File System Server 1.1 Component Specification

Version 1.1 changes are in **BLUE**, and new material is in **RED**.

1. Design

The File System Server and Client component extends the respective IP Server and Client components to provide a mechanism for sending, storing, managing, and retrieving files and groups of files from multiple clients to a server over a TCP/IP socket connection. The component shall never reject any file for any reason (including duplicate file names) other than hardware limitations. The component will handle any file-naming conflicts internally.

The FileSystemClient exposes methods for working with files and groups of files. All these methods return the requestId to the user. The user will use this requestId to receive the response message from the server using receiveResponse(requestId,blocking).

The file transfers (upload and retrieve) are performed behind the scenes, so the user doesn't need to know the actual protocol used.

When uploading a file, the user can get the upload file check status to see if the upload was accepted or not.

The user can plug error handlers that will be invoked if there is an error while uploading/retrieving a file.

Both the client and the server use a persistence object to be used for the file persistence. The persistence is pluggable, in order to support other persistences, except the file system.

The server maintains a registry for the files and groups. The registry associates the fileName sent by the client to a unique fileId, so there will not be name conflicts.

The server has a validator for the upload check messages that will validate the request before the upload is performed. The default implementation will check the file system to see if there is enough space.

Version 1.1 updates:

The client is augmented to support uploading and downloading data from and to a stream, instead of just working from some local file storage. The steps to accomplish this are very similar to the existing code, and closely parallel the existing upload and retrieve methods, as well as the existing workers. In fact, the new upload and retrieve methods, as well as the new direct workers do just about exactly the same work as these existing methods and workers, except that the workers read and write directly from streams instead of using a file persistence, hence their "Direct" prefix. The developer should find it straightforward to use the existing methods and workers as templates.

To accommodate the stream methods and the new workers, there are some other changes to the client, but nothing that breaks back-compatibility.

This version also supports file write locking on the server. This is accomplished by using the atomic operations of creating and deleting a directory that will be named after the file being written. The new SimpleLockingFileSystemPersistence class is introduced to decorate existing FilePersistence implementations by adding such locking capability.

A note on the SimpleLockingFileSystemPersistence. This class is to be used as a decorator for FilePersistence that work with a file system, such as the existing FileSystemPersistence class. As such, it is up to the customer to use compatible

FilePersistence instances. For example, it will not make sense to decorate a FilePersistence that stored data in a DB.

1.1 Design Patterns

Iterator for BytesIterator and InputStreamBytesIterator.

Facade for the upload/retrieve methods provided by the FileSystemClient. These methods are very easy to be used and they hide the underlying protocol used by the client and handler to split the files in segments of bytes, send the segments and reassembly them in a new file.

Strategy for the FileSystemRegistry and FileSystemXmlRegistry, as they are used by the FileSystemHandler.

Strategy for the FilePersistence and SystemFilePersistence, as they are used by the FileSystemHandler and FileSystemClient.

Strategy for the ObjectValidator and UploadRequestValidator, as they are used by the FileSystemHandler.

Strategy for the FreeDiskSpaceChecker and FreeDiskSpaceNativeChecker, as they are used by the UploadRequestValidator.

Strategy for the FileSearcher and RegexFileSearcher, as they are used by the SearchManager.

Strategy for the GroupSearcher and RegexGroupSearcher and FileIdGroupSearcher, as they are used by the SearchManager.

FileSystemHandler is part of the existing Strategy pattern in IPServer 2.0 (IPServer and Handler).

Concrete Messages are part of the existing Strategy pattern in IPServer 2.0 (IPServer,IPClient and Message).

SimpleLockingFileSystemPersistence is a Decorator/Wrapper for adding locking mechanisms to existing simple file-system-based FilePersistence implementations. SimpleLockingFileSystemPersistence is also another strategy implementation of the FilePersistence interface.

1.2 Industry Standards

XML and DOM- the FileSystemXmlRegistry uses xml files.

JNI - the FreeDiskSpaceNativeChecker uses a native method to check the free disk space.

1.3 Required Algorithms

1.3.1 Send request on the client side

```
1. get the request id
String requestId= createUniqueRequestId();

2.0. if the request is not of MessageType.UPLOAD_FILE or MessageType.RETRIVE_FILE
type:
        create a new RequestMessage of the required MessageType,
        and set the arguments in an arguments Object array
        Object[] args = new Object[n];
        args[0] = ...;
        RequestMessage request;
        request = new RequestMessage(handlerId, requestId, MessageType.XXX, args);

        // send the request message
        sendRequest(request);
```

```
2.1. if the request is of MessageType.UPLOAD_FILE type, create and start a
FileUploadWorker thread

2.2. if the request is of MessageType.RETRIEVE_FILE type, create and start a
FileRetrieveWorker thread

3. return the requestId
return requestId;

Note: for the user to receive the actual response from the server he will have to
call client.receiveResponse(requestId,blocking);
```

1.3.2 *Process request by the FileSystemHandler*

```
1. call super.onRequest(connection,request) for argument validation;

2. if the message is not of RequestMessage type, throw a ProcessingException with
a descriptive message

3. process each request according to the message type

3.1. if request.getType().equals(MessageType.UPLOAD_FILE)
// get the fileName and the fileId from the request.getArgs().
// the file has already been uploaded, so only the registry needs to be updated
registry.addFile(fileId,fileName) or renameFile(fileId, filename) method should
used.
// create a new response message of MessageType.UPLOAD_FILE, with the fileId as
the result.
// if there was some exception raised, return the exception in the response
message

3.2. if request.getType().equals(MessageType.CHECK_UPLOAD_FILE)
// check if the file upload request is accepted
valid=uploadRequestValidator.valid(request)
// create a new response message of MessageType.CHECK_UPLOAD_FILE, with a new
Boolean(valid) result.
// if there was some exception raised, return the exception in the response
message

3.3. if request.getType().equals(MessageType.REMOVE_FILE)
// get the fileId from the request.getArgs().
// remove the file from the registry and persistence (the block should be
synchronized on the registry object)
persistence.deleteFile(fileLocation,fileId);
registry.removeFile(fileId);
// create a new response message of MessageType.REMOVE_FILE, with a null result.
// if there was some exception raised, return the exception in the response
message
// this method will return an IllegalStateException if the file to be removed is
retrieved by other clients (check the filesToRetrieve map).

3.4. if request.getType().equals(MessageType.GET_FILE_NAME)
// get the fileId from the request.getArgs().
// get the fileName from the registry
fileName=registry.getFile(fileId);
// create a new response message of MessageType.GET_FILE_NAME, with the fileName
as the result.
// if there was some exception raised, return the exception in the response
message

3.5. if request.getType().equals(MessageType.GET_FILE_SIZE)
// get the fileId from the request.getArgs().
// get the file size from the persistence
fileSize=persistence.getFileSize(fileLocation,fileId);
// create a new response message of MessageType.GET_FILE_SIZE, with the file size
as the result.
// if there was some exception raised, return the exception in the response
message

3.6. if request.getType().equals(MessageType.RENAME_FILE)
// get the fileId and the new fileName from the request.getArgs().
// rename the file
registry.renameFile(fileName)
// create a new response message of MessageType.Rename_FILE, with a null result.
// if there was some exception raised, return the exception in the response
message

3.7. if request.getType().equals(MessageType.RETRIEVE_FILE)
// get the fileId from the request.getArgs().
// get the fileName from the registry
fileName=registry.getFile(fileId);
// get a bytes iterator from the persistence over the file
bytesIterator=persistence.getFileBytesIterator(fileId);
// get a unique bytes iterator id
id=createUniqueBytesIteratorId();
// put the bytes iterator in the map of bytes iterators and the fileId in the map
of files to retrieve
bytesIterators.put(id,bytesIterator);
filesToRetrieve.put(id,fileId);
// initialize the transfer session last access date
filesToRetrieveLastAccessDates.put(bytesIteratorId,new Date());
// create a new response message of BytesMessageType.START_RETIEVE_FILE_BYTES,
with an array with the fileName and the bytes iterator id as the result
// if there was some exception raised, return the exception in the response
message

3.8. if request.getType().equals(MessageType.CREATE_GROUP)
// get the groupName and the fileIds list from the request.getArgs().
// create the group in the registry
registry.createGroup(groupName,fileIds);
// create a new response message of MessageType.CREATE_GROUP, with a null result.
// if there was some exception raised, return the exception in the response
message

3.9. if request.getType().equals(MessageType.UPDATE_GROUP)
// get the groupName and the new fileIds list from the request.getArgs().
// create the group in the registry
registry.updateGroup(groupName,fileIds);
// create a new response message of MessageType.UPDATE_GROUP, with a null result.
// if there was some exception raised, return the exception in the response
message

3.10. if request.getType().equals(MessageType.RETRIEVE_GROUP)
// get the groupName from the request.getArgs().
// get the group’s list of file ids from the registry
fileIds=registry.getGroupFiles(groupName);
// create a new response message of MessageType.RETRIEVE_GROUP, with the fileIds
list as the result.
// if there was some exception raised, return the exception in the response
message

3.11. if request.getType().equals(MessageType.REMOVE_GROUP)
// get the groupName and the removeFiles flag from the request.getArgs().
// if the removeFiles is true, get the list of file ids of the group and remove
them
fileIds=registry.getGroupFiles(groupName);
for each fileId in fileIds do
...persistence.deleteFile(fileLocation,fileId);
...registry.removeFile(fileId);
// remove the group from the registry
registry.createGroup(groupName,fileIds);
// create a new response message of MessageType.REMOVE_GROUP, with a null result.
// if there was some exception raised, return the exception in the response
message
// this method will return an IllegalStateException if a file to be removed is
retrieved by other clients (check the filesToRetrieve map).


3.12. if request.getType().equals(MessageType.ADD_FILE_TO_GROUP)
// get the groupName and the fileId from the request.getArgs().
// add the file to the group in the registry
registry.addFileToGroup(groupName,fileId);
```

```
// create a new response message of MessageType.ADD_FILE_TO_GROUP, with a null
result.
// if there was some exception raised, return the exception in the response
message

3.13. if request.getType().equals(MessageType.REMOVE_FILE_TO_GROUP)
// get the groupName and the fileId from the request.getArgs().
// remove the file from the group in the registry
registry.removeFileToGroup(groupName,fileId);
// create a new response message of MessageType.REMOVE_FILE_TO_GROUP, with a null
result.
// if there was some exception raised, return the exception in the response
message

3.14. if request.getType().equals(MessageType.SEARCH_FILES)
// get the searcherName and the criteria from the request.getArgs().
// search files using the searchManager
files=searchManager.searchFiles(searcherName,criteria);
// create a new response message of MessageType.SEARCH_FILES, with the list of
files as the result.
// if there was some exception raised, return the exception in the response
message

3.15. if request.getType().equals(MessageType.SEARCH_GROUPS)
// get the searcherName and the criteria from the request.getArgs().
// search groups using the searchManager
files=searchManager.getGroups(searcherName,criteria);
// create a new response message of MessageType.SEARCH_GROUPS, with the list of
groups as the result.
// if there was some exception raised, return the exception in the response
message

3.16. if request.getType().equals(BytesMessageType.START_UPLOAD_FILE_BYTES)
// get the fileName and the fileId, in case the file should be over written, from
the request.getArgs().
// create a new fileId in case it is not provided
fileId=registry.getNextFileId();
// create the file in the persistence and get the creation id
fileCreationId=persistence.createFile(fileLocation,fileId);
// create a new response message of BytesMessageType.START_UPLOAD_FILE_BYTES, with
an array containing the fileId and the fileCreationId result.
filesToUpload.put(fileCreationId,fileId);
// initialize the transfer session last access date
filesToUploadLastAccessDates.put(fileCreationId,new Date());
// if there was some exception raised, return the exception in the response
message

3.17. if request.getType().equals(BytesMessageType.UPLOAD_FILE_BYTES)
// get the fileCreationId and the array of bytes from the request.getArgs().
// append the bytes to the file
persistence.appendBytes(fileCreationId,bytes);
// create a new response message of BytesMessageType.UPLOAD_FILE_BYTES, with a
null result.
// update the transfer session last access date
filesToUploadLastAccessDates.put(fileCreationId,new Date());
// if there was some exception raised, return the exception in the response
message

3.18. if request.getType().equals(BytesMessageType.STOP_UPLOAD_FILE_BYTES)
// get the fileCreationId from the request.getArgs().
// close the file in the persistence
persistence.closeFile(fileCreationId);
// create a new response message of BytesMessageType.STOP_UPLOAD_FILE_BYTES, with
a null result.
// remove the transfer session last access date
filesToUpload.remove(fileCreationId);
filesToUploadLastAccessDates.remove(fileCreationId);
// if there was some exception raised, return the exception in the response
message

3.19. if request.getType().equals(BytesMessageType.START_RETRIEVE_FILE_BYTES)
```

```
// get the bytesIteratorId and the fileCreationId from the request.getArgs().
// get the next bytes from the iterator
bytesIterator=bytesIterators.get(bytesIteratorId);
if bytesIterator.hasNextBytes()
...bytes=bytesIterator.nextBytes
...// create a new response message of BytesMessageType.RETRIEVE_FILE_BYTES, with
the fileCreationId and the array of bytes as the result.
else
... // create a new response message of BytesMessageType.STOP_RETRIEVE_FILE_BYTES,
with the fileCreationId as the result.
// update the transfer session last access date
filesToRetrieveLastAccessDates.put(bytesIteratorId,new Date());
// if there was some exception raised, return the exception in the response
message

3.20. if request.getType().equals(BytesMessageType.RETRIEVE_FILE_BYTES)
// get the bytesIteratorId and the fileCreationId from the request.getArgs().
// get the next bytes from the iterator
bytesIterator=bytesIterators.get(bytesIteratorId);
if bytesIterator.hasNextBytes()
...bytes=bytesIterator.nextBytes
...// create a new response message of BytesMessageType.RETRIEVE_FILE_BYTES, with
the fileCreationId and the array of bytes as the result.
else
... // create a new response message of BytesMessageType.STOP_RETRIEVE_FILE_BYTES,
with the fileCreationId as the result.
// update the transfer session last access date
filesToRetrieveLastAccessDates.put(bytesIteratorId,new Date());
// if there was some exception raised, return the exception in the response
message

3.21. if request.getType().equals(BytesMessageType.STOP_RETRIEVE_FILE_BYTES)
// get the bytesIteratorId from the request.getArgs().
// remove the iterator from the map of iterators and the fileId from the files to
be retrieved (and get the fileId);
bytesIterators.remove(bytesIteratorId);
fileId= filesToRetrieve.get(bytesIteratorId);
filesToRetrieve.remove(bytesIteratorId);
// remove the transfer session last access date
filesToRetrieveLastAccessDates.remove(bytesIteratorId);
// create a new response message of MessageType.RETRIEVE_FILE, with the fileId as
the result.
// if there was some exception raised, return the exception in the response
message


4. send the response to the client
connection.getIPServer().sendResponse(connection.getId(),response);
```

1.3.3 Upload file request validation

This check is performed when a CHECK_UPLOAD request message is received. The request is processed by the UploadRequestValidator (an ObjectValidator), which delegates to the FreeDiskSpaceChecker.
The FreeDiskSpaceNativeChecker implements FreeDiskSpaceChecker and checks if there is enough space on the disk to upload the file.

FreeDiskSpaceNativeChecker usage:
It uses JNI to perform its job.
The freeDiskSpaceExceedsSize(long) delegates its work to a private native method - freeDiskSpaceExceedsSize(..).
The name of the dinamic library used by this class is "FSSDiskSpaceChecker" (FSS - File System Server).
Impl. notes for the native code for
"freeDiskSpaceExceedsSize(fileLocation:String,fileSize:long) : boolean":
Windows:

- using the function GetFreeDiskSpace, or GetFreeDiskSpaceEx, or GetFreeDiskSpaceExA. These functions require to extract the drive from the fileLocation

Linux:

- using the functions ls, or df, or dp.

Solaris:

- using the functions di or df.
FreeDiskSpaceNonNativeChecker implementation:

1.3.4 Upload file protocol

Client code:

```
String requestId = fsClient.uploadFile([fileId],fileLocation,fileName);
FileUploadCheckStatus status =
fsClient.getFileUploadCheckStatus(requestId, blocking);
if (status==FileUploadCheckStatus.UPLOAD_ACCEPTED) {
       while(fsClient.isFileTransferWorkerAlive(requestId)) {
            Thread.sleep(500);
       }
       ResponseMessage response =
       (ResponseMessage) fsClient.receiveResponse(requestId,blocking);
       // receiving may be timeout.
       if (response != null && response.getException()==null) {
              String fileId = (String) response.getResult();
       } else {
              // handle the exception
       }
}
```

FileSystemClient uploadFile(..) method flow:

```
1. get the request id
String requestId= createUniqueRequestId();
2. create and start a FileUploadWorker thread
3. put an entry in fileTransferWorker map. Using requested as key, thread as value.
4. return the requestId
```

FileUploadWorker run() method flow:

```
Note: each new message will have a new requestId, except for the last UPLOAD_FILE
message sent.
1. send a MessageType.CHECK_UPLOAD request message to see if the upload is
accepted (the server will reply with a MessageType.CHECK_UPLOAD_FILE response
message).
       - if the upload is accepted, it will add a (requestId,
       FileUploadCheckStatus.UPLOAD_ACCEPTED) pair to the checkUploadStatuses map.
       - if the upload is not accepted, it will add a (requestId,
       FileUploadCheckStatus.UPLOAD_NOT_ACCEPTED) pair to the checkUploadStatuses
       map and returns.
       (Note: the user can query this status using
       fsClient.getFileUploadCheckStatus(..) method).
2. send a BytesMessageType.START_UPLOAD_FILE_BYTES message to get the
fileCreationId from the server (the server will reply with a
MessageType.START_UPLOAD_FILE_BYTES response message).
3. get the bytesIterator over the file denoted by the fileLocation and FileName
fields from the persistence
4. while the bytesIterator has next bytes, get the bytes and send them in a
BytesMessageType.UPLOAD_FILE_BYTES message to the server (the server will reply
with a MessageType. UPLOAD_FILE_BYTES response message).
5. when there are no more bytes, send a BytesMessageType.STOP_UPLOAD_FILE_BYTES
message (the server will reply with a MessageType.STOP_UPLOAD_FILE_BYTES response
message).
6. send a MessageType.UPLOAD_FILE message to the server  with the finalRequestId
7. remove the entry with key requested.
8. return // the response will be received by the user.
```

1.3.5 Retrieve file protocol

Client code:

```
String requestId = fsClient.retrieveFile(fileId,fileLocation);
While(fsClient.isFileTransferWorkerAlive(requestId)) {
    Thread.sleep(500);
}
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,blocking);
if (response != null && response.getException()==null) {
       String fileName = (String) response.getResult();
} else {
       // handle the exception
}
```

```
FileSystemClient retrieveFile(..) method flow:
1. get the request id
String requestId= createUniqueRequestId();
2. create and start a FileRetrieveWorker thread
3. put an entry in fileTransferWorker map.
4. return the requestId

FileRetreiveWorker run() method flow:
Note: each new message will have a new requestId, except for the last
STOP_RETRIEVE_FILE_BYTES message sent.
1. send a MessageType.RETRIEVE_FILE request message to get the fileName and the
bytesIteratorId from the server (the server will reply with a
MessageType.START_RETRIEVE_FILE response message).
2. receive the BytesMessageType.START_RETRIEVE_FILE_BYTES response message
3. get the fileName and bytesIteratorId from the response message
4. create a new file in the persistence and get the creationFileId
4. send a BytesMessageType.START_RETRIEVE_FILE_BYTES mesage, with the
bytesIteratorid received in the previous message (the server could respond with a
RETRIEVE_FILE_BYTES or a STOP_RETRIEVE_FILE_BYTES response message).
5. while it receives BytesMessageType.RETRIEVE_FILE_BYTES response messages, get
the bytes and append them in the persistence and send a new
BytesMessageType.RETRIEVE_FILE_BYTES message to the server
6. when a BytesMessageType.STOP_RETRIEVE_FILE_BYTES message is received, close the
file in the persistence and send a BytesMessageType.STOP_RETRIEVE_FILE_BYTES
message with the finalRequestId
7. Last remove the entry in fileTransferWorker map with the finalRequestId and
return // the server will respond with a MessageType.RETRIEVE_FILE response
message, which should be received by the user.
```

1.3.6 *FileSystemXmlRegistry - the format of the xml files used*

```
1. the format of the xml file for the files (fileId-fileName mappings)
--- the dtd schema ---
<!ELEMENT files (file*)>
<!ELEMENT file EMPTY>
<!ATTLIST file
	id CDATA #REQUIRED
	name CDATA #REQUIRED
>
---
File example:
<files>
	<file id=”1001” name=”file1.txt” />
	<file id=”1002” name=”file2.txt” />
	<file id=”1003” name=”file3.txt” />
	<file id=”1004” name=”file4.jpg” />
</files>

2. the format of the xml file for the groups
--- the dtd schema ---
<!ELEMENT groups (group*)>
<!ELEMENT group (file*)>
<!ELEMENT file EMPTY>
<!ATTLIST group
	name CDATA #REQUIRED
>
<!ATTLIST file
	id CDATA #REQUIRED
>
---
File example:
<groups>
	<group name=”group1” />
		<file id=”1002” />
		<file id=”1003” />
		<file id=”1004” />
	</group>
	<group name=”group2” />
	</group>
	<group name=”group3” />
		<file id=”1001” />
```

```
               <file id=”1003” />
        </group>
</groups>
```

1.3.7 *FileSystemXmlRegistry’s constructor - create xml documents using the files*

```
1. create document builder factory for making Document
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

2. create document builder
DocumentBuilder builder = factory.newDocumentBuilder();

3. if the files’ file exists, then parse it; if not, create a new Document
if (filesFile.exists()) {
       filesDocument = builder.parse(filesFile);
       // parse out the exist elements.
       NodeList nodeList = filesDocument.getElementsByTagName(“file”);
       For each Element in the node List:
           Get the id and name attributes, and add to registry.
} else {
       filesDocument = builder.newDocument();
       filesDocument.appendChild(
              filesDocument.createElement(“files”));
}

4. if the groups’ file exists, then parse it; if not, create a new Document
if (groupsFile.exists()) {
       groupsDocument = builder.parse(groupsFile);
       // parse out the exist elements.
       NodeList nodeList = filesDocument.getElementsByTagName(“group”);
       For each Element in the node List:
           Get the id and name attributes, and add to registry.

} else {
       groupsDocument = builder.newDocument();
       groupsDocument.appendChild(
              groupsDocument.createElement(“groups”));
}
```

1.3.8 *FileSystemXmlRegistry’s commitDocument - save the xml document in the file*

```
1. get instance of transformer factory
TransformerFactory factory = TransformerFactory.newInstance();

2. create transformer to transform document object to file
  Transformer transformer = factory.newTransformer();

3. create the file if it doesn't exist
if (!file.exists()) {
       file.createNewFile();
}

4. transform the document to the output file
Result result = new StreamResult(file);
Source source = new DOMSource(doc);
transformer.transform(source, result);
```

1.3.9 *Upload file from stream protocol*

```
Client code:
String requestId = fsClient.uploadFile([fileId],fileStream, [filename]);
FileUploadCheckStatus status =
fsClient.getFileUploadCheckStatus(requestId, blocking);
if (status==FileUploadCheckStatus.UPLOAD_ACCEPTED) {
       while(fsClient.isFileTransferWorkerAlive(requestId)) {
            Thread.sleep(500);
       }
       ResponseMessage response =
       (ResponseMessage) fsClient.receiveResponse(requestId,blocking);
       // receiving may be timeout.
       if (response != null && response.getException()==null) {
              String fileId = (String) response.getResult();
```

```
        } else {
                // handle the exception
        }
}
```

FileSystemClient InputStream-using uploadFile(…) method flow:

```
1. get the request id
String requestId= createUniqueRequestId();
2. create and start a DirectFileUploadWorker thread
3. put an entry in fileTransferWorker map. Using requested as key, thread as value.
4. return the requestId
```

DirectFileUploadWorker run() method flow:

```
Note: each new message will have a new requestId, except for the last UPLOAD_FILE
message sent.
1. send a MessageType.CHECK_UPLOAD request message to see if the upload is
accepted (the server will reply with a MessageType.CHECK_UPLOAD_FILE response
message).
        - if the upload is accepted, it will add a (requestId,
        FileUploadCheckStatus.UPLOAD_ACCEPTED) pair to the checkUploadStatuses map.
        - if the upload is not accepted, it will add a (requestId,
        FileUploadCheckStatus.UPLOAD_NOT_ACCEPTED) pair to the checkUploadStatuses
        map and returns.
        (Note: the user can query this status using
        fsClient.getFileUploadCheckStatus(..) method).
2. send a BytesMessageType.START_UPLOAD_FILE_BYTES message to get the
fileCreationId from the server (the server will reply with a
MessageType.START_UPLOAD_FILE_BYTES response message).
2.1. If the process did not get a fileName from the user, then make the response’s
fileId as the filename.
3. while the dataStream has more bytes, get the bytes of size transferByteSize and
send them in a BytesMessageType.UPLOAD_FILE_BYTES message to the server (the
server will reply with a MessageType. UPLOAD_FILE_BYTES response message).
4. when there are no more bytes, send a BytesMessageType.STOP_UPLOAD_FILE_BYTES
message (the server will reply with a MessageType.STOP_UPLOAD_FILE_BYTES response
message).
5. send a MessageType.UPLOAD_FILE message to the server  with the finalRequestId
6. remove the entry with key requested.
7. return // the response will be received by the user.
```

1.3.10 Retrieve file into stream protocol

Client code:

```
String requestId = fsClient.retrieveFile(fileId,fileLocation);
While(fsClient.isFileTransferWorkerAlive(requestId)) {
    Thread.sleep(500);
}
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,blocking);
if (response != null && response.getException()==null) {
        String fileName = (String) response.getResult();
} else {
        // handle the exception
}
```

FileSystemClient OutputStream-using retrieveFile(..) method flow:

```
1. get the request id
String requestId= createUniqueRequestId();
2. create and start a DirectFileRetrieveWorker thread
3. put an entry in fileTransferWorker map.
4. return the requestId
```

FileRetreiveWorker run() method flow:

```
Note: each new message will have a new requestId, except for the last
STOP_RETRIEVE_FILE_BYTES message sent.
1. send a MessageType.RETRIEVE_FILE request message to get the fileName and the
bytesIteratorId from the server (the server will reply with a
MessageType.START_RETRIEVE_FILE response message). The filename will be ignored.
2. receive the BytesMessageType.START_RETRIEVE_FILE_BYTES response message
3. get the bytesIteratorId from the response message
```

```
4. send a BytesMessageType.START_RETRIEVE_FILE_BYTES mesage, with the
bytesIteratorid received in the previous message (the server could respond with a
RETRIEVE_FILE_BYTES or a STOP_RETRIEVE_FILE_BYTES response message).
5. while it receives BytesMessageType.RETRIEVE_FILE_BYTES response messages, get
the bytes and write them into the dataStream and send a new
BytesMessageType.RETRIEVE_FILE_BYTES message to the server
6. when a BytesMessageType.STOP_RETRIEVE_FILE_BYTES message is received, close the
file in the persistence and send a BytesMessageType.STOP_RETRIEVE_FILE_BYTES
message with the finalRequestId
7. Last remove the entry in fileTransferWorker map with the finalRequestId and
return // the server will respond with a MessageType.RETRIEVE_FILE response
message, which should be received by the user.
```

1.4 Component Class Overview

[com.topcoder.file.transfer]

FileSystemHandler

This is the file system handler required; it extends the Handler class from IP Server 2.0. It provides methods for processing requests and sending responses back to the client. It uses a file registry to maintain files and groups, a file persistence to persist files, it has the file location configured, it uses an ObjectValidator to validate check upload requests, a search manager to search for files or groups and a GUID generator to generate unique ids. The class is immutable.

TimeOutTransferSessionsTask

This task is used by the FileSystemHandler to clear up expired sessions. It should clear both upload and retrieve sessions using the filesToRetrieveLastAccessDates and filesToUploadLastAccessDates maps to detect expired sessions.

FileSystemClient

The FileSystemClient extends the IPClient class from IP Server 2.0. The return type of its methods is of String type - the requestId to be used by the user to receive the response in blocking or non-blocking mode using receiveResponse(requestId,blocking) method (the returned Message should be of ResponseMessage type). The user is advised not to use IP client's receiveResponse(blocking) method, as it might lead to data corruption. The user should use only the receiveResponse(requestId,blocking) method. With version 1.1., the class can now also upload data directly from a stream instead of just the file persistence, and it can equally retrieve a file from the server directly into a supplied stream.

FileUploadWorker

This thread is used by the client to upload a file on the server, in order to perform the job in background and not expose the protocol to the user.

FileRetrieveWorker

This thread is used by the client to retrieve a file from the server, in order to perform the job in background and not expose the protocol to the user.

DirectFileUploadWorker

This thread is used by the client to upload a file on the server directly from a stream, in order to perform the job in background and not expose the protocol to the user.

If no fileName is provided, then it will make the fileId as the fileName.

DirectFileRetrieveWorker

This thread is used by the client to retrieve a file from the server and put it directly into a stream, in order to perform the job in background and not expose the protocol to the user.

FileTransferHandler

This interface should be implemented by the user to handle the errors that might occur during a file transfer or transfer progress if upload or retrieve bytes successfully. Instances are used by the upload worker and retrieve worker threads to signal errors or transfer progress.

FileUploadCheckStatus

This class represents the upload check status of an upload file request to the server. The FileSystemClient delegated the upload of the file to a FileUploadWorker. The FIleUploadWorker sends a CHECK_UPLOAD message. If the upload is approved by the server, it will add a (requestId,FileUploadCheckStatus.UPLOAD_ACCEPTED) pair to the statuses map. If the upload is not approved by the server, it will add a (requestId,FileUploadCheckStatus.UPLOAD_NOT_ACCEPTED) pair to the statuses map. The user can query this status by using the getFileUploadCheckStatus(requestId,blocking) method of the client.

[com.topcoder.file.transfer.message]

FileSystemMessage

This is a base class for the ResponseMessage and RequestMessage. It provides the request type field. It extends SimpleSerializableMessage, in order for the subclasses to be serialized using SerializableMessageSerializer. The class is immutable.

RequestMessage

This class represents the request message sent by the FileSystemClient in order to perform an operation on the server side. It has an array of objects initialized in the constructor. The constructor that receives the args argument validates the array according to the message type provided. The class is immutable.

ResponseMessage

This class represents the response message sent by the IPServer, using FileSystemHandler as a response to an operation on the server side. It has an object result initialized in the constructor, or an exception, in case the operation was errored. The constructor that receives the result argument validates the object according to the message type provided. The class is immutable.

MessageType

This class declares the types of messages currently supported by this component, except the types that involve bytes transfer. Each instance of this class has a message type name associated with it (the type:String serves to determine the hashCode and whether two instances are equal). The class is immutable. The public constants declare the rules of validation that should be performed inside the RequestMessage and ResponseMessage's constructors.

BytesMessageType

This class declares the types of messages currently supported by this component that involve bytes transfer. Each instance of this class has a message type name associated with it (the type:String serves to determine the hashCode and whether two instances are equal). The class is immutable. The public constants declare the rules of validation that should be performed inside the RequestMessage and ResponseMessage's constructors.

MessageTypeValidator

This is a package-friendly class that used to validate the RequestMessage or ResponseMessage depending on the type. The required validation of every message type is in the comment of the type in MessageType and BytesMessageType.

[com.topcoder.file.transfer.registry]

FileSystemRegistry

Defines the contract that each file system should respect. This interface is used by the FileSystemHandler to maintain a mapping between the file ids and file names, and to maintain a list of groups and their associated files. The registry should have a persistent storage, so it can function after an application has restarted. The implementations should be thread safe. More, all the methods should be synchronized - this is required because some operations require several calls to this instance and, in order to achieve transactional thread safety, the application should perform the calls inside a synchronized block using this instance's lock.

FileSystemXmlRegistry

Represents an implementation of the FileSystemRegistry interface. It maintains a mapping between the file ids and file names, and it maintains a list of groups and their associated files. The registry works with two XML files as a persistent storage: one file is for the fileId-fileName mappings and one file is for the list of groups. It uses two files because at the end of each method which alters a document field, the XML file is rewritten to persist the changes, and saving only the files document or only the groups document takes less time. It is thread safe for atomic method calls. All the methods are synchronized - this is required because some operations require several calls to this instance and, in order to achieve transactional thread safety, the application should perform the calls inside a synchronized block using this instance's lock. See Component Specification, Required Algorithms section for more details about the structure of the xml files used by this class.

[com.topcoder.file.transfer.persistence]

FilePersistence

This interface defines the contract that each file persistence should respect. It declares methods to create a file (createFile, appendBytes and closeFile), a method to delete a file and a method to retrieve a BytesIterator for a given file. The creation of the file is performed in 3 steps:

1. createFile() - the file is created and the user gets a file id to use in the next steps.

2. appendBytes() - this method should be used several times by the user to write all the bytes.

3. closeFile() - the file is closed

The sub-classes do not need to be fully thread safe as the creation of the file should be performed by only one thread, or the file could be corrupted (and no other user should be aware of the file's id in order to try to access the file). The deleteFile and getFileBytesIterator could be accessed by multiple threads, so the methods should be synchronized.

FileSystemPersistence

This is the default implementation of the FilePersistence interface that works with files from the file system. The creation of the file is performed in 3 steps:

1. createFile() - the file is created and the user gets a file id to use in the next steps.

2. appendBytes() - this method should be used several times by the user to write all the bytes.

3. closeFile() - the file is closed

It is not fully thread safe as the creation of the file should be performed by only one thread, or the file could be corrupted (and no other user should be aware of the file's id in order to try to access the file). The deleteFile and getFileBytesIterator could be accessed by multiple threads, so the methods are synchronized.

SimpleLockingFileSystemPersistence

This is a simple implementation of the FilePersistence interface that decorates other file-system-based FilePersistence implementations with a file-locking mechanism.Specifically, it provides a write block for all writing operations. Reading is not a locking operation to allow multiple readings at the same time.

The locking is accomplished by directory creation, as this is an atomic process. A directory will be created in the fileLocation with the name persistenceFileName+"_lock". Once the lock is ready to be released, it will be deleted.

BytesIterator

This interface provides the contract that each bytes iterator should respect. Instances of this interface can be obtained from the FilePersistences. This interface is needed because large files cannot be sent over the network in just one message using the IPServer. And the files need to be split and sent in smaller consecutive messages. The subclasses need not to be thread safe, as instances of this interface will be used just by one thread. More threads using this interface will lead to file corruption after the file is reconstructed.

InputStreamBytesIterator

This is the default implementation of the BytesIterator interface. It reads the bytes from an InputStream. The byte arrays returned have the same pre-configured length, but the last array could be smaller (as end of stream could be reached). It is not thread safe as just one thread should access one instance.

[com.topcoder.file.transfer.search]

SearchManager

This class performs file and group searches using FileSearcher and GroupSearcher instances. Its purpose is to provide a centralized access point for the FileSystemHandler. It is not tread-safe, as this class should not be changed after it is initialized. Only the two search methods (getFiles and getGroups) should be used after initialization by the FileSystemHandler. Thus, the implementations of the FileSearcher and GroupSearcher interfaces should be thread-safe.

FileSearcher

Defines the contract that each file searcher should respect. This interface is used by the SearchManager. The implementations of this interface should be thread-safe.

RegexFileSearcher

Implements the contract defined by the FileSearcher interface.This class returns a list of file ids whose corresponding file names match the given criteria.The criteria is interpreted as a java regex pattern (this class uses java.util.regex package).It is thread safe. However, it uses a FileSystemRegistry instance which should be thread-safe also.

GroupSearcher

Defines the contract that each group searcher should respect. This interface is used by the SearchManager. The implementations of this interface should be thread-safe.

RegexGroupSearcher

Implements the contract defined by the GroupSearcher interface. This class returns a list of group names which match the given criteria. The criteria is interpreted as a java regex pattern (this class uses java.util.regex package). It is thread safe. However, it uses a FileSystemRegistry instance which should be thread-safe also.

FileIdGroupSearcher

Implements the contract defined by the GroupSearcher interface. This class returns a list of group names which contain a file id equal with the given criteria. The criteria is interpreted as a file id string. It is thread safe as the access to the FileSystemRegistry instance is done inside a synchronized block.

[com.topcoder.file.transfer.validator]

UploadRequestValidator

This class represents the default upload request validator used by the file system handler. It validates objects of RequestMessage type. The request message must have the type equal with RequestType.CHECK_UPLOAD_FILE. It uses a FreeDiskSpaceChecker to check if the upload request is accepted. It will get the file size from request message's args and pass it to the FreeDiskSpaceChecker. This class is thread safe. However, it uses the free disk space checker, so this one should be thread safe.

FreeDiskSpaceChecker

This interface declares the contract for the concrete free disk space checkers. This interface is used by the UploadRequestValidator. Implementations should be thread safe.

FreeDiskSpaceNativeChecker

This is the default implementation of the FreeDiskSpaceChecker. It uses JNI to perform its job. The freeDiskSpaceExceedsSize(long) delegates its work to a private native method. The class is thread safe, as freeDiskSpaceExceedsSize(long) is synchronized, to avoid conflicts in the native code. The name of the dinamic library used by this class is "FSSDiskSpaceChecker" (FSS - File System Server).

Impl. notes for the native code for "-freeDiskSpaceExceedsSize(fileLocation:String, fileSize:long) : boolean":

Windows: using the function GetFreeDiskSpace, or GetFreeDiskSpaceEx, or GetFreeDiskSpaceExA; these functions require to extract the drive from the fileLocation

Linux: using the functions ls, or df, or dp.

Solaris: using the functions di or df.

FreeDiskSpaceNonNativeChecker

This is the implementation of the *FreeDiskSpaceChecker*, it doesn’t use JNI to perform the job like *FreeDiskSpaceNativeChecker* do. The free space is calculated via the command line. It uses 'dir /-c' on Windows and 'df' on *nix.

FreeDiskSpaceUtils

This is the utility class used by **FreeDiskSpaceNonNativeChecker** to calculate the free disk space via command line. It uses ‘dir /-c’ on Windows and ‘df’ on *nix.

1.5 Component Exception Definitions

[com.topcoder.file.transfer.registry]

RegistryException

This is the base exception for the exceptions in the registry package. It is also thrown by the FileSystemRegistry's methods if an exception occurred while performing the operations.

RegistryConfigurationException

This exception is thrown by the FileSystemXmlRegistry's constructors if an exception occurs while initializing the object.

RegistryPersistenceException

This exception is thrown by the FileSystemRegistry if an exception occurs while saving changes in the persistent storage.

FileIdException

This is the base exception for the exceptions related to a file id.

FileIdExistsException

This exception is thrown by the FileSystemRegistry if a given file id already exists.

FileIdNotFoundException

This exception is thrown by the FileSystemRegistry if a given file id cannot be found.

GroupException

This is the base exception for the exceptions related to a group name.

GroupExistsException

This exception is thrown by the FileSystemRegistry if a given group name already exists.

GroupNotFoundException

This exception is thrown by the FileSystemRegistry if a given group name cannot be found.

[com.topcoder.file.transfer.persistence]

FilePersistenceException

Thrown by the FilePersistence and BytesIterator's methods if an exception occurs while performing some operation.

FileAlreadyLockedException

Thrown by the FilePersistence methods if the write operation cannot take place because the file to be modified is already locked by another process. Extends FilePersistenceException.

FileNotYetLockedException

Thrown by the FilePersistence methods if the write operation cannot take place because the file to be modified has not yet had a lock obtained on it. Extends FilePersistenceException.

[com.topcoder.file.transfer.search]

SearcherException

This is the base class for all the exceptions in the search package.

FileSearcherNotFoundException

This exception is thrown by the SearchManager if a given file searcher name cannot be found when performing a search.

GroupSearcherNotFoundException

This exception is thrown by the SearchManager if a given group searcher name cannot be found when performing a search.

SearchException

This exception is thrown if an error occurs while performing a search.

[com.topcoder.file.transfer.validator]

FreeDiskSpaceCheckerException

This exception is thrown by the FreeDiskSpaceChecker if an error occurs while performing the check.

Other exceptions.

ConfigurationException [Custom]

This exception is thrown by FileSystemClient's constructor if there is an error while loading the configuration properties.

ProcessingException [Custom]

This exception is thrown by the cache handler's onRequest method if a fatal error occurs.

IOException

This exception is thrown by FileSystemClient's methods if a socket error occurs during sending data.

IllegalStateException

This exception is thrown by FileSystemClient's methods if the client is not connected to the server.

NullPointerException

This exception is thrown in various methods where null value is not acceptable.
Refer to the documentation in Poseidon for more details.

IllegalArgumentException

This exception is thrown in various methods if the given string argument is empty or some other cases. Refer to the documentation in Poseidon for more details.

NOTE: Empty string means string of zero length or string full of white spaces.

1.6 Thread Safety

This component can be used in a thread-safe manner by multiple threads.

The FileSystemHandler is thread safe, of course, as required by the base class Handler (IPServer will call it's methods using multiple threads). It is immutable and the classes used are thread safe. In case a transaction is required, the handler synchronizes on the registry field, as this instance is required to have all its methods synchronized.

The FileSystemClient is thread safe, as it is immutable. It could be used in a multi-threaded environment. These classes use the base IPClient classes methods to do their job. And IPClient is thread safe.

The Message concrete implementations (RequestMessage and ResponseMessage) are immutable, thus they are thread safe. However, instances of these classes should not be shared between threads. There is no need for it.

MessageType and BytesMessageType are immutable, so they are thread safe.

Transfer transaction strategy:

"Upload' file is safe, as its fileId will not be added to the registry until the transfer is over, so nobody will interfere.

'Retrieve' file has the smallest priority among other operations. It will always fail if the file involved in transaction is renamed, removed or over-written.

‘Upload-and-over-write’ and ‘remove’ file are performed no matter if other clients are retrieving the file (their bytesIterators are disposed and the clients would get some exception there).

'Remove' is done in one shot, but the 'upload-and-over-write' requires more messages. So, ‘remove’ will always succeed, and future message for 'upload-and-over-write' will fail using some exception.

‘Rename’ will behave just like ‘remove’. Existing bytesIterators are disposed and the clients would get some exception (because the client has created the file on his persistence with one name, and will get from the final message a different name - so, it’s better to stop the process).

Version 1.1 does not alter the thread-safety of this component. The new methods of the FileSystemClient are thread-safe. The lack of thread-safety of the SimpleLockingFileSystemPersistence may only come from a FilePersistence implementation that it wraps.

The file-locking in SimpleLockingFileSystemPersistence is thread-safe and atomic. However, the locking consistency is only reserved in the file system if only SimpleLockingFileSystemPersistences are managing the locking. Nothing stops other processes from deleting/creating the locks. It is up to the system administrator to ensure this does not happen.

2. Environment Requirements

2.1 Environment

Development: Java 1.4

Compile: Java 1.4

2.2 TopCoder Software Components

IPServer 2.0

Provides the request/processor framework used by this component to perform server-side caching.

Configuration Manager 2.1.5

It is used in FileSystemClient to load configuration values from configuration file.

GUID Generator 1.0

Is used in the CacheClient class to generate unique request ids. This component has the advantage of not requiring persistent storage (such as ID Generator requires), making the component easier to use. A generator is obtained with UUIDUtility.getGenerator(UUIDType.TYPEINT32). Then using generator.getNextUUID().toString() ids are generated as needed.

ID Generator 3.0

It is used by the FileSystemXmlRegistry to generate unique ids for the files.

Data Validation 1.0

Used to validate the check upload request message by the server.

BaseException 2.0

It is used as base class for the exceptions.

Type Safe Enum 1.1.0

Used for FileUploadCheckStatus.

It is not used for MessageType because other message types might be desired to be supported.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.file.transfer

com.topcoder.file.transfer.message

com.topcoder.file.transfer.registry

com.topcoder.file.transfer.persistence

com.topcoder.file.transfer.validator

com.topcoder.file.transfer.search

3.2 Configuration Parameters

The IPServer component needs to be properly configured.

3.3 Dependencies Configuration

None.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Configure your database connection in test_files/DBConnectionFactoryImpl.xml
- Run test_files/database/id_generator.sql against your configured database.
- Place test_files/accuracy/FSSDiskSpaceChecker.dll on your PATH (if not testing in Windows, your will need to compile the library from the provided sources).
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

A FileSystemHandler should be created and plugged programmatically.
The IPServer could have a FileSystemHandler mapped under the "HandlerId" handler id.

4.3 Demo

The change in the version 1.1. demo is simply to use the new SimpleLockingFileSystemPersistence for file persistence in the server. When being demoed, the user can confirm the existence of the lock directories when writing files.

4.3.1 Register a FileSystemHandler to the IPServer programatically

```
// create or get the IPServer (read the IP Server 2.0 docs)
IPServer ipServer = ...;

// create a file registry
IDGenerator idGenerator = IDGeneratorFactory.getIDGenerator(namespace);
FileSystemRegistry registry = new FileSystemXmlRegistry(filesFileLocation,
groupsFileLocation, idGenerator)
```

```
// create the file persistence
FilePersistence persistence = new SimpleLockingFileSystemPersistence(new
FileSystemPersistence());

// create the upload request validator
// the checker is non-JNI implementation.
ObjectValidator validator = new UploadRequestValidator(new
FreeDiskSpaceNonNativeChecker(SERVERFILELOCATION));
// create the search manager
SearchManager searchManager = new SearchManager()
SearchManager.addFileSearcher(“regex”, new RegexFileSearcher(registry));
SearchManager.addGroupSearcher(“regex”, new RegexFileSearcher(registry));
SearchManager.addGroupSearcher(“fileId”, new
FileIdFileSearcher(registry));

// create the handler
Handler fsHandler = new
FileSystemHandler(maxRequests,registry,fileLocation,validator,searchManag
er);

// map the handler in the IPServer under a String id
ipServer.addhandler(“HandlerId”,handler);
```

4.3.2 Create a FileSystemClient

```
// get the handlerId
String handlerId = “HandlerId”; // or get it from somewhere

// create the file persistence
FilePersistence persistence = new FileSystemPersistence();

// create the error handlers
FileTransferErrorHandler uploadErrorHandler = ...; //some error handler
FileTransferErrorHandler retrieveErrorHandler = ...; //some error handler

// create the client
FileSystemClient fsClient = new FileSystemClient("192.168.0.1",
139, ”com.topcoder.util.filesystemclient”, handlerId,persistence,
uploadErrorHandler, retrieveErrorHandler);

// connect and disconnect the client, as specified in IP Server 2.0 docs
// fsClient.connect();
// fsClient.disconnect();
```

4.3.3 Perform blocking / non-blocking requests

```
Every operation is performed in the same manner. The user calls the
method he wants to use and receives a requestId:String (this method
returns immediately). Then, the user will use the requestId to get the
response message from the server. And this last operation can be
performed in blocking or non-blocking mode.

// we’ll use getFileName as an example

// call getFileName method
String fileId = ...; // some file id
String requestId = fsClient.getFileName(fileId);

// get the response message in blocking mode
ResponseMessage response = (ResponseMessage)
fsClient.receiveResponse(requestId,true);

// get the response message in non blocking mode
ResponseMessage response = null;
```

```
while (response==null) {
      response = (ResponseMessage)
      fsClient.receiveResponse(requestId,false);
      // do something else here, as the method above doesn’t block
}

// extract the result from the response message
if (response.getException()==null) {
      String fileName = (String) response.getResult();
} else {
      // handle the exception
}
```

4.3.4 Perform file related operations

4.3.4.1 Upload a file, without over-writing an existing file

```
String requestId = fsClient.uploadFile(fileLocation,fileName);
FileUploadCheckStatus status =
fsClient.getFileUploadCheckStatus(requestId, true);
if (status==FileUploadCheckStatus.UPLOAD_ACCEPTED) {
       while(fsClient.isFileTransferWorkerAlive(requestId)) {
           Thread.sleep(500);
       }
      ResponseMessage response =
      (ResponseMessage) fsClient.receiveResponse(requestId,true);
      if (response != null && response.getException()==null) {
            String fileId = (String) response.getResult();
      } else {
            // handle the exception
      }
}
```

4.3.4.2 Upload a file, over-writing an existing file

```
String requestId = fsClient.uploadFile(fileId,fileLocation,fileName);
FileUploadCheckStatus status =
fsClient.getFileUploadCheckStatus(requestId, true);
if (status==FileUploadCheckStatus.UPLOAD_ACCEPTED) {
       while(fsClient.isFileTransferWorkerAlive(requestId)) {
           Thread.sleep(500);
       }

      ResponseMessage response =
      (ResponseMessage) fsClient.receiveResponse(requestId,true);
      if (response != null && response.getException()==null) {
            String fileId = (String) response.getResult();
      } else {
            // handle the exception
      }
} else {
      // status==FileUploadCheckStatus.UPLOAD_NOT_ACCEPTED
      // file is not accepted
}
```

4.3.4.3 Check if upload file operation is accepted

```
String requestId = fsClient.checkUploadFile(fileLocation, fileName);
ResponseMessage response = (ResponseMessage)
fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
      Boolean accepted = (Boolean) response.getResult();
} else {
      // handle the exception
}
```

4.3.4.4 Retrieve a file

```
String requestId = fsClient.retrieveFile(fileId,fileLocation);
While(fsClient.isFileTransferWorkerAlive(requestId)) {
    Thread.sleep(500);
}
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
       String fileName = (String) response.getResult();
       // use the file with the resulted file name
} else {
       // handle the exception
}
```

4.3.4.5 Get file name

```
String requestId = fsClient.getFileName(fileId);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
       String fileName = (String) response.getResult();
} else {
       // handle the exception
}
```

4.3.4.6 Get file size

```
String requestId = fsClient.getFileSize(fileId);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
       Long size = (Long) response.getResult();
} else {
       // handle the exception
}
```

4.3.4.7 Rename file

```
String requestId = fsClient.renameFile(fileId, fileName);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
       // response.getResult()==null
} else {
       // handle the exception
}
```

4.3.4.8 Remove file

```
String requstId = fsClient.removeFile(fileId);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
       // response.getResult()==null
} else {
       // handle the exception
}
```

4.3.5 Demonstrate stream-based uploading and retrieving (version 1.1)

The functionality for uploading and retrieving a file using streams is basically the same as for the local-file-persistence-based actions, and the other methods for managing files will work the same way. Here we will show one method to upload a file from a stream, then to retrieve it.

4.3.5.1 Upload a file in a stream, without over-writing an existing file, and give it a name

```
// a stream with file data
```

```
InputStream dataStream = new FileInputStream(new
File(ConfigHelper.CLIENT_FILE_LOCATION + "/" + UPLOAD_FILE_NAME));

String requestId = fileSystemClient.uploadFile(dataStream,
UPLOAD_FILE_NAME);

// check status
FileUploadCheckStatus status =
fileSystemClient.getFileUploadCheckStatus(requestId, true);
if (status == FileUploadCheckStatus.UPLOAD_ACCEPTED) {
     while (fileSystemClient.isFileTransferWorkerAlive(requestId)) {
         Thread.sleep(500);
    }

// get response message
ResponseMessage response = (ResponseMessage)
fileSystemClient.receiveResponse(requestId, true);
if (response != null && response.getException() == null) {
    String fileId = (String) response.getResult();
    // use the file with the resulted file name and remove file
    requestId = fileSystemClient.removeFile(fileId);
    response = (ResponseMessage)
       fileSystemClient.receiveResponse(requestId, true);
} else {
     // handle the exception
}
```

4.3.5.2 Retrieve a file

```
// a stream connected to the destination of the data
OutputStream outStream = new ByteArrayOutputStream();
requestId = fileSystemClient.retrieveFile("fileId", outStream);

while (fileSystemClient.isFileTransferWorkerAlive(requestId)) {
    Thread.sleep(500);
}

// get response message
ResponseMessage response = (ResponseMessage)
fileSystemClient.receiveResponse(requestId, true);
if (response != null && response.getException() == null) {
    String fileId = (String) response.getResult();
    // use the file with the resulted file name and remove file
    requestId = fileSystemClient.removeFile(fileId);
    response = (ResponseMessage)
        fileSystemClient.receiveResponse(requestId, true);
} else {
    // handle the exception
}
```

4.3.6 Perform group related operations

4.3.6.1 Create a group

```
String requestId = fsClient.createGroup(groupName, fileIds);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
      // response.getResult()==null
} else {
      // handle the exception
}
```

4.3.6.2 Update a group

```
String requestId = fsClient.updateGroup(groupName, newFileIds);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
      // response.getResult()==null
} else {
      // handle the exception
}
```

4.3.6.3 Retrieve a group

```
String requestId = fsClient.retrieveGroup(groupName);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
      List fileIds = (List) response.getResult();
      // iterate over the list of file ids
} else {
      // handle the exception
}
```

4.3.6.4 Remove a group

```
// the user can permanently remove the files from this group
boolean removeFiles = true;

// or, it can remove just the group and leave the files
boolean removeFiles = false;

String requestId = fsClient.removeGroup(groupName, removeFiles);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
      // response.getResult()==null
} else {
      // handle the exception
}
```

4.3.6.5 Add a file to a group

```
String requestId = fsClient.addFileToGroup(groupName, fileId);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
      // response.getResult()==null
} else {
      // handle the exception
}
```

4.3.6.6 Remove a file from a group

```
String requestId = fsClient.removeFileFromGroup(groupName, fileId);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
      // response.getResult()==null
} else {
      // handle the exception
}
```

4.3.7 Perform search operations

4.3.7.1 Search files using a regex pattern

```
// we must know the searcher’s name
String searcherName = “regex”;

String criteria = “.*\.jpg”;

String requestId = fsClient.searchFiles(searcherName, criteria);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
       List fileIds = (List) response.getResult();
       // iterate over the list of file ids
} else {
       // handle the exception
}
```

4.3.7.2 Search groups using a regex pattern

```
// we must know the searcher’s name
String searcherName = “regex”;

String criteria = “TopCoder.*”;

String requestId = fsClient.searchGroups(searcherName, criteria);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
       List groupNames = (List) response.getResult();
       // iterate over the list of group names
} else {
       // handle the exception
}
```

4.3.7.3 Search groups that contain some file id

```
// we must know the searcher’s name
String searcherName = “fileId”;

String fileId = ...; // some file id
String criteria = fileId;

String requestId = fsClient.searchGroups(searcherName, criteria);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
       List fileIds = (List) response.getResult();
       // iterate over the list of group names
} else {
       // handle the exception
}
```

4.4 Regular Expression constructs

Notes: the regular expression used in this component even in java, the pattern should be cared. It does similar to the Windows search function used, but has some difference. For example, *.jpg should be .*\.jpg.
That is say * should be like .*; . should be like \., jpg can remain the same.

Construct	**Matches**

Characters

x	The character *x*
`\\`	The backslash character
`\0`*n*	The character with octal value `0`*n* (0 <= *n* <= 7)
`\0`*nn*	The character with octal value `0`*nn* (0 <= *n* <= 7)
`\0`*mnn*	The character with octal value `0`*mnn* (0 <= *m* <= 3, 0 <= *n* <= 7)
`\x`*hh*	The character with hexadecimal value `0x`*hh*
`\u`*hhhh*	The character with hexadecimal value `0x`*hhhh*
`\t`	The tab character ('`\u0009`')
`\n`	The newline (line feed) character ('`\u000A`')
`\r`	The carriage-return character ('`\u000D`')
`\f`	The form-feed character ('`\u000C`')
`\a`	The alert (bell) character ('`\u0007`')
`\e`	The escape character ('`\u001B`')
`\c`*x*	The control character corresponding to *x*

Character classes

`[abc]`	`a`, `b`, or `c` (simple class)
`[^abc]`	Any character except `a`, `b`, or `c` (negation)
`[a-zA-Z]`	`a` through `z` or `A` through `Z`, inclusive (range)
`[a-d[m-p]]`	`a` through `d`, or `m` through `p`: `[a-dm-p]` (union)
`[a-z&&[def]]`	`d`, `e`, or `f` (intersection)
`[a-z&&[^bc]]`	`a` through `z`, except for `b` and `c`: `[ad-z]` (subtraction)
`[a-z&&[^m-p]]`	`a` through `z`, and not `m` through `p`: `[a-lq-z]`(subtraction)

Predefined character classes

.	Any character (may or may not match line terminators)
`\d`	A digit: `[0-9]`
`\D`	A non-digit: `[^0-9]`
`\s`	A whitespace character: `[ \t\n\x0B\f\r]`
`\S`	A non-whitespace character: `[^\s]`
`\w`	A word character: `[a-zA-Z_0-9]`
`\W`	A non-word character: `[^\w]`

POSIX character classes (US-ASCII only)

`\p{Lower}`	A lower-case alphabetic character: `[a-z]`
`\p{Upper}`	An upper-case alphabetic character:`[A-Z]`
`\p{ASCII}`	All ASCII:`[\x00-\x7F]`

`\p{Alpha}`	An alphabetic character: `[\p{Lower}\p{Upper}]`
`\p{Digit}`	A decimal digit: `[0-9]`
`\p{Alnum}`	An alphanumeric character: `[\p{Alpha}\p{Digit}]`
`\p{Punct}`	Punctuation: One of `` !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~ ``
`\p{Graph}`	A visible character: `[\p{Alnum}\p{Punct}]`
`\p{Print}`	A printable character: `[\p{Graph}]`
`\p{Blank}`	A space or a tab: `[ \t]`
`\p{Cntrl}`	A control character: `[\x00-\x1F\x7F]`
`\p{XDigit}`	A hexadecimal digit: `[0-9a-fA-F]`
`\p{Space}`	A whitespace character: `[ \t\n\x0B\f\r]`

Classes for Unicode blocks and categories

`\p{InGreek}`	A character in the Greek block (simple block)
`\p{Lu}`	An uppercase letter (simple category)
`\p{Sc}`	A currency symbol
`\P{InGreek}`	Any character except one in the Greek block (negation)
`[\p{L}&&[^\p{Lu}]]`	Any letter except an uppercase letter (subtraction)

Boundary matchers

`^`	The beginning of a line
`$`	The end of a line
`\b`	A word boundary
`\B`	A non-word boundary
`\A`	The beginning of the input
`\G`	The end of the previous match
`\Z`	The end of the input but for the final terminator, if any
`\z`	The end of the input

Greedy quantifiers

X?	*X*, once or not at all
*X**	*X*, zero or more times
X+	*X*, one or more times
X{*n*}	*X*, exactly *n* times
X{*n*,}	*X*, at least *n* times
X{*n*,*m*}	*X*, at least *n* but not more than *m* times

Reluctant quantifiers

X??	*X*, once or not at all

`X*?`	*X*, zero or more times
`X+?`	*X*, one or more times
`X{n}?`	*X*, exactly *n* times
`X{n,}?`	*X*, at least *n* times
`X{n,m}?`	*X*, at least *n* but not more than *m* times

Possessive quantifiers

`X?+`	*X*, once or not at all
`X*+`	*X*, zero or more times
`X++`	*X*, one or more times
`X{n}+`	*X*, exactly *n* times
`X{n,}+`	*X*, at least *n* times
`X{n,m}+`	*X*, at least *n* but not more than *m* times

Logical operators

`XY`	*X* followed by *Y*
`X|Y`	Either *X* or *Y*
`(X)`	X, as a capturing group

Back references

`\n`	Whatever the *n*th capturing group matched

Quotation

`\`	Nothing, but quotes the following character
`\Q`	Nothing, but quotes all characters until `\E`
`\E`	Nothing, but ends quoting started by `\Q`

Special constructs (non-capturing)

`(?:X)`	*X*, as a non-capturing group
`(?idmsux-idmsux)`	Nothing, but turns match flags on - off
`(?idmsux-idmsux:X)`	*X*, as a non-capturing group with the given flags on - off
`(?=X)`	*X*, via zero-width positive lookahead
`(?!X)`	*X*, via zero-width negative lookahead
`(?<=X)`	*X*, via zero-width positive lookbehind
`(?<!X)`	*X*, via zero-width negative lookbehind
`(?>X)`	*X*, as an independent, non-capturing group

For more details, please see Pattern class's javadoc or other book about regular expression in java.

4.5 FreeDiskSpaceNonNativeChecker Impelementation

1. *Check the “os.name” system property, to get the operating system information.*
2. if it is used in windows system,
   normailize the path in windows system, and get the disk label.
   build and run the 'dir' command
   read in the output of the command to an ArrayList
   now iterate over the lines we just read and find the LAST non-empty line (the free space bytes should be in the last element of the ArrayList anyway, but this will ensure it works even if it's not, still assuming it is on the last non-blank line)
   if found, remove commas and dots in the bytes count.
   Return the free disk space size.
3. if it is used in *nix system.
   normalize the path in *nix ,
   build and run the 'df' command
   read the output from the command until we come to the second line
   tokenize the string. The fourth element is what we want.
   Return the free disk space size.

5. Future Enhancements

Plug different implementations of the interfaces.