with any content:
`user.home/.kawansoft/no_notify.txt`, where `user.home` is the one of your Java EE web server. Notification will be deactivated and at server startup the message
`"[AWAKE START] Notification to Kawan Servers: OFF"`
will be inserted in the log defined by `CommonsConfigurator.getLogger()`
You can check the notification mechanism following source code in class `org.kawanfw.file.servlet.KawanNotifier.java.`

6.7 Managing temporary files

Awake uses temporary files on client side only. These temporary files contain:

- Result of the `RemoteFile.list` methods.
- Result of the `RemoteFile.listFiles` method.
- Chunks created by `RemoteInputStream` and `RemoteOutputStream.`

Temporary files are created to allows streaming and/or to release as soon as possible network resources (Servlet streams).

These temporary files are automatically cleaned (deleted) by Awake.

If you want to insure the cleaning of temporary files, they are located in the `user.home/.kawansoft/tmp` directory.
(Where `user.home` is the `System.getProperty("user.home")` value of the user that starts the client application and/or the servlet container on the server side.)

7 Troubleshooting

7.1 Session hang with Java 7+

It has been reported that Awake FILE session could in rare cases hang on client side with Java 7 and Java 8. If it happens, set this system property before creating an `RemoteSession` instance:

```
        System.setProperty("java.net.preferIPv4Stack", "true");
```

7.2 Defining a proxy with Java 7+

Awake FILE could hang on `RemoteSession` constructor call when using `ProxySelector.getDefault()` in intend to define a proxy with Java 7 or Java 8.

This could happen with early versions of Java 7 and Java 8.

```
        System.setProperty("java.net.useSystemProxies", "true");
        List<Proxy> proxies = ProxySelector.getDefault().select(
                new URI("http://www.google.com/"));

        String url = "http://www.acme.org:9090/ServerFileManager";
        String username = "login";
        char[] password = "password".toCharArray();

        Proxy proxy = new Proxy(Proxy.Type.HTTP, new InetSocketAddress(
                    "127.0.0.1", 8080));

        // Code could hang
        RemoteSession remoteSession = new RemoteSession(url, username,
                password, proxy, null);
```