CORBA - FTAM/FTP Interworking Specification

dtc/2001-08-06
Annotated revisions from telcom/00-11-05

Note – Company list updated

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Note – This is the table of contents from the original specification. The new table of contents follows

Note – This is the new table of contents

Appendix A Complete OMG IDL 1

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Associated OMG Documents

The CORBA documentation set includes the following:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBA Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
- *CORBAservices: Common Object Services Specification* contains specifications for OMG's Object Services.
- *CORBAfacilities: Common Facilities Specification* includes OMG's Common Facility specifications.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Med*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted parts of this specification:

- Ericsson
- ~~Siemens~~
- Siemens AG
- Broadcom Eireann Research
- Distributed Systems Technology Centre
- Floorboard
- IONA
- Lucent
- PrismTech
- University of California, Irvine

~~File Transfer in Telecoms Systems~~

Service Description 1

1.1 File ~~Transfer~~Transfer in Telecoms Systems

Retrieving data from a remote Network Element (NE) and maintaining the software that runs on that node is relatively straightforward but performing the same operations on potentially thousands of Network Elements presents the telecommunication operator with a significant challenge. These tasks are currently performed using either the ISO specified File Transfer, Access and Maintenance (FTAM) protocol or the File Transfer Protocol (FTP). Currently Operations Support Systems (OSS) employ either FTAM or FTP to perform both data retrieval and software maintenance tasks.

This specification describes a single set of IDL interfaces that will allow any OSS to perform its file management operations on underlying Network Elements regardless of the type of file management mechanism the underlying node is using. There are a number of reasons that identify the need for such interfaces:

- OSSs may be implemented in a large number of programming languages and deployed in a platform-independent manner. In addition to using existing OSS systems, telecommunication operators may also employ an alternative, lightweight OSS client that has all of the features of the legacy systems but performs the management of Network Elements through the IDL interfaces.
- The complexity of performing data retrieval and file maintenance operations is hidden from the OSS user by a single set of IDL interfaces. No knowledge of -FTP, FTAM, or other file access mechanisms is necessary for them to perform their job.

- The task of extending the set of data retrieval and file maintenance operations is made easier. New management or retrieval operations to meet changing requirements may be exposed to the OSS through a new IDL interface. Existing OSSs may continue to use the original IDL interfaces without interruption.
- The task of migrating a large installed base of OSSs to use a new file management mechanism will be less complex and take considerably less time to perform since the same set of IDL interfaces is being used.

There are a number of system configurations that are possible through the deployment of the proposed interfaces. One such configuration is illustrated in Figure 1-1.

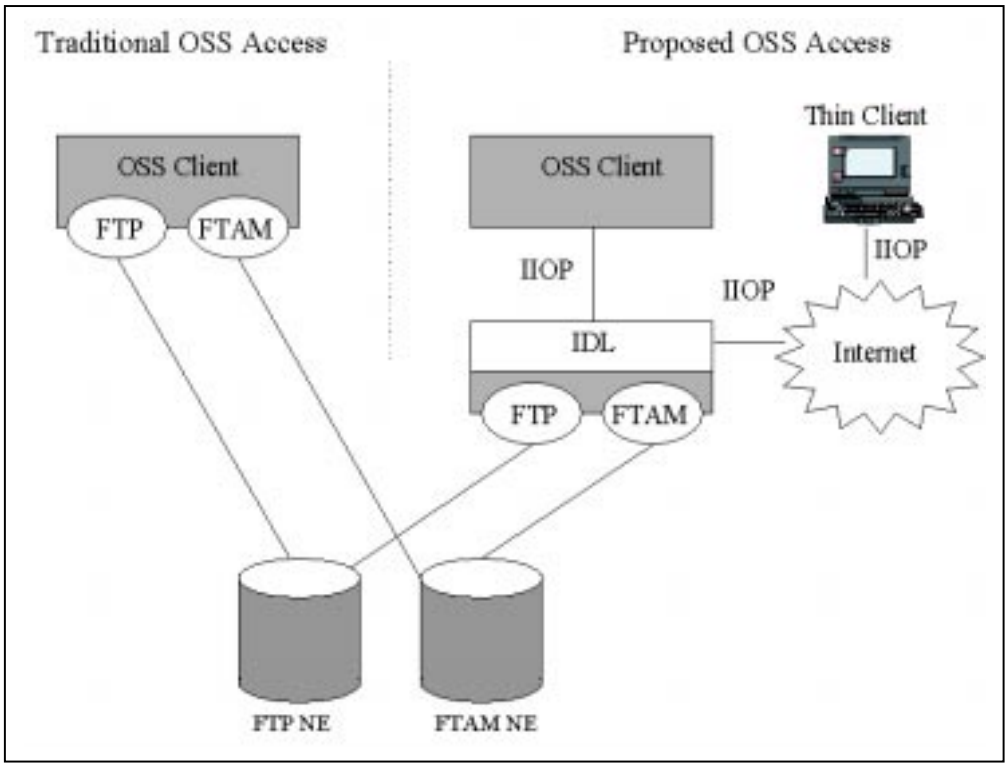

Figure 1-1 High-level system overview

Traditionally different file transfer clients were required for each type of fileserver within the telecoms OSS. By exposing basic file transfer functionality through a set of IDL interfaces it is possible to develop less complex file transfer clients that are independent of the underlying file transfer protocols. The use of CORBA allows remote management of systems over corporate intranets.

1.1.1 File Transfer Capable Network Elements

The primary focus of this specification is defining a file transfer IDL that provides uniform access to FTAM and FTP NEs. However, the scope and utility of the file transfer IDL is not limited to use ~~soley~~ with only FTAM and FTP. Any NE ~~can~~ may support the file transfer IDL ~~to transfer information~~for data transfer. Clients ~~using the file transfer IDL will~~ often transfer ~~a file~~ files to a local file system, which itself can be represented by the IDL. Non-file based information can also be transferred. For example, a NE may support access to operational and performance data through "virtual" ~~files~~files and directories, accessible by the file transfer IDL~~, even though the~~. The NE itself may not actually store this data in ~~a~~ physical ~~file~~files and directories.

~~Architectural Overview~~Service

Architecture 2

2.1 Overview

This ~~specification proposes~~ service defines a set of interfaces that ~~implement~~ model a ~~distributed~~ simplified virtual file ~~transfer, access and maintenance framework. The various interfaces of this framework include:~~system.

- ~~**Virtual File Systems** – represented by the **VirtualFileSystem** interface~~
- ~~**File Transfer Sessions** – represented by the **FileTransferSession** interface~~
- ~~**File** – represented by the **File** interface~~
- ~~**Directory** – represented by the **Directory** interface~~

~~The **VirtualFileSystem** interface provides users with a standard interface to a specific file management system. It facilitates the authentication of the user by establishing a trusted relationship between them and the remote server. For each successful user authentication, the **VirtualFileSystem** provides the user with a **FileTransferSession** enabling them to conduct file management operations across the framework. A different implementation of the **FileTransferSession** interface is provided depending on the protocol used by the remote file transfer mechanism. The **FileTransferSession** must maintain a direct communication link with the remote file transfer mechanism. Additionally it must ensure that user operation invocations are mapped correctly to the appropriate protocol specific primitives that will be passed to the remote server or responder. Similarly, the **FileTransferSession** must ensure that reply messages returned by the remote server or responder will be translated into non-protocol specific messages.~~

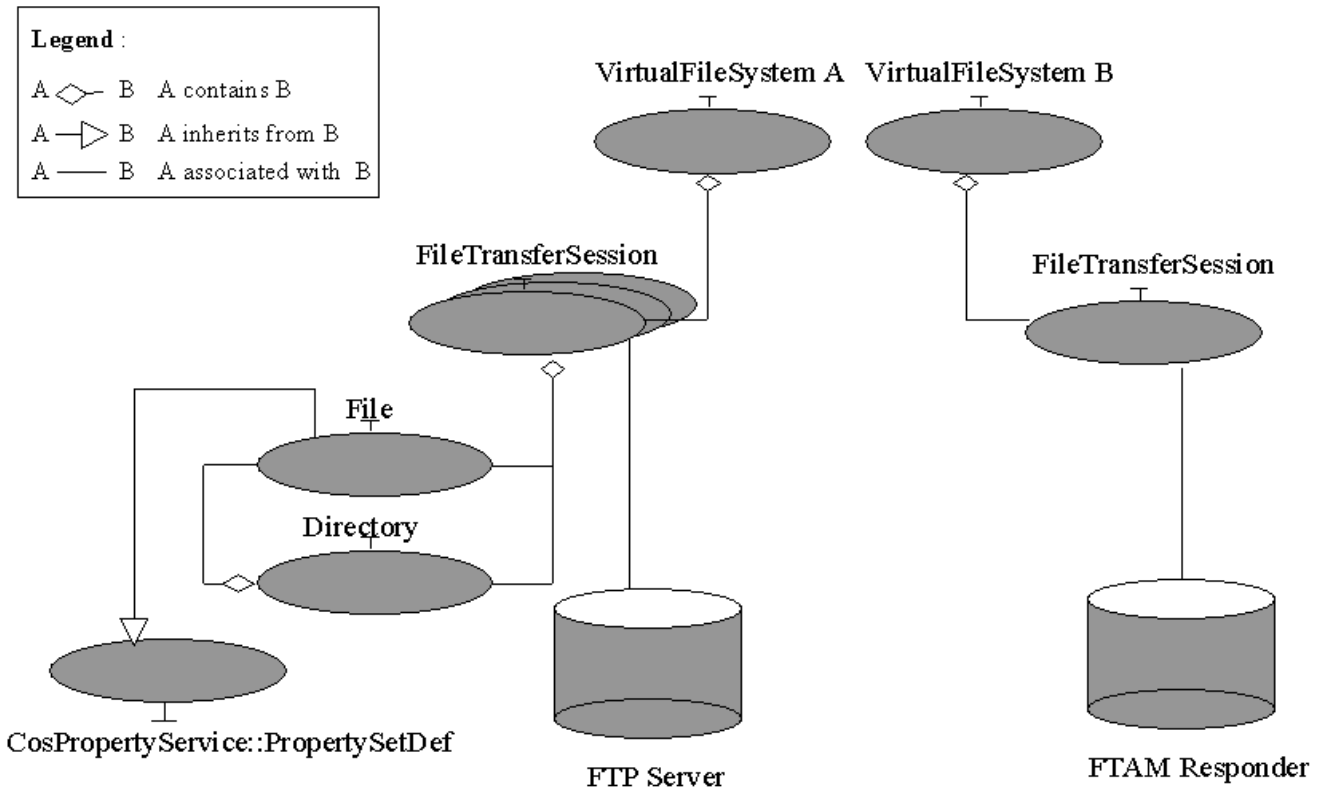

Figure 2-1 An example File Transfer Configuration

Figure 2-1 illustrates how **FileTransferSession** manages a group of **File** and **Directory** references that represent the real files and directories that reside within the currently active working directory of the file server.

The **File** interface provides the client with a proxy to a physical file stored on the File Transfer Server and represents the basic unit of the file system. The **Directory** interface is a specialization of the **File** Interface that maintains information relative to a physically related group of **File** objects. Although it is derived from the **File** Interface, it also provides the ability to list all **File** references maintained by the **Directory**.

A client obtains access to a file system by logging in and accessing an initial directory. A directory provides access to the file system entries that it contains. A file system entry is a data file or a directory.

A client may perform basic maintenance tasks on file system entries. A client may also log on to multiple file systems to transfer files between them. The types of operations a client may perform include:

- Copy, insert, or append the contents a file to another file
- List the entries in a directory.
- Create a new directory.
- Remove an existing directory or file.
- Query a file or directory for properties such as creation time or size.

An implementation may restrict a client's access to any particular file, directory, property, or operation based on the credentials the client used to login to the file system.

2.1.1 File System Servers

The files and directories a client accesses through the service interfaces are virtual proxies for entities internal to the service. The specification places no restrictions on the internal structure or form of these entities.

The service interface is capable of providing virtual file systems for:

- FTP servers
- FTAM responders
- Local file systems
- NEs presenting arbitrary data as virtual files and directories through the service interfaces.

No details specific to FTAM, FTP, or a specific NE are exposed in the IDL. A client is unaware of the underlying service implementation and may transfer files between services through a CORBA interface or another negotiated transfer protocol such as FTP.

2.1.2 Principal Components

The **CosFileTransfer** module defines the following primary interfaces:

- FileSystem - The virtual file system the service represents.
- FileSession - The login session a client is granted to access the file system.
- FileSystemEntry - A base interface providing common operations for files and directories.
- Directory - A virtual directory that a client can list the entries in.
- DirEntryIterator - An iterator to access a list of file system entry properties.
- File - A virtual file that can be copied, inserted, or appended to another file.

The following two interfaces provide more advanced transfer control and direct access to a file's content:

- TransferEndPoint - An object that represents one end of a file's transfer connection. It is used for a single transfer.
- OctetTransferIterator - An iterator to read and write file contents.

The above two interfaces are used internally by a service implementation to provide the basic file transfer operations.

2.1.3 Files and Directories

Names

FileSystem entries have a simple single component name, **EntryName**, that is unique to their immediate parent **Directory** and a multi-component **EntryPath** that is relative to any ancestor **Directory**.

Basic Maintenance Operations

The basic operations such as **get_path**, **remove**, **exists**, **create_directory**, are described starting in Section 3.1 .

Directory Lists

The following pseudo-code illustrates logging in to a **FileSystem** and listing the names of the entries.:

```
...
session = fileSys.login(user, password, lprops, home_dir);

// relative dir path: “sub1/sub2/dir3”
String [] dirPath = {
    “sub1”, “sub2”, “dir3”
}

subDir = home_dir.get_directory(dirPath);

// desired properties: file name and size
String[] dirProps = {
    “name”, “size”
}

entryItor = subDir.list(dirProps);

// Iterate through entries, printing returned properties
offset = 0;
if (entryItor != null){
    do{
        entries = entryItor.next(0,0);
        for(e=0; e<entries.length(); ++e){
            printNameAndSize(entries[e]);
        }
        offset += entries.length();
    }
    while(entries.length()!=0);
}

session.destroy();
```

2.1.4 File Transfer

The service transfers files between file systems. The protocol used for the transfer is negotiated when the transfer is initiated. The supported protocols are:

- `CORBA` - “IDL:omg.org/CosFileTransfer/OctetTransferIterator:1.0” - mandatory
- `FTP` - optional
- `FTAM` - optional
- Additional CORBA interfaces - optional

Clients are coded identically regardless of the transfer protocol used. **OctetTransferIterator** support is mandatory to guarantee that any two service implementations will be able to transfer files if no other common transfer protocol is available. A service may offer additional CORBA transfer interfaces besides this.

Binary File Transfer

All file transfers are binary. This service has no concept of character code-sets and does not make a distinction between text and binary files as defined by ftp and ftam.

High Level File Transfer Operations

Basic file transfer operations for transferring data from one file system to another are available on the **File** interface. The pseudo-code below illustrates logging on to two file systems and performing the high level transfer operations: **copy**, **append**, and **insert**. The full IDL descriptions are in Section 3.1, “CosFileTransfer Module”.

```
fromSess = fsFrom.login(user1, password1, lprops1, dirFrom);
toSess   = fsTo.login(user2, password2, lprops2, dirTo);

String[] fromName = {
    // filename is: “from_dir_name/from_file_name”
    “from_dir_name”, “from_file_name”
};

String [] toName = {
    // filename is: “to_dir_one/to_dir_two/to_file_name”
    “to_dir_one”, “to_dir_two”, “to_file_name”
};

fromFile = dirFrom.get_file(fromName, true); // must exist
toFile   = dirTo.get_file(toName, false);    // need not

fromFile.copy(toFile);
fromFile.append(toFile);
fromFile.insert(toFile, 1024);

fromSess.destroy();
toSess.destroy();
```

When the client is finished, the file sessions are destroyed to release all server resources. Support for the **append** and **insert** operations is optional.

File Transfer Implementation

Additional transfer primitives are required for services to implement the high level transfer operations described above. Clients may also use these primitives to directly control more advanced transfer operations.

To implement a file transfer, the **File** interface has a few additional methods. The interface **TransferEndPoint** is defined to represent a file’s connection endpoint for the duration of a single file transfer.

A transfer between two **Files** is carried out in the following steps.

1. Negotiate the protocol to be used for the file transfer:
   - Determine a common transfer protocol: `ftp`, `ftam`, or a `corba` interface.
   - Determine which end point of the transfer connection will wait for connection, *the passive end point*, and which end will actively connect, *the active endpoint*.
2. Create the appropriate **TransferEndPoint** objects for each **File**.
3. The passive endpoint is put in a listening state, awaiting connection.
4. The active endpoint makes the connection.
5. The passive endpoint is notified the active connection has been made.
6. The transfer operation is called on the source endpoint.

These steps are described in more detail in the next sections.

Protocol Negotiation

The method **File::get_transfer_protocols** returns a preference ordered list of the transfer protocols supported by the **File**. Some example return lists are:

`"IDL:omg.org/CosFileTransfer/OctetTransferIterator:1.0"`

`"ftp"`

This list says that the **File** can be transferred using either the specified corba interface or a `ftp` data connection in either `active` or `passive` mode. Support for the **CosFileTransfer::OctetTransferIterator** interface is mandatory. In this case it is listed to indicate that it is preferred over `ftp`.

`"ftp;active"`

`"IDL:CompanyX.com/CryptoTransfer/CompressedIterator:1.0"`

`"ftam;passive"`

This list says that the **File** can be transferred using `ftp` if the **File** actively makes the data connection. If `ftp` cannot be used, the specified corba interface is the next preferred transfer protocol. Finally, `ftam` may be used with this endpoint taking on the passive role. Since support for the **OctetTransferIterator** interface is mandatory it is not required to be listed.

To transfer from **File** A to **File** B, the **Files** are queried for their supported protocols. This list is examined and a compatible set is chosen. An example being "`ftp;active`" for **File** A and "`ftp;passive`" for **File** B. If a transfer protocol string does not specify active or passive, it supports both. This is always the case for the **OctetTransferIterator** protocol.

Transfer protocol syntax is specified in Section 2.2.1 .

TransferEndPoint Creation

The method **File::create_transfer_endpoint** is used to create the necessary **TransferEndPoints**. It takes arguments that specify whether this endpoint is the source or a destination of the transfer, the read/write offset into the **File**, and whether

the offset is relative to the beginning or end of the **File**. These parameters can specify endpoints usable as the source or sink of **copy**, **append**, and **insert** operations. See section Section 3.1.7 for details.

Passive Endpoint Listen

The passive **TransferEndPoint** is put into a wait for connection (listening) state by calling **go_to_listen**. It is then ready to accept a connection from the active **TransferEndPoint**. This method returns a **TransferDetail** describing the passive endpoint.

Active Endpoint Connection

The active **TransferEndPoint** completes the connection circuit when **connect_to_peer** is called. The argument to this method is the **TransferDetail** returned from **go_to_listen**. This method returns a **TransferDetail** string describing the active endpoint protocol specific details. For some protocols, the returned **TransferDetail** may be an empty string.

Passive Endpoint Connect Notify

The last step in the connection establishment is calling **set_peer** on the passive endpoint to notify it that the connection has been made. The argument to this method is the **TransferDetail** returned from the **connect_to_peer** operation. For some protocols, **set_peer** may accept an empty string.

Low Level Transfer Example

The following example illustrates the execution of an append operation, where the negotiated protocol is "`ftp`". The sender is `passive` and the receiver is `active`.

```
...
fromFile = dirFrom.get_file(fromName);
toFile   = dirTo.get_file(toName);

fromProtocols = fromFile.get_end_point_protocols();
toProtocols   = toFile.get_end_point_protocols();

// From the protocol lists, find a matching
// protocol set. “ftp” is used for this example,
// the sender will be passive, listening
// for ftp data connection
...
fromProtocol = “ftp;passive”;
toProtocol = “ftp;active”;

// create endpoints to append the file

fromEP =
fromFile.create_endpoint(TransferEndPointRole::SOURCE,
                                    FilePos::BEGIN,
                                    0,
                                    fromProtocol);

toEP = fromFile.create_endpoint(TransferEndPointRole::SINK,
                                    FilePos::END,
                                    0,
                                    toProtocol);

// establish connection
passiveDetail = fromEP.go_to_listen();
activeDetail  = toEP.connect_to_peer(passiveDetail);
fromEP.set_peer(activeDetail);

fromEP.transfer();

fromEP.destroy();
toEP.destroy();
```

This example would follow the same form if a different transfer protocol were used. To change the operation to a **copy**, the **SINK** endpoint would have **FilePos::BEGIN** and offset of zero. Inserts are performed by specifying a **TransferEndPointRole** of **SINK_INSERT** for the destination endpoint. An implementation may restrict the types of **TransferEndPoints** supported.

Direct File Access

To allow direct access to the contents of a file from a client that cannot provide another **TransferEndPoint** or **File**, the **OctetTransferIterator** interface can be used to read and write file contents directly. An example of reading the contents of a "text" file for display is shown in the pseudo-code below:

```
...
protocol =
"IDL:omg.org/CosFileTransfer/OctetTransferIterator:1.0"
fromEP =
fromFile.create_endpoint(TransferEndPointRole::SOURCE,
                                    FilePos::BEGIN,
                                    0,
                                    protocol);

// go_to_listen returns "IOR:...."
// as the TransferDetail for a corba protocol

corbaDetail = fromEP.go_to_listen();
octetItorObj = orb.string_to_object(corbaDetail);
octetItor = OctetTransferIterator.narrow(octetItorObj);

do{
    octetBuf = octetItor.get_octet_seq(offset, 0);
    printBuffer(octetBuf); // print file as text
    offset = offset + octetBuf.length();
}
while(octetBuf.length()!=0);

fromEP.destroy();
```

2.2 File Transfer Protocols

This section describes the details of the supported file transfer protocols.

2.2.1 Protocol Syntax

The protocol syntax defines protocol names and protocol specific attributes. The syntax is extensible to allow new protocols and attributes to be added. The syntax for the currently supported protocols is:

<ProtocolSpec> ::= <CORBA> | <FTP> | <FTAM> | <NewProtocol>

<CORBA> ::= <OctetTransfer> | <OtherCORBA>
<OctetTransfer> ::=
“IDL:org.omg.CosFileTransfer/OctetTransferIterator:1.0”
<OtherCORBA> ::= <InterfaceID> [<Options>]
<InterfaceID> ::= Valid Repository ID

<FTP> ::= “ftp” [<ActivePassiveOption>]
<FTAM> ::= “ftam” [<ActivePassiveOption>]

<ActivePassiveOption> ::= “;” [“active” | “passive”]
<NewProtocol> ::= <AlphaNumericString> [<Options>]
<Options> ::= “;” <Tag>[“=” <Value>][<Options>]
<Tag> ::= <AlphaNumericString>
<Value> ::=<AlphaNumericString>

2.2.2 Transfer Connection Establishment

Service implementations and clients using transfer primitives are required to use connection establishment semantics that are functionally equivalent to the following:

```
// protocol independent connection establishment
passiveDetail = passiveEP.go_to_listen();
activeDetail  = activeEP.connect_to_peer(passiveDetail);
passiveEP.set_peer(activeDetail);
```

The one exception is if a client is directly accessing a **File** using the **OctetTransferIterator** interface as described previously in the “Direct File Access” section. In this case only, it sufficient to call **go_to_listen** and then use the returned **OctetTransferIterator** immediately.

2.2.3 CORBA Transfer Protocol

The following is required for a service implementation to support a `corba` transfer protocol.

File::create_end_point must return a corba aware **TransferEndPoint** when the endpoint protocol argument begins with an interface repository ID.

TransferEndPoint::go_to_listen must return a stringified object reference that can be passed to **TransferEndPoint::go_to_listen** or used directly by a client.

TransferEndPoint::connect_to_peer must return a stringified object reference that can be passed to **TransferEndPoint::set_peer**.

The **OctetTransferIterator** corba protocol does not have a concept of active or passive, so either endpoint can be used as passive or active. This may not be true for other corba transfer interfaces. An implementation supporting **OctetTransferIterator** may implement the high level transfer operations in a manner similar to the one outlined by the example in the “Direct File Access” section above.

There is no requirement for an implementation to make use of the stringified object reference that is passed to **set_peer** for a `corba` transfer protocol.

An implementation must allow the **set_peer** argument to be an empty string. This represents the case where a client is using an **OctetTransferIterator** directly.

2.2.4 FTP Transfer Protocol

The ftp transfer protocol, refers specifically to a file transfer that takes place as if it were the data connection of an ftp[1] service transfer. A service implementation need not use a true ftp server to implement this transfer protocol.

The following is required for a service implementation to support the `ftp` transfer protocol.

File::create_end_point must return an ftp aware **TransferEndPoint** when the endpoint protocol argument an ftp type.

TransferEndPoint::go_to_listen must return a string of the form:

```
host:port
```

where host is either a DNS style host name or a dotted decimal IP address and port identifies the port number that will accept the `ftp` data connection.The returned `host:port` string is passed to **TransferEndPoint::go_to_listen**.

TransferEndPoint::connect_to_peer must return a host:port string identifying the local end of the ftp data connection that has been established. In some cases this information may not be available, in which case an empty string is returned. The returned string is passed to **TransferEndPoint::set_peer**.

There is no requirement for an implementation to make use of the `host:port` that is passed to **set_peer** for the `ftp` transfer protocol.

2.2.5 FTAM Transfer Protocol

The following is required for a service implementation to support the `ftam`[2] transfer protocol.

1. IETF RFC 959 “File Transfer Protocol (FTP)”, J. Postel, J.Reynolds. October 1985

2. ISO/IEC 8571-1,8571-2,8571-3,8571-4 Information Processing Systems - Open Systems Interconnection - File Transfer, Access, and Management Parts 1 - 4. 1993

File::create_end_point must return an ftam aware **TransferEndPoint** when the endpoint protocol argument an ftam type.

TransferEndPoint::go_to_listen must return a string identifying a ftam responder.

The returned `responder` string is passed to **TransferEndPoint::go_to_listen**.

TransferEndPoint::connect_to_peer must return a string identifying the ftam initiator. The returned string is passed to **TransferEndPoint::set_peer**.

~~Principal Components~~Service

Interfaces 3

~~3.1 Virtual File System~~

~~A Virtual File System abstracts remote file servers and is described by the IDL interface **VirtualFileSystem**. All Virtual File Systems support operations to allow a user to login to a remote file server without prior knowledge of the underlying protocols that remote system employs. For example, a user shall call the following operation on the **VirtualFileSystem** interface:~~

~~**FileTransferSession login(in Istring username, in Istring password, in Istring account, out Directory root) raises(SessionException, FileNotFoundException, IllegalOperationException);**~~

~~The parameters **username** and **password** specify the client's name and their password for the remote system associated with the Virtual File System. Similarly the **account** parameter determines the specific account at the remote system they are attempting to login to. The **account** parameter is optional since some clients may not have more than one designated account. In this circumstance a client shall invoke the **login()** operation with their **username**, **password** and an empty string for the **account** parameter to indicate the login is for the default user account. If successful, **login()** shall return a reference to a **FileTransferSession**, in addition to a reference to the starting **Directory**. The client may then utilize the operations at the remote system via the **FileTransferSession**.~~

~~The **login()** operation is capable of throwing three exceptions:~~

- ~~a SessionException is thrown by **login()** if it is not possible to establish a connection with the remote file server represented by the **VirtualFileSystem** (for example, if the user's details have not been validated successfully).~~

- a FileNotFoundException is thrown by **login()** if the root directory at the remote file server cannot be determined.
- an IllegalOperationException is thrown by **login()** if the client does not have permission to access the root directory.

3.2 File Transfer Session

The **FileTransferSession** interface abstracts a period of communication between a file transfer client and a remote file transfer server that is maintained by the framework. A **FileTransferSession** is created in response to a successful login and provides the user with an interface to the remote server. When a client chooses to quit a current session, the **FileTransferSession** associated with that session is destroyed. This interface attempts to abstract the complexity associated with either the FTAM or FTP protocols from the client by providing a set of operations related to generic file transfer.

The **FileTransferSession** interface provides the **transfer()** operation that allows a file transfer client to transfer a **File** from a source to a destination file transfer server.

```
void transfer(in File src, in File dest)
raises(SessionException, TransferException,
 FileNotFoundException, RequestFailureException,
 IllegalOperationException);
```

The **src** parameter of the **transfer()** operation is used to identify the file to be transferred and the **dest** parameter is used to specify the destination that the file will be copied to. The **transfer()** operation abstracts the complexities associated with the mechanism for establishing a connection between a source and target **FileTransferSession**. Although the client is not concerned with the **protocols_supported** attribute exposed by the **FileTransferSession** interface, it is provided to enable implementations to establish data connections between source and target **FileTransferSession**s during a transfer. A detailed description of this attribute and its use by one possible implementation of the specification during a transfer is provided in page 4 ¶ Font>.

The **transfer()** operation can throw the following exceptions:

- a SessionException is thrown by **transfer()** when the existing connection with the remote server is not available for the requested transfer.
- a TransferException is thrown by **transfer()** when an error occurs with the current connection between the source and destination **FileTransferSession**s during the transfer of data.
- a FileNotFoundException is thrown by **transfer()** when the file to be transferred cannot be found at the remote file server.
- a RequestFailureException is thrown by **transfer()** if the requested transfer cannot be completed due to an internal problem at the gateway.

- an IllegalOperationException is thrown by **transfer()** if the user does not have read access for the source file or if the user is not permitted to create the new file at the target location on the destination file transfer server.

In some circumstances a client may wish to insert the contents of a file into a target file that resides on the remote system. The **FileTransferSession** interface provides the **insert()** operation to facilitate such a scenario. Invoking the operation requires that the client provide the following parameters: the **src** parameter identifies the file to be added to the remote file, the **dest** parameter identifies the file that the **src** file is to be added to and the **offset** parameter indicates the location in the **dest** file where the **src** will be added.

```
void insert (in File src, in File dest, in long offset)
raises(CommandNotImplementedException,
 SessionException,
 TransferException,
 FileNotFoundException,
 RequestFailureException,
 IllegalOperationException);
```

The **FileTransferSession** interface also provides the **append()** operation to enable a client to append the contents of a specified file onto the end of another file.

```
void append (in File src, in File dest)
raises(CommandNotImplementedException,
 SessionException,
 TransferException,
 FileNotFoundException,
 RequestFailureException,
 IllegalOperationException);
```

Both the **insert()** and **append()** operations are capable of throwing the same set of exceptions:

- a CommandNotImplementedException is thrown by **insert()** and **append()** if the remote server abstracted by the current **FileTransferSession** does not implement the insert or append functionality.
- a SessionException is thrown by **insert()** when the existing connection with the remote server is not available for the requested insert and by **append()** when the existing connection with the remote file server is not available for the requested append.
- a TransferException is thrown by **insert()** and **append()** when an error occurs with the current connection between the source and destination **FileTransferSession**s during the transfer of data.
- a FileNotFoundException is thrown by **insert()** and **append()** when the source file for the operation cannot be found at the remote server or when the destination file for the operation cannot be found at the target server.

- a RequestFailureException is thrown by **insert()** and **append()** when the requested insert or append cannot be completed due to an internal problem at the gateway.
- an IllegalOperationException is thrown by **insert()** and **append()** if the user does not have read access for the source file or if the user is not permitted to write to the file at the target location on the destination file transfer server.

The **FileTransferSession** interface also provides the **create_directory()** operation that allows a client to create a new directory on the remote server. This returns a reference to the **Directory** interface that may be used by the client to obtain complimentary information pertaining to the directory they have just created.

Directory create_directory (in FileNameList name)
raises(SessionException, FileNotFoundException,
RequestFailureException, IllegalOperationException);

The **create_directory()** operation takes the **name** parameter that specifies the name of the directory to be created and its full pathname as a sequence of strings. This operation is independent of the working directory from where it is invoked since the absolute pathname of the file is specified by the **name** parameter.

The **FileTransferSession** interface also defines the **create_file()** operation that can be used to create a proxy to a physical file at the remote file server before a transfer of a file can occur. **create_file()** is invoked by specifying the name of the file to be created, and its full pathname using the **name** parameter. In the same manner as the **create_directory()** operation, a client can create a new file at a target location on the remote server without concern for the current working directory.

File create_file (in FileNameList name)
raises(SessionException, FileNotFoundException,
RequestFailureException, IllegalOperationException);

Both the **create_directory()** and **create_file()** operations can throw the following exceptions:

- a SessionException is thrown by **create_directory()** when the existing connection with the remote server is not available when attempting to create the new directory and by **create_file()** when the existing connection with the remote server is not available when attempting to create the new file.
- a FileNotFoundException is thrown by **create_directory()** and **create_file()** when the pathname of the directory or file to be created, that is specified by the **name** parameter, is not found on the remote server. For example, the FileNotFoundException will be thrown by this operation if any directory in the specified path (referenced within a **FileNameList**) does not exist.
- a RequestFailureException is thrown by **create_directory()** and **create_file()** when the request cannot be completed due to an internal problem at the gateway.

- an IllegalOperationException is thrown by **create_directory()** and **create_file()** when the user is not permitted to access the destination directory or file at the target location.

Another operation typically associated with a file transfer service is for the client to navigate around the remote file system. When a client attempts to use a reference to a **Directory** object for some operation it is not guaranteed that its contents accurately reflect the contents of the physical directory it mirrors at the remote server. The **FileTransferSession** interface defines the **set_directory()** operation enabling the client to update or populate the contents of a **Directory** object specified by the **new_directory** parameter.

```
void set_directory (in Directory new_directory)
raises(SessionException, FileNotFoundException,
 RequestFailureException, IllegalOperationException);
```

A user application shall invoke **create_file()** or **create_directory()** passing a **FileNameList** as a parameter. The value of the **FileNameList** parameter indicates the absolute pathname of the file or directory that is to be created. Employing the use of absolute pathnames when invoking **create_file()** or **create_directory()** ensures that the creation of a file or directory is not affected by the invocation of the **set_directory()** operation.

The **set_directory()** operation can throw the following exceptions:

- a SessionException is thrown by **set_directory()** when the existing connection with the remote server is not available when attempting to change directory.
- a FileNotFoundException is thrown by **set_directory()** when the target directory cannot be found on the remote server.
- a RequestFailureException is thrown by **set_directory()** when it is not possible to change directory at the remote server.

3.3 CosFileTransfer Module

This chapter describes the **CosFileTransfer** module in detail.

3.3.1 Exceptions

The following IDL shows the exceptions defined for the service:

```
    typedef short ErrorCode;
    const ErrorCode UNSPECIFIED       = 0;
    const ErrorCode UNAVAILABLE       = 1;
    const ErrorCode UNSUPPORTED       = 2;
    const ErrorCode NO_PERMISSION     = 3;

    const ErrorCode ENTRY_EXISTS      = 4;
    const ErrorCode ENTRY_PATH_ERROR  = 5;
    const ErrorCode ENTRY_IO_ERROR    = 6;
    const ErrorCode DIR_NOT_EMPTY     = 7;

    const ErrorCode TRANSFER_IO_ERROR = 8;
    const ErrorCode TRANSFER_ABORT    = 9;


    exception FileSystemError {
        ErrorCode error;
        wstring desc;
    };

    // Error transferring between two files

    exception TransferError {
        TransferEndPointRole error_endpoint;
        ErrorCode error;
        wstring desc;
    };
```

ErrorCode

The exceptions defined in the **CosFileTransfer** module contain an **ErrorCode** field which identifies the category of the error. The values are:

- `UNSPECIFIED` - The error category is none of the below.
- `UNAVAILABLE` - The **FileSystem** is temporarily unavailable. This is only raised by the **FileSystem::login** method.
- `UNSUPPORTED` - The operation or the particular parameter values are unsupported by the implementation.
- `NO_PERMISSION` - The user credentials are insufficient or invalid for the requested operation.
- `ENTRY_PATH_ERROR` - A component of the name specified for a **File** or **Directory** is invalid or the entry does not exist.
- `ENTRY_EXISTS` - The operation expected the entry not to already exist.
- `ENTRY_IO_ERROR` - There has been an error opening, reading, writing, or closing a **File** or **Directory**.

- DIR_NOT_EMPTY -The implementation does not allow removal of a **Directory** that is not empty.
- TRANSFER_IO_ERROR - There has been an opening, reading, writing, or closing a data transfer connection.
- TRANSFER_ABORT - A file transfer operation has been aborted.

Client ErrorCode Handling

In this chapter, each operation description lists the exceptions raised along with specific **ErrorCode** values. A service implementation may use **ErrorCode** values other than those specifically listed. A client must handle these values gracefully, at the very least handling them like **UNSPECIFIED**.

FileSystemError

This exception is raised when an operation involving a single **CosFileTransfer** object fails. The fields are:

- error - A broad classification of the error.
- desc - Optional text detail about the error.

TransferError

TransferError is raised by operations that involve copying one **File**'s contents to another. Since there are two **Files** involved, the one that raised the exception must be identified. The fields are:

- error_endpoint - Identifies whether the exception originated from the source or sink of the data transfer.
- error - A broad classification of the error.
- desc - Optional text detail about the error.

3.3.2 FileSystem Interface

The **FileSystem** interface provides access to the virtual file system represented by the service. The IDL is:

```
interface FileSystem {

    FileSession login(in wstring user,
                      in wstring password,
                      in CosPropertyService::Properties login_properties,
                      out Directory initial_dir)
        raises(FileSystemError);

    wstring get_system_id();
};
```

login

Before transferring files or performing maintenance operations, a client must provide credentials to login to the **FileSystem** to obtain an initial **Directory** reference. The **FileSystem** validates the user credentials in an implementation specific manner.

Parameters

- `user` - **FileSystem** specific text string identifying the user.
- `password` - **FileSystem** specific text string identifying the user password.
- `login_details` - sequence of **FileSystem** specific properties providing login details. A **FileSystem** implementation may use any property names and values that are appropriate. The following properties with **wstring** values are defined:
  - `user` - Same value as the user parameter. If this property is present, the **user** parameter is ignored.
  - `password` - Same value as the password parameter. If this property is present, the **password** parameter is ignored.
  - `account` - Many systems have the concept of an account in addition to a user.
- `initial_dir` - returns the initial **Directory** for the supplied login details.

Return value

This method returns a **FileSession** (see section 3.1.3) for the supplied login parameters.

Exceptions

FileSystemError. The following **ErrorCode** values are defined:

- `UNAVAILABLE` - The **FileSystem** is unavailable for login. In this case, no attempt has been made to validate the user credentials. A retry by the client may be successful.
- `NO_PERMISSION` - The supplied user credentials were rejected.

get_system_id

Returns implementation specific text providing identification of the file system. This text shall be suitable for display to an end user.

Return value

Returns a **wstring** identifying the file system. This string is for informational purposes only and cannot be used to determine object identity. An implementation is not required to make this string globally unique. An empty string is a legal return value.

3.3.3 FileSession Interface

The **FileSession** interface controls the lifecycle of all object references obtained from the server. The IDL is:

```
interface FileSession {
    void destroy();
};
```

destroy

The **destroy** operation terminates the session with the service established by the call to **FileSystem::login**. All objects associated with the **FileSession** such as **Directories**, **Files**, etc. are destroyed. After the **destroy** method is invoked, further operations on the **FileSession** or any of its associated objects will raise an **OBJECT_NOT_EXIST**.

The status of any file transfers that are in progress at the time of a call to **destroy** are undefined.

3.3.4 FileSystemEntry Interface

FileSystemEntry is a base interface that defines operations that are common to the **Directory** (Section 3.1.5) and **File** (Section 3.1.7) interfaces.

Properties

The interface derives from **CosProperty::PropertySet**. The following properties are defined:

Table 3-1 FileSystemEntry Properties

Property Name	Data Type	Property Mode	Description
name	**EntryName**	mandatory, **fixed_readonly**	Simple name relative to parent **Directory**
path	**EntryPath**	optional, **fixed_readonly**	Full pathname relative to initial **FileSession Directory**.
owner	**wstring**	optional, **fixed_readonly**	If defined, the owner of the **Entry**.
creation_time	**TimeBase::UtcT**	optional, **fixed_readonly**	If defined, the entry creation time.
modification_time	**TimeBase::UtcT**	optional, **fixed_readonly**	If defined, the last time the entry was modified.

A mandatory property is one that a service implementation must always allow a client to access. An optional property is one that a service implementation may restrict a client's access to, may not provide a value for a particular **File** or **Directory**, or not provide at all. For purposes of discussion, the properties from the above list and any other implementation defined properties that a specific client is allowed access to are called *client accessible* properties.

The behavior of the **CosProperties::PropertySet** methods specific to **FileSystemEntry** objects are:

define_property

For a read only *client accessible* property, a **CosProperties::ReadOnlyProperty** exception will be raised. If the property is not client accessible, a **CosProperties::UnsupportedProperty** is raised.

define_properties

An implementation will behave as for **define_property**, except that the exception raised is **CosProperties::MultipleExceptions** containing **PropertyException** structs having a **reason** codes of **read_only_property** or **unsupported_property**.

get_number_of_properties

An implementation must not include any non client accessible properties in the return count. The returned count may be less than the total number of properties associated with the **FileSystemEntry**.

get_all_property_names

An implementation must not include any non client accessible properties in the returned sequence. The returned sequence size may be less than the total number of properties associated with the **FileSystemEntry**.

get_property_value

For all client accessible properties that a value is defined for, the property value is returned. Otherwise the exception **PropertyNotFound** is raised.

get_properties, get_all_properties

For all client accessible properties that a value is defined for, the property is returned. All other properties will denote an exception by appearing in the return sequence with a type of **tk_void** as described in the CosProperty Service specification.

delete_property, delete_properties, delete_all_properties

For all fixed client accessible properties, an exception denoting **fixed_property** shall be raised. For **delete_all_properties**, client accessible fixed properties will not be deleted and the operation shall return true.

FileSystemEntry Methods

The next sections describe the methods available on the **FileSystemEntry** interface.

get_name

Returns the simple name for this **FileSystemEntry**. This is the same value returned by the `name` property.

Return Value

EntryName for the **FileSystemEntry.**

get_path

Returns the `path` name for this **FileSystemEntry** relative to the initial **Directory** returned from **FileSystem::login**. This is the same value returned by the `path` property.

Return Value

EntryPath for the **FileSystemEntry.**

Exceptions

A **FileSystemError** may be raised for an implementation defined reason. No specific **ErrorCode** values are defined.

exists

Report the existence of a **FileSystemEntry** on the **FileSystem**.

Return Value

- `true` - The **FileSystemEntry** exists on the **FileSystem**.
- `false` - The **FileSystemEntry** does not exist on the **FileSystem**.

Exceptions

A **FileSystemError** may be raised for an implementation defined reason. No specific **ErrorCode** values are defined.

get_parent

Returns the `parent` **Directory** for this **FileSystemEntry.**

Exceptions

A **FileSystemError** may be raised with an ErrorCode value of:

- `NO_PERMISSION` - If the client is not allowed to access the parent **Directory**. Many implementations will raise this exception if **get_parent** is called on the initial **Directory** returned from **FileSystem::login**.

get_session

Returns the `associated` **FileSession** for this **FileSystemEntry.**

Exceptions

A **FileSystemError** may be raised with an ErrorCode value of:

- `NO_PERMISSION` - If the client is not allowed to access the **FileSession** from this **FileEntry**.

remove

This operation removes the entry from the service. A **Directory** may only be removed if it is empty. Once removed an **Entry** will not appear in a listing of its parent directory.

Exceptions

A **FileSystemError** is raised on error. The following **ErrorCode** values are defined:

- `NO_PERMISSION` - If the client is not allowed to `remove` this **Entry**.
- `DIR_NOT_EMPTY` - If this is a **Directory** and contains child entries.
- `ENTRY_PATH_ERROR` - If the **Entry** does not exist.

destroy

This operation releases the **FileSystemEntry** object. It does not **remove** the entry's representation from the **FileSystem**. A client should call **destroy** on an **Entry** when it has finished with it.

3.3.5 Directory Interface

The **Directory** interface represents a collection of **File** and **Directory** entries. The interface defines operations to list and obtain references to these entries. The IDL is:

```
interface Directory: FileSystemEntry {

    DirEntryIterator list(in CosPropertyService::PropertyNames listProps)
        raises (FileSystemError);

    Directory create_directory(in EntryPath fpath)
        raises( FileSystemError);

    File get_file(in EntryPath fpath, in boolean must_exist)
        raises( FileSystemError);

    Directory get_directory(in EntryPath fpath)
        raises( FileSystemError);

    void remove_entry(in EntryPath fpath)
        raises( FileSystemError);
};
```

Directory Properties

In addition to the properties for **FileSystemEntry**, **Directory** objects have one additional property listed in the table below.

Table 3-2 Directory Properties

Property Name	Data Type	Property Mode	Description
num_children	**DirEntryCount**	optional, fixed_readonly	The number of entries in the Directory. In some cases it is not practical to provide this value directly. In this case the directory must be iterated through to count the entries.

list

The `list` operation allows a client to iterate through a set of **Directory** entries and their properties.

Parameters

- `list-props` - A sequence containing the names of the desired entry properties. A service implementation is not required to return all the properties requested.

Return value

A **DirEntryIterator** (see Section 3.1.6). If the **DirEntryIterator** value is `nil`, there were no entries to return. If the value is `non-nil` there may or may not be entries to be retrieved.

An implementation is not required to return sequence members that represent the current or parent **Directory** entries.

The properties returned are dependent on client permissions and whether an entry has a value for the property. If a client does not have permission to retrieve a property, an implementation must not raise an exception with an **ErrorCode** of **NO_PERMISSION**. The denied property shall be silently omitted.

Exceptions

FileSystemError. The following **ErrorCode** value is defined:

- NO_PERMISSION - The client is not permitted to obtain the **Directory** list.

create_directory

This operation creates a child **Directory**. It is similar to the familiar `mkdir` command.

Parameters

- dir_path - The `Path` of the **Directory** to create. This **EntryPath** is relative to the `Directory`. If dir_path contains more than one component, the intermediate directories will be created as well.

Return value

The newly created **Directory**.

Exceptions

A **FileSystemError** may be raised with following **ErrorCode** values:

- `ENTRY_PATH_ERROR`. If any component of the path is invalid or one of the intermediate components is a **File**.
- `NO_PERMISSION` - If the client is not allowed to create or access any component of the dir_path.
- ENTRY_EXISTS - If this **Directory** already exists.

get_file

This operation returns a `File` for the specified `Path`.

Parameters

- file_path - The `File's Path` relative to the `Directory`.
- must_exist -if `true`, the operation will only succeed if the file already exists on the **FileSystem**.

Return value

A **File** reference for the file.

Exceptions

A **FileSystemError** may be raised with following **ErrorCode** values:

- `ENTRY_PATH_ERROR` - If any component of the path is invalid or one of the intermediate components is a **File**. If the **must_exist** parameter is `true` and the file does not exist.
- `NO_PERMISSION` - If the client is not allowed to access any component of the file_path.

get_directory

This operation returns a **Directory** corresponding to an existing directory.

Parameters

- `dir_path` - The relative **EntryPath** for the **Directory**.

Return value

The requested **Directory**.

Exceptions

A **FileSystemError** may be raised with following **ErrorCode** values:

- ENTRY_PATH_ERROR. If any component of the path is invalid or one of the intermediate components is a **File**, or the **Directory** does not exist.
- ~~an IllegalOperationException is thrown by **set_directory()** when~~ NO_PERMISSION - If the ~~user~~ client is not ~~permitted~~ allowed to access any component of the ~~destination directory at the target location~~dir_path.

~~While moving from one directory to another within the remote server a client may wish to perform typical operations on a file. The specification offers two alternatives for obtaining a reference to a specific **File**. If the client has a reference to the **File**'s parent **directory**, they can obtain a reference to the **File** by invoking the **Directory** interface's **list()** operation. The **get_file()** operation is also provided by the **FileTransferSession** interface to enable a client to reference a specified file at a specific location if the absolute pathname of the file is known. The **complete_file_name** parameter represents a sequence of strings that identify the file and its location. Therefore, the **get_file()** operation is independent of the working directory from where it is invoked. In both cases a **File** reference will be returned within a **FileWrapper** struct that also contains an enumeration to indicate whether the file is a directory or not.~~

~~The **get_file()** operation can throw a number of exceptions:~~

- ~~a SessionException is thrown by **get_file()** when the existing connection with the remote server is not available when attempting to reference a remote file.~~
- ~~a FileNotFoundException is thrown by **get_file()** when the target file cannot be found on the remote server.~~
- ~~a RequestFailureException is thrown by **get_file()** when it is not possible to retrieve a reference to the target file due an internal problem at the gateway.~~

remove_entry

This operation removes a **File** or **Directory** entry. If the entry is a **Directory**, it must be empty before it can be removed.

Parameters

- entry_path - The relative **EntryPath**.

Exceptions

A **FileSystemError** may be raised with following **ErrorCode** values:

- ENTRY_PATH_ERROR - If any component of the path is invalid or one of the intermediate components is a **File**.

- an IllegalOperationException is thrown by **get_file()** when NO_PERMISSION - If the user client is not permitted allowed to access the target file or access the location any component of the target filepath.

A client can also delete a file at the remote server by invoking the **delete()** operation specifying the file to delete with the **File** parameter. **delete()** may also be used by a client to delete a directory at the remote system since the **Directory** interface is a specialization of the **File** interface.

void delete(in File file)
raises(SessionException, FileNotFoundException,
RequestFailureException, IllegalOperationException);

delete() can throw a number of exceptions:

- a SessionException is thrown by **delete()** when the existing connection with the remote server is not available when attempting to delete a remote file or directory.
- a FileNotFoundException is thrown by **delete()** when the file or directory to be deleted cannot be found on the remote server.
- a RequestFailureException is thrown by **delete()** when it is not possible to delete the target file or directory due an internal problem at the gateway.
- an IllegalOperationException is thrown by **delete()** when the user is not permitted to delete the target file or directory or access the location of the target file or directory.

Finally, the **FileTransferSession** interface provides the **logout()** operation to enable a client to terminate their active session.

void logout ();

3.4 File

The **File** interface exposes all features that are traditionally associated with a file to the client and abstracts the complexity of performing protocol specific operations by providing a proxy for a physical file on the remote file serving mechanism. The **File** interface inherits from the **CosPropertyService::PropertySetDef** interface. As a consequence the **File** interface should implement the **CosPropertyService::PropertySetDef** operations to allow a client to get and/or set attribute values. Additionally, the interface defines three attributes that are always associated with a file: the file's name, the absolute pathname of the file, and a reference to its parent. Both the **name** and **complete_file_name** attributes are defined as readonly. A File Transfer Client can change the name of a file or its complete file name by accessing the associated properties (see Table 3-1) if they are supported by the implementation. A **FileTransferSession** attribute, **associated_session**, is also specified since it is ubiquitous for all **File** references and is essential for implementations to indicate the **FileTransferSession** that a **File** reference is associated with.

```
interface File:CosPropertyService::PropertySetDef {
readonly attribute string name;
readonly attribute FileNameList complete_file_name;
readonly attribute Directory parent;
readonly attribute FileTransferSession
associated_session;
};
```

3.4.1 DirEntryIterator Interface

The **DirEntryIterator** interface is used to iterate through the results of a **Directory::list** operation. The IDL is:

```
  // Directory listing size and list offset

  typedef unsigned long long DirEntryCount;
  typedef unsigned long long DirEntryOffset;

  // Directory listing Types

  typedef short DirEntryType;
  const DirEntryType FILE_ENTRY = 0;
  const DirEntryType DIR_ENTRY  = 1;

  struct DirEntry {
    EntryName name;
    DirEntryType type;
    CosPropertyService::Properties props;
  };

  typedef sequence<DirEntry> DirEntrySeq;

  interface DirEntryIterator {
    DirEntrySeq next(in DirEntryOffset from_dir_entry,
                     in DirEntryCount max_dir_entries)
      raises (FileSystemError);
    void destroy();
  };
```

Related Types

DirEntryType

This type defines the type of an entry, either **DIR_ENTRY**, or **DIR_FILE**.

DirEntry

Directory::list returns FileSystemEntry information in DirEntry structures. The fields of this struct are:

- `name` - The simple (single component) name of the entry in this **Directory**.
- `type` - The **DirEntryType** of the entry.
- `props` - A sequence containing the requested entry properties.

DirEntrySeq represents a sequence of **DirEntry**.

DirEntryCount, DirEntryOffset

These types are used to control the iteration through a **Directory**.

- `DirEntryCount` - The maximum number of entries to return to the client.
- `DirEntryOffset` - The offset into the **Directory's** entry list from which the **DirEntryCount** applies.

See the section "next" below for details on the use of these types.

next

This operation returns a sequence of **DirEntry**. The **DirEntryIterator** is a recoverable iterator and allows a client to repeat a failed call to **next**, requesting a smaller sequence in the event of an exception.

Parameters

- `from_entry_number` - return entries starting from the specified entry number.
- `max_dir_entries` - The maximum number of entries to return to the client. If the value is zero value, there is no upper bound.

In normal operation **next** is called repeatedly until all the directory entries are returned. The first time **next** is called, **from_entry_number** must be zero. For subsequent calls, the value of **from_entry_number** is set to its previous value plus the length of the returned entry sequence.

In the event that a call to **next** results in an exception indicative of resource exhaustion on either the client or the server, such as **NO_MEMORY**, the client can retry the `next` operation by invoking `next` with the previous **from_entry_number** and a smaller **max_dir_entries** value.

If the **next** operation fails with a **max_dir_entries** value of one, the iteration cannot be completed and the client must handle the error.

Return value

A **DirEntrySeq** with a length of up to **max_dir_entries** for non-zero values of **max_dir_entries**. If **max_dir_entries** is zero, the returned sequence length is implementation defined. In either case, an implementation may not return a **DirEntrySeq** of length zero unless there are no further entries to retrieve.

Exceptions

A **FileSystemError** may be raised with following **ErrorCode** value:

- UNSUPPORTED. If the **from_entry_number** parameter is illegal for the current iterator state.

destroy

After a client is finished with a **DirEntryIterator**, **destroy** should be called to release the internal resources held by the service implementation.

3.4.2 File Interface

The IDL is:

```
interface File: FileSystemEntry {

    void copy(in File dest)
        raises( TransferError);

    void append(in File dest)
        raises( TransferError);

    void insert(in File dest, in FileOffset offset)
        raises( TransferError);

    TransferEndPoint create_end_point(in TransferEndPointRole ep_role,
                                      in FilePos seek,
                                      in FileOffset offset,
                                      in TransferProtocol ep_protocol)
        raises (FileSystemError);

    TransferProtocolSeq get_end_point_protocols();
};
```

File Properties

In addition to the ~~attributes defined~~ properties for ~~the~~ **FileSystemEntry**, **File** ~~interface there are a number of file properties identified~~ objects have one additional property listed in the ~~following table~~table below.

Table 3-3 List of properties that can be associated with a file

Property Name	Data Type	Description
is_directory	**boolean**	Indicates whether the file represents a remote directory.
creator	**string**	Indicates the name of the user that created the file.

Table 3-3 List of properties that can be associated with a file

size	**unsigned long**	Indicates the size of the file in bytes.
modification_time	**string**	Indicates the time and date on which the file was last modified
creation_time	**string**	Indicates the time and date on which the file was created.
access_rights	**AccessLevel**	Indicates operations associated with a file that are available to a user.
name	**string**	Indicates the name of the file
complete_file_name	**FileNameList**	Indicates the absolute name of the file
num_children	**long**	If the file is a directory this property indicates the number of files associated with that directory.

3.5 Directory

The **Directory** interface exposes all features that are traditionally associated with a directory to the client and abstracts the complexity of performing protocol-specific operations by providing a proxy for a physical directory on the remote file serving mechanism. In the same manner that a directory on the file system is considered to be a specialization of a file, the **Directory** interface defined in the framework is a specialization of the **File** interface.

```
interface Directory : File {
void list(in unsigned long how_many,
 out FileList fl,
 out FileIterator fi);
};
```

Each **Directory** provides a **list()** operation that enables a client to access a designated number of files associated with the directory. Parameters of the list operation include:

- **how_many**: used to specify the number of **File** references that are initially returned within the **FileList** parameter.
- **fl**: A sequence of **FileWrapper** references associated with the directory.
- **fi**: A reference to the **FileIterator** interface used to return subsequent **File** references that were not returned in the initial **FileList** parameter

Table 3-4 File Properties

Property Name	Data Type	Property Mode	Description
size	**FileSize**	Optional, **fixed_readonly**	The size of the file in octets. In some implementations it may not be practical to determine the size of an entity being represented by a File. In this case the property is not provided.

copy

The **copy** operation copies the contents of this **File** to the destination **File**. If the destination **File** currently exists it is overwritten.

Parameters

- `dest` - The destination (sink) **File**.

Exceptions

A **TransferError** may be raised with following **ErrorCode** values:

- `ENTRY_PATH_ERROR`. If any component of a File is invalid or one of the intermediate components is a **File**.
- `NO_PERMISSION` - If the client cannot access any component of a file path
- `ENTRY_IO_ERROR` - There was an error in opening, closing, reading, or writing a file.
- `TRANSFER_IO_ERROR` - There was an error in opening, closing, reading, or writing a data connection.
- `TRANSFER_ABORT` - The transfer was aborted.

append

The `append` operation appends the contents of this **File** to the destination **File**.

Parameters

- dest - The destination **File**.

Exceptions

A **TransferError** may be raised with following **ErrorCode** values:

- ENTRY_PATH_ERROR. If the sink **File** does not exist. If any component of a **File** is invalid or one of the intermediate components is a **File**.
- UNSUPPORTED - If the sink **File** does not allow an append.
- NO_PERMISSION - If the client cannot access any component of a file path
- ENTRY_IO_ERROR - There was an error in opening, closing, reading, or writing a file.
- TRANSFER_IO_ERROR - There was an error in opening, closing, reading, or writing a data connection.
- TRANSFER_ABORT - The transfer was aborted.

insert

The insert operation inserts the contents of the **File** at the specified offset in the destination **File**.

Parameters

- dest - The destination **File**.
- file_offset - The **FileOffset** into the destination **File**.

Exceptions

A **TransferError** may be raised with following **ErrorCode** values:

- ENTRY_PATH_ERROR. If the sink **File** does not exist. If any component of a **File** path is invalid or one of the intermediate components is a **File**.
- UNSUPPORTED - If the sink **File** does not allow an insert.
- NO_PERMISSION - If the client cannot access any component of a file path
- ENTRY_IO_ERROR - There was an error in opening, closing, reading, or writing a file or the file_offset parameter is larger than the sink **File** size.
- TRANSFER_IO_ERROR - There was an error in opening, closing, reading, or writing a data connection.
- TRANSFER_ABORT - The transfer was aborted.

create_end_point

The **create_end_point** method is used to create a **TransferEndPoint** (see section 3.1.8), which is used by a service to implement the high level **copy**, **append**, and **insert** operations. Clients performing more complex transfer operations may also make use of this method.

Parameters

- `ep_role` - Specifies whether the role of the **TransferEndPoint** is to read or write the `File`'s contents. Values are **TransferEndPointRole::SOURCE**, **TransferEndPointRole::SINK, and TransferEndPointRole::SINK_INSERT**. **TransferEndPointRole::SINK** will overwrite and truncate to the last written octet.
- `file_pos` - Specifies whether the data transfer will be relative to the beginning or end of the **File**. Values are **FilePos::BEGIN** and **FilePos::END.**
- offset - The offset from the **file_pos** to begin reading or writing.
- `ep_protocol` - Specifies the type of **TransferEndPoint** to be created. The specification currently defines transfer protocols using `corba` interfaces, `ftp`, and `ftam`. See section 3.1.8 for details.

Return value

TransferEndPoint for use in a single transfer of the `File`. The **TransferEndPoint** should be destroyed after use.

Exceptions

A **TransferError** may be raised with following **ErrorCode** values:

- `ENTRY_PATH_ERROR` - If the **SOURCE** file does not exist. If any component of a **File** path is invalid or one of the intermediate components is a **File**.
- `UNSUPPORTED` - If an unsupported **ep_protocol** is specified.
- `NO_PERMISSION` - If the client cannot create the **TransferEndPoint**.
- `ENTRY_IO_ERROR` - There was an error in opening, closing, reading, or writing a file.
- `TRANSFER_IO_ERROR` - There was an error in opening, closing, reading, or writing a data connection.

get_end_point_protocols

Obtains a sequence of supported transfer protocols for this **File**. An implementation is not required to provide the same transfer protocols for all **Files**. An implementation may also change the set of available transfer protocols for a **File** if there are no **TransferEndPoints** for that **File** in existence at the time of the change.

Return value

TransferProtocolSeq listing supported protocols. The sequence is in preferred protocol order.

An implementation is not required to return the corba interface "IDL:omg.org/CosFileTransfer/OctetTransferIterator:1.0" since it is mandatory. An implementation may choose to return it in the list to indicate a preference over other protocols.

3.5.1 TransferEndPoint Interface

TransferEndPoint objects represent a **File** during a transfer operation. The IDL is:

```
    interface TransferEndPoint;
    typedef wstring TransferProtocol;
    typedef sequence<TransferProtocol> TransferProtocolSeq;

    typedef short TransferEndPointRole;

    const TransferEndPointRole SOURCE = 0;
    const TransferEndPointRole SINK = 1;
    const TransferEndPointRole SINK_INSERT = 2;

    // transfer protocol specific information

    typedef wstring TransferDetail;

    typedef short TransferState;
    const TransferState CREATE  =  0;
    const TransferState LISTEN  =  1;
    const TransferState CONNECT =  2;
    const TransferState ACTIVE  =  3;
    const TransferState COMPLETE = 4;
    const TransferState ABORT =    5;

    struct TransferStatus {
        TransferState state;     // current transfer state
        FileCount current_count; // current transfer count
        FileCount max_count;     // expected transfer size bytes/chars
    };

    interface TransferEndPoint
    {
        TransferDetail go_to_listen()
            raises(FileSystemError);

        TransferDetail connect_to_peer(in TransferDetail passive_detail)
            raises(FileSystemError);

        void set_peer(in TransferDetail active_detail)
            raises(FileSystemError);

        TransferStatus get_transfer_status()
            raises (FileSystemError);

        void transfer()
            raises (FileSystemError);

        void abort()
            raises (FileSystemError);

        void destroy();
    };
```

Related Types

TransferProtocol

A string type that identifies a transfer protocol such as "`ftp`". **TransferProtocolSeq** is the sequence typedef for **TransferProtocol**.

TransferDetail

This is a string type with a format that is specific to the transfer protocol used. During connection negotiation, **TransferEndPoints** exchange protocol information in **TransferDetails**.

TransferState

An enumeration that provides state information about a **TransferEndPoint**. The defined states are:

- `CREATE` - Initial state after creation.
- `LISTEN` - waiting for an active connection, **go_to_listen** has been called.
- `CONNECT` - connected to its peer, either **connect_to_peer**, or **set_peer** has been called.
- `ACTIVE` - data transfer has started.
- `COMPLETE` - data transfer completed successfully.
- `ABORT` - data transfer error

TransferStatus

This struct provides information about the progress of a transfer that a **TransferEndPoint** is involved in. The fields are:

- `state` - the **TransferState** for the endpoint.
- `current_count` - expected transfer size. If this is unknown or not provided by the service implementation, it is set to zero. This value is usually available from the source endpoint but not the sink.
- `max_count` - For a source endpoint this is the octets sent. For a sink endpoint this is the octets received. In the case of a transfer error this value represents the transfer count before the abort. If the value is unknown or not provided by the service implementation it is set to zero.

go_to_listen

This method is called on the passive **TransferEndPoint** to establish the listening side of a data connection. On return the **TransferEndPoint** is ready to accept an active connection. This is the first step in negotiating a transfer connection.

Return value

TransferDetail describing the passive **TransferEndPoint** details. For example in the case of a `corba` protocol transfer, the returned **TransferDetail** would be an `IOR` string, and for an `ftp` transfer, "`host:port`".

Exceptions

A **TransferError** may be raised with following **ErrorCode** values:

- `ENTRY_PATH_ERROR` - If a file does not exist, any component of a **File** path is invalid or one of the intermediate components is a **File**.
- `UNSUPPORTED` - If an invalid **active_detail** is specified for those protocols that use this parameter or this method is called on an active **TransferEndPoint**.
- `NO_PERMISSION` - If the client does not have the proper credentials to perform the operation.
- `ENTRY_IO_ERROR` - There was an error in opening, closing, reading, or writing the file associated with the **TransferEndPoint**.
- `TRANSFER_IO_ERROR` - There was an error in opening, closing, reading, or writing the data connection.

connect_to_peer

This method is called on an active **TransferEndPoint** to make the connection to the passive **TransferEndPoint**. This is the second step in negotiating a transfer connection.

Parameters

- passive_detail - This **TransferDetail** provides the required details to allow the active **TransferEndPoint** to connect to the passive **TransferEndPoint**. This parameter is set to the return value from the **go_to_listen** call on the passive **TransferEndPoint**.

Return value

TransferDetail describing the active **TransferEndPoint** details.

Exceptions

A **TransferError** may be raised with following **ErrorCode** values:

- `ENTRY_PATH_ERROR`. If a file does not exist. If any component of a **File** path is invalid or one of the intermediate components is a **File**.
- `UNSUPPORTED` - If an invalid **passive_detail** is specified for those protocols that use this parameter or this method is called on an active **TransferEndPoint**.
- `NO_PERMISSION` - If the client does not have the proper credentials to perform the operation.

- ENTRY_IO_ERROR - There was an error in opening, closing, reading, or writing the file associated with the **TransferEndPoint**.
- TRANSFER_IO_ERROR - There was an error in opening, closing, reading, or writing the data connection.

set_peer

This method is called on the passive **TransferEndPoint** to complete the transfer connection negotiation. It is the final step in negotiating a transfer connection. It allows the passive **TransferEndPoint** to obtain any remaining **TransferDetail** about the active end of the connection. The use of this information is protocol dependent.

Parameters

- **active_detail** - This **TransferDetail** provides information about the active end of the data connection to the passive **TransferEndPoint**. The value of this parameter is set to the result of the **connect_to_peer** operation.

Exceptions

A **TransferError** may be raised with following **ErrorCode** values:

- ENTRY_PATH_ERROR. If a file does not exist. If any component of a **File** path is invalid or one of the intermediate components is a **File**.
- UNSUPPORTED - If an invalid **active_detail** is specified for those protocols that use this parameter or this method is called on an active **TransferEndPoint**.
- NO_PERMISSION - If the client does not have the proper credentials to perform the operation.
- ENTRY_IO_ERROR - There was an error in opening, closing, reading, or writing the file associated with the **TransferEndPoint**.
- TRANSFER_IO_ERROR - There was an error in opening, closing, reading, or writing the data connection.

get_transfer_status

This method returns the status of the **TransferEndPoint**.

Exceptions

A **FileSystemError** may be raised. The following specific **ErrorCode** value is defined.

- UNSUPPORTED - If a service implementation does not provide this information.

transfer

Transfer the **File** contents between the source and sink **TransferEndPoints**. This method is called on the source TransferEndPoint.

Exceptions

A **FileSystemError** may be raised. The following specific **ErrorCode** value is defined.

- `UNSUPPORTED` - If this operation is called on a sink **TransferEndPoint**.

abort

This method causes the **TransferEndPoint** to terminate the current **transfer** operation the transfer at its end of the connection. The other **TransferEndPoint** will see the abort an unexpected termination of the transfer operation or connection.

An implementation may not be able to abort a transfer or even respond to the request until the current transfer is complete.

Exceptions

A **FileSystemError** may be raised. The following specific **ErrorCode** values is defined.

- `UNSUPPORTED` - If it is not possible to abort the transfer operation.

The system exception **BAD_INV_ORDER** will be raised if **abort** is called on a transfer that has not yet started, is already completed, or has aborted.

destroy

This method closes a transfer, releasing any internal resources the **TransferEndPoint** has obtained. Further invocations on this object will receive an **OBJECT_NOT_EXIST** exception.

3.5.2 OctetTransferIterator Interface

The **OctetTransferIterator** interface allows for transfer of a **File's** contents using only CORBA calls and without requiring another **File** object to transfer to or from. **OctetTransferIterator** is a recoverable iterator. It does not provide random access to a **File's** contents.

The IDL is:

```
typedef unsigned long long FileLength;
typedef unsigned long long FileOffset;
typedef unsigned long long FileCount;
typedef sequence<octet> FileOctetSeq;
```

```
interface OctetTransferIterator {

    FileOctetSeq get_octet_seq(in FileOffset from_octet, in FileCount
max_octets)
        raises (FileSystemError);

    void put_octet_seq(in FileOffset to_octet, in FileOctetSeq octetSeq)
        raises(FileSystemError);

    void destroy()
        raises(FileSystemError);

};
```

Related Types

FileOffset

This type represents an offset into a **File's** contents. Normally an **OctetTransferIterator** is created by a **TransferEndPoint**, in which case an **OctetTransferIterator's FileOffset** values are relative to the **FileOffset** specified when the **TransferEndPoint** was created (**File::create_end_point**).

FileCount

This type represents a **File** octet count. It is used to represent **File** size and the number of octets transferred.

FileOctetSeq

An octet sequence representing the binary contents of a **File**.

get_octet_seq

This operation returns the next unread sequence of **File** octets.

Parameters

- `from_octet` - return octets starting from the specified offset.
- `max_octets` - The maximum number of octets to return. If the value is zero, there is no upper bound.

In normal operation **get_octet_seq** is called repeatedly until all **File** octets are returned. The first time **get_octet_seq** is called, **from_octet** is set to zero. For subsequent calls, the value of **from_octet** is set to its previous value plus the length of the returned sequence of **File** octets.

If **get_octet_seq** raises an exception that may be indicative of resource exhaustion on either the client or server such as **NO_MEMORY**, the client can retry the failed read by invoking **get_octet_seq** with the previous **from_octet** and a smaller **max_octets**.

If **get_octet_seq** fails with a **max_octets** value of one, the get iteration cannot be completed and the client must handle the error.

Exceptions

A **FileSystemError** may be raised with following **ErrorCode** values:

- `ENTRY_PATH_ERROR`. If a file does not exist. If any component of a **File** path is invalid or one of the intermediate components is a **File**.
- `UNSUPPORTED` - If this **TransferOctetIterator** does not allow reads.
- `NO_PERMISSION` - If the client does not have the proper credentials to perform the operation.
- `ENTRY_IO_ERROR` - There was an error in opening, closing, reading, or writing the file.
- TRANSFER_ABORT - An associated **TransferEndPoint** has been aborted.

put_octet_seq

This operation writes an octet sequence to a **File**.

Parameters

- `octet_offset` - write octets starting at the specified offset.
- `octet_seq` - The octet sequence to write.

In normal operation **put_octet_seq** is called repeatedly until all `the` **File** octets are transferred. The first time **get_octet_seq** is called, **from_octet** is set to zero. For subsequent calls, the value of **octet_offset** is set to its previous value plus the length of the previous **octet_seq**.

If **put_octet_seq** raises an exception indicative of resource exhaustion on either the client or server such as **NO_MEMORY**, the client can retry the operation by invoking **put_octet_seq** with the previous **octet_offset** and a smaller **octet_seq**.

If **put_octet_seq** fails with a **octet_seq** length of one, the put iteration cannot be completed and the client must handle the error.

Exceptions

A **FileSystemError** may be raised with following **ErrorCode** values:

- `ENTRY_PATH_ERROR`. If a file does not exist. If any component of a **File** path is invalid or one of the intermediate components is a **File**.

- UNSUPPORTED - If the TransferOctetIterator does not allow writes.
- NO_PERMISSION - If the client does not have the proper credentials to perform the operation.
- ENTRY_IO_ERROR - There was an error in opening, closing, reading, or writing the file.
- TRANSFER_ABORT - An associated **TransferEndPoint** has been aborted.

destroy

After a client is finished with an **OctetTransferIterator**, **destroy** must be called to complete the transfer and gracefully release any associated resources held by the service implementation. Further calls to the iterator will raise an **OBJECT_NOT_EXIST**.

Exceptions

A **FileSystemError** may be raised with following **ErrorCode** values:

- ENTRY_PATH_ERROR. If a file does not exist. If any component of a **File** path is invalid or one of the intermediate components is a **File**.
- ENTRY_IO_ERROR - There was an error in opening, closing, reading, or writing the file.

If **destroy** raises a **FileSystemError**, the **OctetTransferIterator** is still destroyed.

3.6 Object Lifecycle

All of the interfaces except for **FileSystem** have a **destroy** operation. After the **destroy** method is invoked, any further operations on the object reference will raise an **OBJECT_NOT_EXIST**.

A client should invoke **destroy** on an object after use is complete to allow a service implementation to reclaim resources. An implementation is free to reap objects at any time in order to reclaim resources.

Clients should expect that any operation on a **CosFileTransfer** object may raise an **OBJECT_NOT_EXIST** as a server may reclaim an object, particularly if inactive, at anytime.

3.7 Conformance Criteria

3.7.1 Interfaces

A service implementation must provide all of the interfaces defined in this specification. An implementation is not required to support the following operations on all **Files** or **TransferEndPoints**:

- **File::append**
- **File::insert**
- **TransferEndPoint::abort**
- **TransferEndPoint::get_transfer_status**

If an implementation does not support these operations on a given object it must raise a **FileSystemError** exception with an **ErrorCode** value of **UNSUPPORTED.**

3.7.2 Transfer Protocols

A service implementation must support transfers using the corba interface "IDL:omg.org/CosFileTransfer/OctetTransferIterator:1.0". All other protocols are optional.

Example Scenarios *4*

Note – This entire chapter has been deleted. It describes the old IDL interface.

4.1 Introduction

The purpose of this chapter is to clearly illustrate the interactions between an end user and the components within the framework during a number of scenarios that are typical of any file transfer mechanism. It is important to note that these scenarios represent one of many possible implementations of the proposed framework. Each of the scenarios presented are divided into three sections:

1. A general description of how the scenario is enabled by the CORBA interfaces defined by the framework.
2. A code sample is provided in the Java™ programming language to further demonstrate the application of the interfaces.
3. An interaction diagram describing component interactions during each scenario for one possible implementation of the framework.

Figure 4-1 introduces each of the scenarios using a USE case approach illustrated in the Unified Modeling Language (UML).

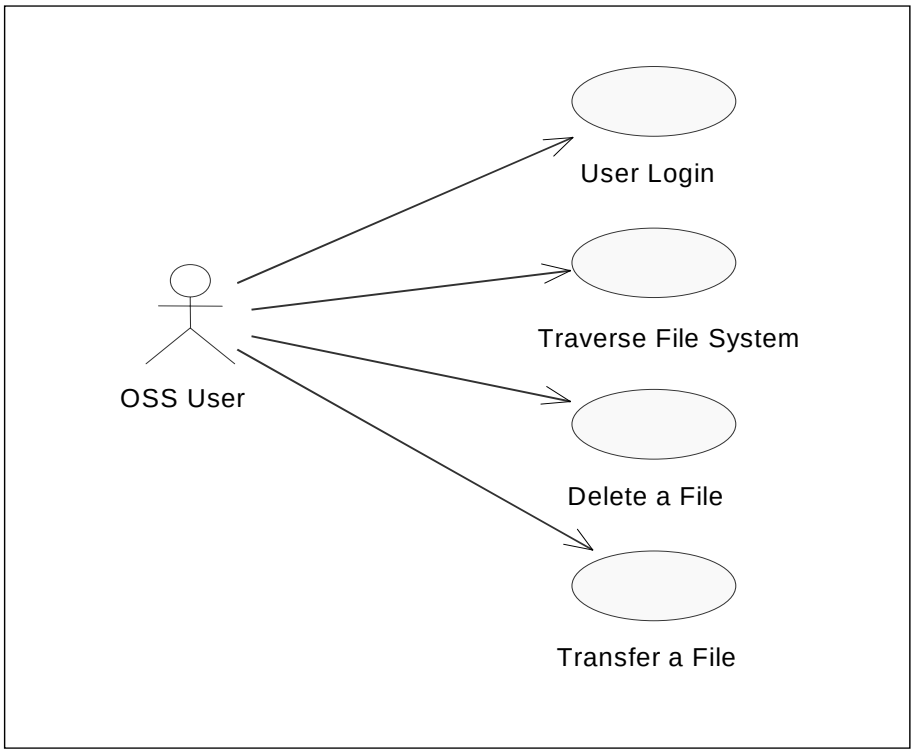

Figure 4-1 Typical scenario the proposed framework must consider

4.2 User Login

4.2.1 Description

To login to a **VirtualFileSystem**, it is first necessary for the client to obtain a list of available **VirtualFileSystem**s. The CORBA Name Service may be used for this purpose. **VirtualFileSystem**s are registered with a particular Name Service. **VirtualFileSystem** objects do not have to reside on the same host. A client first connects to the Name Service and traverses it to find a particular **NamingContext** that contains the entries for each available **VirtualFileSystem**. A list of those entries is acquired, and after they have been resolved, the client may attempt to login.

Login details are passed to a selected **VirtualFileSystem** by invoking its **login()** operation. A given **VirtualFileSystem** may represent one of a number of different kinds of file transfer servers. These may include FTP servers, FTAM responders, or variants of either. Each protocol requires different functionality from a **FileTransferSession**.

Using the **VirtualFileSystem** properties (a typical implementation may use the property indicating the preferred protocol driver[1] of the **VirtualFileSystem**) a new **FileTransferSession** is instantiated. Using the login details provided by the client,

an attempt is made to login to the file transfer server. If the attempt is successful, a reference to the appropriate instantiation of the **FileTransferSession** interface is returned to the client. In the event of an unsuccessful login, the operation will throw an appropriate exception.

4.2.2 Code Sample

The use of the CORBA Name Service allows multiple **VirtualFileSystem**s from different hosts to be registered and located using one network reference. A configuration class may be used to instantiate an arbitrary number of **VirtualFileSystem**s and register them with a Name Service running on a particular host.

References to **VirtualFileSystem**s should be registered within a specific context in the Name Service. Clients may inspect the entries in that context, and select a required **VirtualFileSystem** to login to.

```
VirtualFileSystem my_VirtualFileSystem =
   VirtualFileSystemHelper.narrow(
      my_VFS_context.resolve(name_of_VFS));
FileTransferSession my_FTS =
   my_VirtualFileSystem.login( user_name, pass_word,
      account, my_DirectoryHolder );
```

Typically, login details are passed to a Factory object along with the class name of the preferred **FileTransferSession** protocol driver implementation. The class name for these protocol drivers could follow the general format described below to enable a **FileTransferSession** instantiation to load protocol drivers from different vendors based on the properties of the **VirtualFileSystem**.

<protocol>.<vendor> (e.g., **ftam.Foobar**)

If the preferred implementation is unavailable, a default one may be utilized.

```
try {
   Class driverClass = Class.forName("ftam.default");
      Driver fooDriver =
         Driver(driverClass.newInstance());
   FileTransferSession fooFTS =
      new FileTransferSession(fooDriver);
} catch (Throwable t) {}
```

If the login is successful, a valid **FileTransferSession** reference will be returned to the client. Otherwise one of the exceptions SessionException, FileNotFoundException, or IllegalOperationException will be thrown to indicate why the login request failed.

1. The sample implementations described within this section refer to a protocol driver implementation that a **FileTransferSession** instantiation will use to communicate with the remote file serving mechanism. It is only one implementation approach to enable communication between a **FileTransferSession** and a remote server.

4.2.3 Interaction Diagram for a successful login

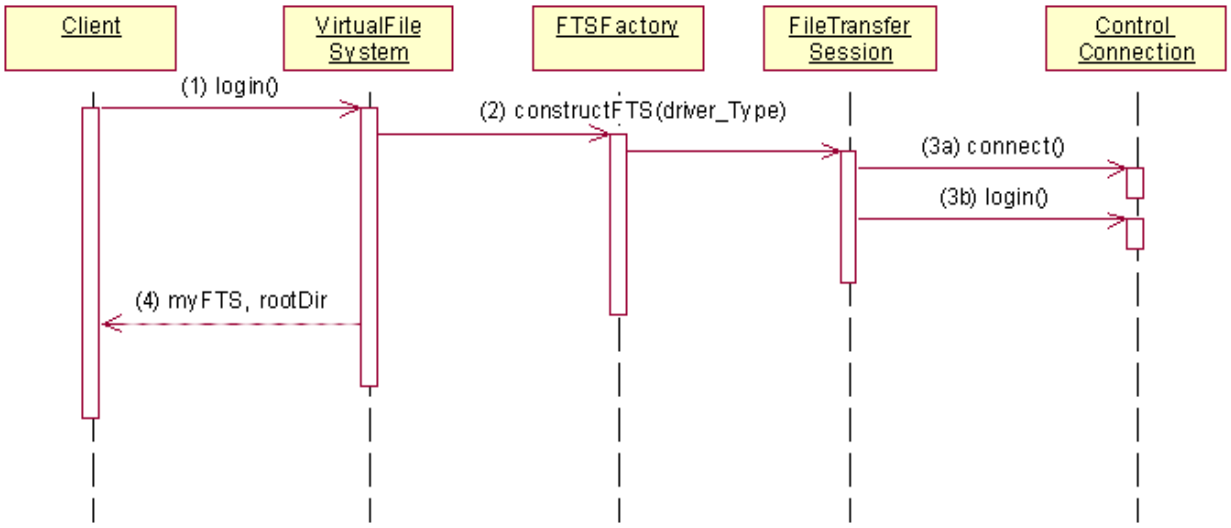

Figure 4-2 Login Diagram

1. The login operation is called on an object implementing the **VirtualFileSystem** interface, with login details as parameters.
2. The **VirtualFileSystem** object calls **constructFTS()** on an object implementing a Factory interface. In addition to the login details, the class name of the preferred protocol driver implementation that the **FileTransferSession** instantiation will use to communicate with the remote server are used to create the new **FileTransferSession** object. This new **FileTransferSession** object will then load the appropriate driver class.
3. The **FileTransferSession** implementation then attempts to login using protocol specific primitives and operations provided by an appropriate driver for the remote server:
   - A connection is first made to the file transfer server.
   - A login is attempted.

4.3 Traversing the File System

4.3.1 Description

Once a client has logged into a **VirtualFileSystem**, a reference to a **Directory** interface is returned as an **out** parameter from the **login()** operation provided by the **VirtualFileSystem** interface. The **Directory** interface represents the “root” directory of the **VirtualFileSystem** and is the starting point upon login for any client.

Since the **Directory** interface inherits from the **File** interface, **Directory** interfaces may be referenced as **File** interfaces. To perform an operation on a file (for example, transfer, delete) it is necessary to obtain an IOR for the **File**. The proposed framework provides two different ways to obtain an IOR:

1. The client can obtain an IOR for a **File** directly by invoking the **get_file()** operation if they have knowledge of the full pathname of the file in question. This operation is independent of the working directory from where it is invoked.

2. Alternatively, a client can obtain **File** IORs through a discovery mechanism that consists of a number of **list()**, **set_directory()**, **list()** iterations. A client may get the references contained in the **FileList** sequence through the **list()** operation provided by the **Directory** interface.

When the client has the references contained in the **Directory** interface's **FileList** sequence, they may be examined to determine whether any refer to a **Directory** object. The **FileList** contains a number of **FileWrapper struct**s, each of which contain a reference to a **File** and an enumeration that identifies its type.

Having selected a particular **Directory** reference, it is then necessary for the client to ensure that the contents of the **Directory** reference accurately mirrors the physical directory it represents at the remote server. The **set_directory()** operation is provided by the **FileTransferSession** interface to enable a client to perform this task. At this stage it may be appropriate for the **FileTransferSession** to populate the **FileList** sequence with **FileWrapper** references. Any further changes of directory can be made in the same way.

The references to the **File** interfaces may then be used as parameters in the various file related operations (for example, **delete()**, **transfer ()**, **append()**, etc.).

4.3.2 Code Sample

The client obtains the initial **Directory** reference from the output parameter of the **VirtualFileSystem** interface's **login()** operation.

```
DirectoryHolder dh = new DirectoryHolder ();
try {
   FileTransferSession my_fits =
      my_VFS.login (user, pass, acct, dh);
} catch(Throwable t) {}
Directory root_dir = dh.value;
```

The references to the **File** interfaces contained within the **Directory** interface implementation may be accessed through an **out** parameter of the **Directory** interface's **list()** operation. An integer is passed as an **in** parameter to specify how many references to return initially. The **FileIterator** object is used to retrieve subsequent references.

```
FileListHolder flh = new FileListHolder ();
FileIteratorHolder fi = new FileIteratorHolder ();
int how_many = 20;
root_dir.list (how_many, flh, fi);
FileWrapper [ ] my_list = flh.value;
Vector directory_list = new Vector ();
Directory test_dir;
for ( int i = 0; i< my_list.length; i++) {
   if(my_list[i].file_type == FileType.directory)
      directory_list.addElement (
         DirectoryHelper.narrow(my_list[i]);
}
```

Having obtained a reference to a particular **Directory**, it is then necessary to update its contents. The operation **set_directory()** in the **FileTransferSession** interface provides this functionality and can be invoked in the following manner:

```
Directory new_dir = (Directory)directory_list.elementAt (0);
my_fts.set_directory(new_dir);
```

Typically the **File** references returned from the **list()** operation can be used as parameters in various file related operations provided by the **FileTransferSession** interface.

```
File my_file = my_list[0].the_file;
File another_file = my_list[1].the file;
my_fts.append(my_file, another_file);
my_fts.delete(my_file);
```

It is also possible for a client to obtain a reference to a specific **File** via the **get_file()** operation provided by the **FileTransferSession** interface if the location of the file is known.

```
String[ ] fullPathName = new String [3];
fullPathName[0] = new String(“my”);
fullPathName[1] = new String(“path”);
fullPathName[2] = new String (“any.txt”);
FileWrapper file_reference =
   my_fts.get_file(fullPathName);
File reference = file_reference.the_file;
```

4.3.3 Interaction Diagram for successfully traversing the file system

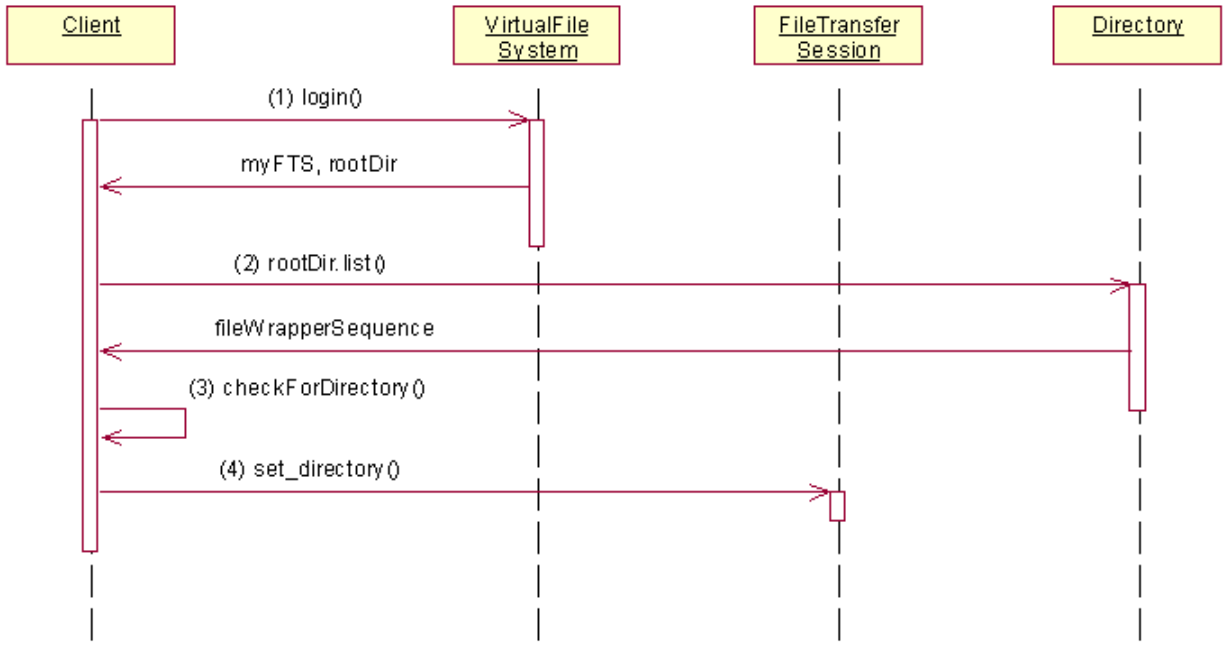

Figure 4-3 Traversing the File System Diagram

This interaction diagram illustrates the communication between the various framework components when a client wishes to traverse the file system's hierarchy.

1. The **login()** operation is called on an object implementing the **VirtualFileSystem** interface, with login details as parameters. An object reference of type **FileTransferSession** is returned, or an appropriate exception is thrown. A reference to the root directory is also returned through an **out** parameter.
2. The reference to the root directory can be used by the client to invoke the **list()** operation provided by the **Directory** interface to obtain a list of **File** references, in the form of a sequence of **FileWrapper**s, associated with that directory.
3. The client iterates through the list of **FileWrapper** references returned by 2) to identify **Directory** interfaces.
4. The client can choose the new working **Directory** reference from the list of available **Directory** interfaces. The **set_directory()** operation is used to update or populate the **Directory** reference ensuring that its contents accurately mirror that of the physical directory at the remote server.

4.4 Deleting a Remote File

4.4.1 Description

The deletion of a remote file would seem to be a straightforward matter. A user must first have the appropriate permissions to complete such a command. An inspection of the access rights within the **File** object determines that. Alternatively, the file transfer server itself can decide whether a user could complete such an action. In either case a client will send the appropriate primitive or set of primitives to the file transfer server.

However, although this would delete the actual file stored on the server, it is also necessary for the appropriate **File** and **Directory** objects to be updated. Each **File** object contains a reference to the parent **Directory** object. This may be used to call the **list()** operation that will return the sequence of **FileWrapper** objects. The appropriate **File** is removed from the sequence, and the updated version passed back to the **Directory** object. The **File** object is then discarded.

Although the **Directory** object would be up to date, a mechanism has to be employed to let any clients know that a change has taken place, and that the **list()** operation should be invoked once more. The Event Service could be used to achieve this.

4.4.2 Code Sample

Once a file has been selected for deletion, the client invokes the operation:

```
try {
my_FileTransferSession.delete(file_for_deletion);
} catch (SessionException e) {
} catch (FileNotfoundException e) {
} catch (RequestFailureException Exception e) {
} catch (IllegalOperationException e) {
}
```

where **file_for_deletion** is the reference to the **File** object representing that file. From the **File** object, the **FileTransferSession** determines the absolute path name of the file, and then sends the appropriate primitive to the file transfer server. Assuming that the client has the appropriate permissions and the operation is a success, it is necessary to update the parent **Directory** object. This may be done by resetting the sequence of **File** objects contained within the **Directory** interface.

At this point, the **Directory** object is aware of the change, but any client with a reference to that object is not. A number of approaches may be used to address this. The simplest is to ignore it, and only update when the user calls an action on the object. Alternatively, the CORBA Event Service could be used to signify to all clients that the object has been changed, or the client could implement an interface that allows callbacks.

4.4.3 Interaction Diagram for successfully deleting a remote file

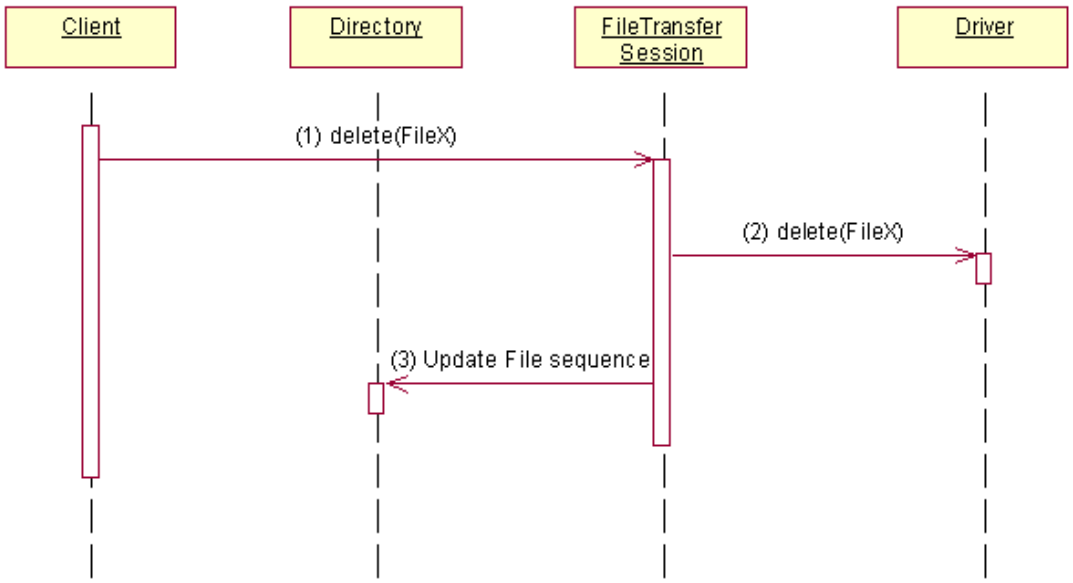

Figure 4-4 Deleting a Remote File Diagram

1. The **delete()** operation is called by the client on a **FileTransferSession** object. The **File** object representing the file to be deleted is passed as a parameter.
2. The appropriate primitive or set of primitives is sent to the file transfer server, with the absolute pathname of the file to be deleted, and the physical file at the remote server is removed.
3. The **Directory** object's sequence of **File** references is then updated.

4.5 Transferring a File

4.5.1 Description

To transfer a file between a source and destination **VirtualFileSystem** a client must be logged into both **VirtualFileSystem**s in order to obtain references to a source and destination **FileTransferSession**. A client will invoke the **transfer()** operation on a source **FileTransferSession**, which is determined by the fact that it is the **FileTransferSession** that contains a reference to the physical file to be transferred to a target location. The client will pass two **File** references as part of the **transfer()** invocation: a reference to the source **File** and a reference to the destination **File** at the target **FileTransferSession**[2]. During the transfer of data, it is always the source **FileTransferSession** that assumes control of the transfer.

By examining the attributes of the source **File** reference passed by the client to its **transfer()** operation, a **FileTransferSession** can determine whether it is to initiate the transfer of a file or to receive a file. A source **FileTransferSession** will then establish a connection with the target **FileTransferSession** using the value of its own **protocols_supported** attribute and that of the target **FileTransferSession**.

The source **FileTransferSession** will initiate data transfer by invoking the **transfer()** operation on the target **FileTransferSession** and sending an appropriate retrieval primitive to its associated file transfer server using its preferred protocol driver. When the destination **FileTransferSession**'s **transfer()** operation is invoked, it can determine that it will be receiving data by examining the properties of the source **File** parameter, and will send an appropriate storage primitive to its file transfer server using its preferred protocol driver.

4.5.2 Code Sample

To transfer files between two **FileTransferSession** objects, a connection must be established between the two. Establishing this connection and initiating the transfer of data across this connection is of no concern to the client but is related to the **protocols_supported** attribute exposed as part of the **FileTransferSession** interface. A client will invoke the **transfer()** operation with references to the source **File** to be transferred and the destination **File** at the target **FileTransferSession**. However, an implementation of the **FileTransferSession** interface will use the values of its own, and the destination **FileTransferSession**'s, **protocols_supported** attribute to establish a connection when attempting to transfer the source file to its destination.

Since the **protocols_supported** attribute is a sequence of **ProtocolSupport** structs,

```
struct ProtocolSupport { string protocol_name;
ProtocolAddressList addresses; };
```

each struct will contain the name of the protocol supported by the **FileTransferSession** and a sequence of addresses and ports (for example, TCP/IP, 255.255.255.1:8001) where a connection from a peer **FileTransferSession** can be established using sockets. An implementation should ensure that the protocol supported by the source and destination **FileTransferSession**s are the same before attempting to create a socket connection. The code sample that follows illustrates how this connection may be established.

2. A new **File** reference must be created by the client by invoking the **create_file()** operation on the target **FileTransferSession**, prior to calling the **transfer()** operation.

```
String my_protocol = “TCP/IP”;
ProtocolSupport[ ]prot =
   secondaryFTS.protocols_supported();
int index = -1;
for (int k=0; (k<prot.length&&index == -1); k++)
   if (prot[k].protocol_name.compareTo(my_protocol)==0)
      index = k;
if (index == -1)
   throw new TransferException (“Unsupported Protocols”);
String[ ] addresses = prot[index].addresses;
boolean connected = false;
for (int k = 0; (k<addresses.length&&!connected); k++ {
   String address = addresses[k];
   try {
      StringTokenizer toke =
         new StringTokenizer (address, “:”);
      host = toke.nextToken();
      port = Integer.parseint(toke.nextToken());
      transfer_socket = new Socket (host, port);
      connected = true;
   } catch (Exception e) { }
}
```

Once the source **FileTransferSession** has established a socket connection with the target it will invoke the target **FileTransferSession**’s **transfer()** operation. The code sample below illustrates that this may be implemented asynchronously to enable the source **FileTransferSession** to continue initializing its side of the file transfer.

```
Thread peer_thread = new Thread {
   public void run () {
      try {
         secondaryFTS.transfer(src_file, dest_file);
      } catch(SessionException e) {
      } catch(TransferException e) {
      } catch(FileNotFoundException e) {
      } catch(RequestFailureException e) {
      } catch(IllegalOperationException e) {
      }
   }
}
peer_thread.start();
```

The source **FileTransferSession** will then establish a buffered queue that will write data to the socket connection with the target **FileTransferSession**. It will then send an appropriate retrieval primitive or set of primitives to its associated remote file server by invoking the appropriate operation on its preferred protocol driver.

At the same time, the target **FileTransferSession** has determined from the properties of the source and destination **File** parameters, that it will be receiving data during the transfer. It establishes a buffered queue that will read data from the socket connection

with the source **FileTransferSession**. It also sends an appropriate storage primitive or set of primitives to its associated remote file server by invoking an appropriate operation on its preferred protocol driver.

4.5.3 Interaction Diagram for successfully transferring a file

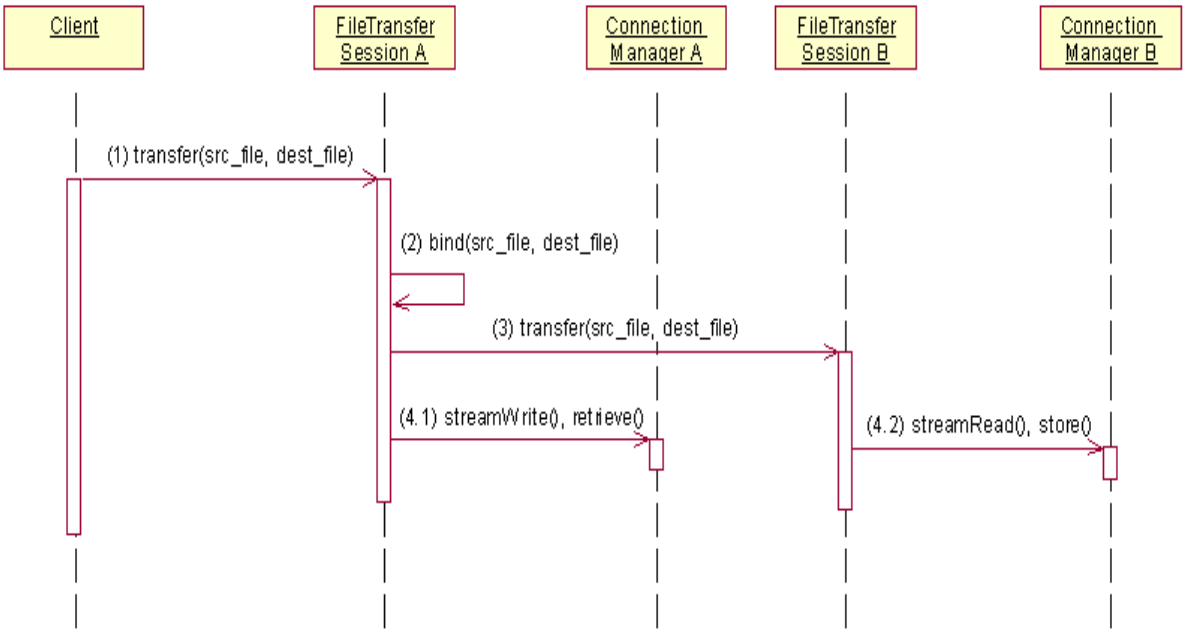

Figure 4-5 Transferring a File between two FileTransferSession Objects Diagram

1. The client invokes the **transfer()** operation of the source **FileTransferSession** (the **FileTransferSession** that has a reference to the file to be transferred).
2. The source **FileTransferSession** has determined, from the properties associated with the **src_file**, that it is to be the source for the transfer and will control the internal operations concerning a file transfer. It then creates a data connection with the target **FileTransferSession** using the **associated_session** property associated with the **src_file** and **dest_file** references.
3. The source **FileTransferSession** then invokes the **transfer()** operation on the target **FileTransferSession**.
4. The data connection between the two **FileTransferSession**s is used to transfer data:

- The source **FileTransferSession** will invoke a **streamWrite()** method on its associated **ConnectionManager** object, that will create an output stream for writing data to the data connection. It will also invoke a **retrieve()** method on its preferred protocol driver that will attempt to retrieve the requested file from the remote file transfer server.
- The destination **FileTransferSession** will invoke a **streamRead()** method on its associated **ConnectionManager** object, that will create an input stream for reading data from the data connection. It will also invoke a **store()** method on its preferred protocol driver that will request the remote file transfer server to store the file it will be sending.

If each of these steps occur successfully, the source file's data should be transferred from the source **FileTransferSession**'s remote server, onto the connection with the destination **FileTransferSession** and onwards to the destination **FileTransferSession**'s remote file server.

References 5

Note – This appendix was removed, the references are listed in footnotes where used in the new document.

0.1 List of References

[1]IETF RFC 959 “File Transfer Protocol (FTP)”, J. Postel, J. Reynolds. October 1985.

[2]IETF RFC 1415 “FTP-FTAM Gateway Specification”, J. Mindel, R. Slaski. January 1993.

[UML]M. Fowler, S. Kendall, “UML Distilled – Applying the Standard Object Modeling Language”, ISBN 0-201-32563-2.

Complete OMG IDL *4*

```
//File: CosFileTransfer.idl
#ifndef _COS_FILE_TRANSFER_IDL_
#define _COS_FILE_TRANSFER_IDL_

#include <CosPropertyService.idl>

#pragma prefix "omg.org"

module CosFileTransfer {
typedef string Istring;
typedef Istring ProtocolAddress;
typedef long ContentType;
const ContentType FTAM_1 = 1;
const ContentType FTAM_2 = 2;
const ContentType FTAM_3 = 3;
const ContentType FTAM_4 = 4;
const ContentType FTAM_5 = 5;
const ContentType NBS_9 = 6;
const ContentType INTAP_1 = 7;

exception CommandNotImplementedException {
Istring reason;
};

exception SessionException {
Istring reason;
};

exception TransferException {
Istring reason;
};

exception FileNotFoundException {
```

```
lstring reason;
};


exception RequestFailureException {
lstring reason;
};

exception IllegalOperationException {
lstring reason;
};

interface VirtualFileSystem;
–
struct AccessLevel {
boolean read;
boolean insert;
boolean replace;
boolean extend;
boolean erase;
boolean read_attr;
boolean change_attr;
boolean delete;
};

typedef sequence<ProtocolAddress> ProtocolAddressList;

struct ProtocolSupport {
lstring protocol_name;
ProtocolAddressList addresses;
};

typedef sequence<ProtocolSupport> SupportedProtocolAddresses;
–
interface Directory;

interface FileTransferSession;

typedef lstring FileName;

typedef sequence<FileName> FileNameList;

interface File:CosPropertyService::PropertySetDef {
readonly attribute FileName name;
readonly attribute FileNameList
complete_file_name;
readonly attribute Directory parent;
readonly attribute FileTransferSession
associated_session;
};
```

```
enum FileType {nfile, ndirectory};

struct FileWrapper {
File the_file;
FileType file_type;
};

typedef sequence<FileWrapper> FileList;

interface FileIterator;

interface Directory : File {
void list(in unsigned long how_many,
out FileList fl,
out FileIterator fi);
};
–
interface FileIterator {
boolean next_one(out FileWrapper f);
boolean next_n(in unsigned long how_many,
  out FileList fl);
void destroy();
};

interface FileTransferSession {
readonly attribute SupportedProtocolAddresses protocols_supported;
void set_directory(in Directory new_directory)
raises( SessionException,
 FileNotFoundException,
 RequestFailureException,
 IllegalOperationException);
File create_file(in FileNameList name)
raises( SessionException,
 FileNotFoundException,
 RequestFailureException,
 IllegalOperationException);
Directory create_directory(in FileNameList name)
raises( SessionException,
 FileNotFoundException,
 RequestFailureException,
 IllegalOperationException);
FileWrapper get_file (in FileNameList  complete_file_name)
raises( SessionException,
 FileNotFoundException,
 RequestFailureException,
 IllegalOperationException);
void delete(in File file)
raises( SessionException,
 FileNotFoundException,
 RequestFailureException,
 IllegalOperationException);
```

```
void transfer(in File src, in File dest)
raises( SessionException,
 TransferException,
 FileNotFoundException,
 RequestFailureException,
 IllegalOperationException);
void append(in File src, in File dest)
raises( CommandNotImplementedException,
 SessionException,
 TransferException,
 FileNotFoundException,
 RequestFailureException,
 IllegalOperationException);
void insert(in File src, in File dest,
                in long offset)
raises( CommandNotImplementedException,
  SessionException,
 TransferException,
 FileNotFoundException,
 RequestFailureException,
 IllegalOperationException);
void logout();
};
–

interface VirtualFileSystem {
enum NativeFileSystemType {
FTAM,
FTP,
NATIVE
};
readonly attribute NativeFileSystemType file_system_type;
typedef sequence<ContentType> ContentList;
readonly attribute ContentList supported_content_types;

FileTransferSession
login(in Istring username, in Istring password,
   in Istring account, out Directory root)
raises(SessionException,
 FileNotFoundException,
 IllegalOperationException);
};
};

#endif //_COS_FILE_TRANSFER_IDL_
```

```
//File: CosFileTransferFTF.idl

#ifndef _COS_FILE_TRANSFER_IDL_
#define _COS_FILE_TRANSFER_IDL_
#include <CosProperty.idl>

#pragma prefix "omg.org"

module CosFileTransfer {

  // FileEntry types

  interface Directory;
  interface File;

  // FileSystem login session

  interface FileSession;


  // Filesystem entries, Files and Directories,
  // have multi-component path names

  typedef wstring EntryName;
  typedef sequence<EntryName> EntryPath;

  // File size, offset, octet count, and contents

  typedef unsigned long long FileLength;
  typedef unsigned long long FileOffset;
  typedef unsigned long long FileCount;
  typedef sequence<octet> FileOctetSeq;

  typedef short FilePos;
  const FilePos BEGIN = 0; // FileOffset is relative to beginning of File
  const FilePos END   = 1; // FileOffset is relative to end of File


  // Directory listing size and list offset

  typedef unsigned long long DirEntryCount;
  typedef unsigned long long DirEntryOffset;


  // Directory listing Types

  typedef short DirEntryType;
  const DirEntryType FILE_ENTRY = 0;
  const DirEntryType DIR_ENTRY  = 1;

  struct DirEntry {
```

```
        EntryName name;
        DirEntryType type;
        CosPropertyService::Properties props;
    };

    typedef sequence<DirEntry> DirEntrySeq;

    interface DirEntryIterator;

    // TransferEndPoint Types

    interface TransferEndPoint;
    typedef wstring TransferProtocol;
    typedef sequence<TransferProtocol> TransferProtocolSeq;

    typedef short TransferEndPointRole;

    const TransferEndPointRole SOURCE = 0;
    const TransferEndPointRole SINK = 1;
    const TransferEndPointRole SINK_INSERT = 2;

    // transfer protocol specific information

    typedef wstring TransferDetail;

    typedef short TransferState;
    const TransferState CREATE  =  0; // the end point has been created (initial
state)
    const TransferState LISTEN  =  1; // the end point is awaiting active
connection
    const TransferState CONNECT =  2; // the end point is connected to its
peer
    const TransferState ACTIVE  =  3; // the transfer is in progress
    const TransferState COMPLETE = 4; // transfer has completed succesfully
    const TransferState ABORT =    5; // transfer has been aborted

    struct TransferStatus {
        TransferState state;       // current transfer state
        FileCount current_count;   // current transfer count
        FileCount max_count;       // expected transfer size bytes/chars
    };


    // Exceptions

    typedef short ErrorCode;
    const ErrorCode UNSPECIFIED       = 0; // Error category not defined
    const ErrorCode UNAVAILABLE       = 1; // The service is not available at
this time
    const ErrorCode UNSUPPORTED       = 2; // operation not supported,
```

```
illegal parameter value
    const ErrorCode NO_PERMISSION     = 3; // No permission to perform the
operation

    const ErrorCode ENTRY_EXISTS      = 4; // Entry should not already exist
for operation
    const ErrorCode ENTRY_PATH_ERROR  = 5; // Entry path component
missing or invalid
    const ErrorCode ENTRY_IO_ERROR    = 6; // error opening, reading,
writing, closing file
    const ErrorCode DIR_NOT_EMPTY     = 7; // (rmdir required empty
directory)

    const ErrorCode TRANSFER_IO_ERROR = 8; // error opening,
transferring, or closing connections
    const ErrorCode TRANSFER_ABORT    = 9;

    exception FileSystemError {
        ErrorCode error;
        wstring desc;
    };

    // Error transferring between two files

    exception TransferError {
        TransferEndPointRole error_endpoint;
        ErrorCode error;
        wstring desc;
    };

    // FileSystem provided by service

    interface FileSystem {

        FileSession login(in wstring user,
                          in wstring password,
                          in CosPropertyService::Properties login_properties,
                          out Directory initial_dir)
            raises(FileSystemError);

        wstring get_system_id();
    };

    // FileSession client obtains by logging in to FileSystem

    interface FileSession {
        void destroy();
    };


    // Common File system entry methods
```

```
interface FileSystemEntry: CosPropertyService::PropertySet {

    EntryName get_name()
        raises (FileSystemError);

    EntryPath get_path()
        raises (FileSystemError);

    boolean exists()
        raises (FileSystemError);

    void remove()
        raises (FileSystemError);

    Directory get_parent()
        raises (FileSystemError);

    FileSession get_session()
        raises (FileSystemError);

    void destroy();
};

interface File;

// Directory manipulation and listing

interface Directory: FileSystemEntry {

    DirEntryIterator list(in CosPropertyService::PropertyNames listProps)
        raises (FileSystemError);

    Directory create_directory(in EntryPath fpath)
        raises( FileSystemError);

    File get_file(in EntryPath fpath, in boolean create)
        raises( FileSystemError);

    Directory get_directory(in EntryPath fpath)
        raises( FileSystemError);

    void remove_entry(in EntryPath fpath)
        raises( FileSystemError);
};

// Iterator to retrieve results of Directory list

interface DirEntryIterator {
    DirEntrySeq next(in DirEntryOffset from_dir_entry,
                     in DirEntryCount max_dir_entries)
```

```
        raises (FileSystemError);
    void destroy();
  };


  // File manipulation and basic transfer

  interface File: FileSystemEntry {

    void copy(in File dest)
        raises( TransferError);

    void append(in File dest)
        raises( TransferError);

    void insert(in File dest, in FileOffset offset)
        raises( TransferError);

    TransferEndPoint create_end_point(in TransferEndPointRole ep_role,
                                      in FilePos seek,
                                      in FileOffset offset,
                                      in TransferProtocol ep_protocol)
        raises (FileSystemError);

    TransferProtocolSeq get_end_point_protocols();
  };


  // File transfer

  interface TransferEndPoint
  {
    TransferDetail go_to_listen()
        raises(FileSystemError);

    TransferDetail connect_to_peer(in TransferDetail passive_detail)
        raises(FileSystemError);

    void set_peer(in TransferDetail active_detail)
        raises(FileSystemError);

    TransferStatus get_transfer_status()
        raises (FileSystemError);

    void transfer()
        raises (FileSystemError);

    void abort()
        raises (FileSystemError);

    void destroy();
```

```
    };

    // File transfer using an iterator

    interface OctetTransferIterator {

        FileOctetSeq get_octet_seq(in FileOffset from_octet, in FileCount
max_octets)
            raises (FileSystemError);

        void put_octet_seq(in FileOffset to_octet, in FileOctetSeq octetSeq)
            raises(FileSystemError);

        void destroy()
            raises(FileSystemError);

    };
};
#endif // COS_FILE_TRANSFER_IDL
```

Compliance Issues *7*

Note – This appendix has been removed. Specific compliance points such as optional operation and transfer protocol support are described fully in Section 3.2 Conformance Criteria in the current specification. Details on support of property values is described in detail in Chapter 3.

~~In order to comply with this specification, all of the interfaces described must be supported and implemented. The specification defines a set of standard file properties associated with implementations of the **File** interface, that must at least be understood (but not necessarily implemented) by all conformant implementations.~~

Glossary

Note – This glossary has been removed from the specification

Glossary of Terms

File Transfer Client: Any reference within this response to a File Transfer Client should be read as FTP client or FTAM Initiator.

File Transfer Server: Any reference within this response to a File Transfer Server should be read as FTP server or FTAM responder.

Network Element: Any piece of software or hardware in the network that can be independently addressed.