AMS Extensions
Version 2.0

Andrew Hanushevsky
Stanford Linear Accelerator Center
7/15/98

Acknowledgements

Urs Bertschinger, Objectivity Inc.
for co-designing the initial oofs() interface and modifying AMS v4 to use it.

Karl Quackenbush, Objectivity Inc.
for suggested improvements and modifying AMS v5 to use it.

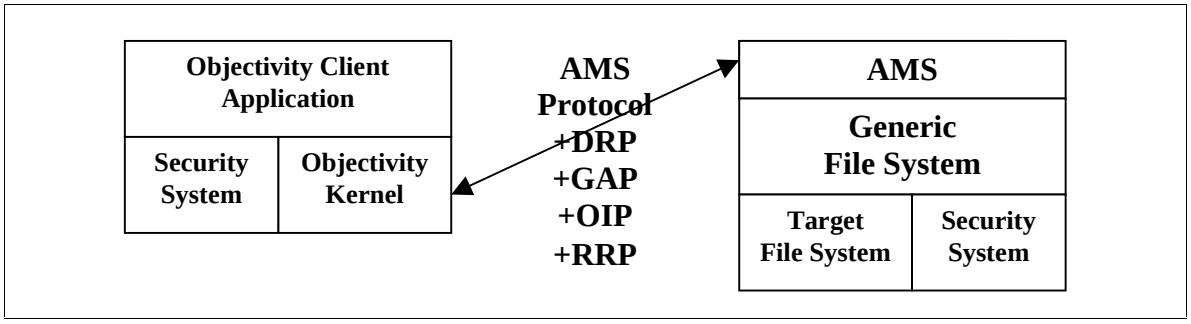
Marcin Nowak, CERN
for invaluable debugging assistance and helpful modifications.

1 Introduction

This document describes Advanced Multithreaded Server (**AMS)** extensions providing:

- a general file system interface, **oofs()**,
- authentication through a Generic Authentication Protocol (**GAP**),
- client-initiated transfer of opaque information through an Opaque Information Protocol (**OIP**),
- variable request timeout through a Defer Request Protocol (**DRP**), and
- dynamic load balancing through a Request Redirection Protocol (**RRP**).

The relationships between these components are shown in the following figure.



Major changes occur in the **AMS**. Prior to the changes, the **AMS** was distributed as a single module. The new **AMS** is distributed as two linkable components:

- the core **AMS** functions, and
- the generic file system interfaced via an **oofs()** interface.

These two components are then linked to two site selectable components:

- a target file system (e.g., Unix, HPSS, Veritas, etc.), and
- a complimentary security service that provides client authentication.

A functional **AMS** is built by linking the appropriate four components. The default **AMS** is normally distributed as an executable module linked to use the native Unix File System (**UFS**) and a null security service.

The client-side components are relatively straightforward. The Objectivity Kernel internally handles **DRP** and **RRP**. Client-supplied programs to transmit authentication and opaque information via registered callback functions implicitly use the **GAP** and **OIP** components. Hence, the Kernel is distributed as a single linkable component, as before.

1.1 Acronyms

AMS Advanced Multithreaded Server
AMSC Advanced Multithreaded Server Collective
DRP Defer Request Protocol
GAP Generic Authentication Protocol
OCSK Objectivity Client-Side Kernel
OOFS Object Oriented File System
OIP Opaque Information Protocol
RRP Request Redirect Protocol

1.2 Definitions

```
// Parameter to the oofs_Open() function to set access mode
#define OOFS_S_IRWXU 0000700        // Owner: read, write, execute perm
#define OOFS_S_IRUSR 0000400        // Owner: read permission
#define OOFS_S_IWUSR 0000200        // Owner: write permission
#define OOFS_S_IXUSR 0000100        // Owner: execute/search permission
#define OOFS_S_IRWXG 0000070        // Group: read, write, execute perm
#define OOFS_S_IRGRP 0000040        // Group: read permission
#define OOFS_S_IWGRP 0000020        // Group: write permission
#define OOFS_S_IXGRP 0000010        // Group: execute/search permission
#define OOFS_S_IRWXO 0000007        // Other: read, write, execute perm
#define OOFS_S_IROTH 0000004        // Other: read permission
#define OOFS_S_IWOTH 0000002        // Other: write permission
#define OOFS_S_IXOTH 0000001        // Other: execute/search permission

// Parameter to the oofs_Open() function to set access type
#define OOFS_O_RDONLY  0            // open read/only
#define OOFS_O_WRONLY  1            // open write/only
#define OOFS_O_RDWR    2            // open read/write
#define OOFS_O_CREAT   0x100        // open creating the file

// Parameter to the oofs_set_info() function to describe handling
#define OOFS_SOI_ALWAYS       0     // keep resending information
#define OOFS_SOI_ONCE         1     // send information only once

// Valid return values from oofs() that return an integer
#define OOFS_OK               0     // Return code -> All is well
#define OOFS_ERROR           -1     // Return code -> Error occurred
#define OOFS_READDIR_LAST    –2     // Return code -> No more entries

// Direction of encryption (i.e., seal or unseal)
#define OOFS_Seal             0     // Encrypt buffer
#define OOFS_UnSeal           1     // Decrypt buffer

// Current version of the oofs() interfaces
#define OOFS_FILE_SYSTEM_DESC_VERSION 1
#define OOFS_SEC_SYSTEM_DESC_VERSION  1
```

1.3 Limits

```
// Longest filename returned by interface (including trailing null)
#define OOFS_MAX_FILE_NAME_LEN      (1024+1)

// Longest error message string returned (including trailing null)
#define OOFS_MAX_ERROR_LEN          (255+1)

// Maximum number of bytes of opaque information that can be set
#define OOFS_MAX_INFO_LEN           4096
```

1.4 Data Types

```
typedef char       oofsFileNameBuf[OOFS_MAX_FILE_NAME_LEN];
typedef char       oofsErrorBuf[OOFS_MAX_ERROR_LEN];

typedef ooInt32    oofsFileCreateMode;
typedef void      *oofsFileDesc;
typedef ooInt32    oofsFileOpenMode;
typedef void      *oofsReadDirDesc;
typedef int        oofsStatus;
typedef ooInt32    oofsXferSize;
typedef void      *oofsSecHandle
```

1.5 Data Structures

```
struct oofsFileOffset   // 64-bit file offset
{
      ooUInt32          high_offset;
      ooUInt32          low_offset;
};

struct oofsCredStruct   // Server/Client Credentials
{
      ooUInt32          cred_len;
      char              *cred_data;
);

struct oofsErrorStruct  // Error information structure
{
      ooUInt32          code;
      oofsErrorBuf      message;
};

struct oofsInfoStruct   // Server/Client Information
{
      ooUInt32          info_len;
      char              *info_data;
};
```

1.6 Server-Side oofs Interface Definition

```
struct oofsFileSystemDesc     // The oofs() interface definition
{
      int version;                  // OOFS_FILE_SYSTEM_DESC_VERSION

      oofsFileDesc     (*open)(     const char *FileName,   // FD-type
                                    oofsFileOpenMode,
                                    oofsFileCreateMode,
                                    oofsInfoStruct &,
                                    oofsCredStruct &,
                                    oofsErrorStruct &);
      int              (*close)(    oofsFileDesc,
                                    oofsCredStruct &,
                                    oofsErrorStruct &);
      oofsXferSize     (*read)(     oofsFileDesc,
                                    oofsFileOffset,
                                    char *buffer,
                                    oofsXferSize buffer_size,
                                    oofsCredStruct &,
                                    oofsErrorStruct &);
      oofsXferSize     (*write)(    oofsFileDesc,
                                    oofsCredStruct &,
                                    oofsFileOffset,
                                    const char *buffer,
                                    oofsXferSize buffer_size,
                                    oofsCredStruct &,
                                    oofsErrorStruct &);
      int              (*sync)(     oofsFileDesc ,
                                    oofsCredStruct &,
                                    oofsErrorStruct &);
      int              (*truncate)( oofsFileDesc,
                                    oofsFileOffset,
                                    oofsCredStruct &,
                                    oofsErrorStruct &);
      int              (*getsize)(  oofsFileDesc,
                                    oofsFileOffset &,
                                    oofsCredStruct &,
                                    oofsErrorStruct &);
      int              (*getmode)(  oofsFileDesc,
                                    oofsFileCreateMode &,
                                    oofsCredStruct &,
                                    oofsErrorStruct &);

      int              (*remove)(   const char *FileName,   // Misc
                                    oofsCredStruct &,
                                    oofsErrorStruct &);
      int              (*rename)(   const char *FileName1,
                                    oofsCredStruct &,
                                    const char *FileName2,
                                    oofsErrorStruct &);
      int              (*exists)(   const char *FileName,
                                    int &exists_flag,
                                    oofsCredStruct &,
                                    oofsErrorStruct &);
```

```
        oofsReadDirDesc  (*opendir)(  const char *DirectoryPath, // Dir
                                      oofsCredStruct &,
                                      oofsErrorStruct &);
        int              (*readdir)(  oofsReadDirDesc ,
                                      oofsFileNameBuf &,
                                      oofsCredStruct &,
                                      oofsErrorStruct &);
        int              (*closedir)( oofsReadDirDesc ,
                                      oofsCredStruct &,
                                      oofsErrorStruct &);

        void (*destruct)(); // Called at exit. Set to 0 -> no destruct

};

oofsFileSystemDesc *oofsGetFileSystem(); // Init & return -> FSDesc
```

1.7 Client-Side Security Interface Definition

```
struct oofsSecRoutinesDesc    // Authentication information
{
      int version;      //<- OOFS_SEC_SYSTEM_DESC_VERSION

      // Routines for client authentication
      oofsSecHandle     (*CreateSec)(const ooHandle(ooDDObj) &,
                                     oofsErrorStruct &);
      oofsCredStruct    *(*GetCred_D)(oofsSecHandle,
                                      const char *buffer,
                                      oofsXferSize buffer_size,
                                      oofsErrorStruct &);
      oofsCredStruct    *(*GetCred_M)(oofsSecHandle,
                                      oofsErrorStruct &);
      oofsStatus        (*DeleteSec)(ofsSecHandle,
                                     oofsErrorStruct &);

      // Routines for server authentication
      oofsStatus        (*AuthCred)(oofsSecHandle,
                                    oofsCredStruct &,
                                    const char *buffer,
                                    oofsXferSize buffer_size,
                                    oofsErrorStruct &);

      // Routine to handle data encryption and decryption
      oofsStatus        (*Crypt)(oofsSecHandle,
                                 const ooInt direction,
                                 const char *in_buffer,
                                 oofsXferSize in_buffer_size,
                                 char **out_buffer,
                                 oofsXferSize *out_buffer_size,
                                 oofsErrorStruct &);
      void              (*FreeBuff)(char *out_buffer);
};

// Routine to register the client-supplied security system
oofsStatus oofs_Register_Security(oofsSecRoutinesDesc *Sec_List);
```

1.8 Client-Side Opaque Information Interface Definition

```
oofsStatus oofs_set_info(oofsInfoStruct &,const ooInt flags);
```

2.1 oofsGetFileSystem()

```
oofsFileSystemDesc *oofsGetFileSystem();     // In
```

Function

Provide the filesystem interface definition.

Parameters

None.

Success

A non-null pointer to the oofsSecRoutinesDesc structure is returned.

Failure

A NULL (i.e., 0) pointer is returned.

Notes

1) The **oofsGetFileSystem()** routine is part of the of the filesystem definition and must be supplied by the underlying filesystem interface. It is called by the **AMS** to instantiate the filesystem interface.
2) The **oofsGetFileSystem()** routine is a constructor function and should perform any required initialization.
3) The filesystem interfaces are defined in subsequent sections.

2.1.1 oofs_close()

```
int oofs_close( oofsFileDesc fd,            // In
                oofsCredStruct &cred,       // In
                oofsErrorStruct &einfo)     // Out
```

Function

Closes an open file.

Parameters

fd

is the filehandle returned by **oofs_open()** associated with the file to be closed.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

2.1.2 oofs_closedir()

```
int oofs_closedir(oofsFileDesc fd,                // In
                  oofsCredStruct &cred,           // In
                  oofsErrorStruct &einfo)         // Out
```

Function

Closes an open directory.

Parameters

fd

is the filehandle returned by oofs_opendir() associated with the directory to be closed.

cred

are the credentials authenticating the remote caller. .A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

2.1.3 oofs_exists()

```
int oofs_close( const char *path,           // In
                int &exists_flag,           // Out
                oofsCredStruct &cred,       // In
                oofsErrorStruct &einfo)     // Out
```

Function

Determine whether a path or file exists.

Parameters

path

is the fully qualified name of the file to be tested for existence.

exists_flag

is the address of the variable to hold the status of the test. When a success indication is returned, exists_flag will have one of the following values:

0 the file was not found but the path exists, or

1 the file was found.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned with exists_flag set.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

2.1.4 oofs_getmode()

```
int oofs_getmode( oofsFileDesc fd,                  // In
                  oofsFileCreateMode &mode,         // Out
                  oofsCredStruct &cred,             // In
                  oofsErrorStruct &einfo)           // Out
```

Function

Return a file's creation mode.

Parameters

fd

is the filehandle returned by **oofs_open()** associated with the file whose creation mode is to be returned.

mode

is a pointer to a variable that is to hold the file's creation mode (i.e., read-write-execute permission bits). The creation mode is returned in POSIX format and only upon success. The returned value may not be meaningful if the file is protected by an access control list.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned with mode set.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

2.1.5 oofs_getsize()

```
int oofs_getsize( oofsFileDesc fd,                  // In
                  oofsFileOffset &flen,             // Out
                  oofsCredStruct &cred,             // In
                  oofsErrorStruct &einfo)           // Out
```

Function

Return a file's current size.

Parameters

fd

is the filehandle returned by **oofs_open()** associated with the file whose length is to be returned.

flen

is a pointer to a variable to hold the file's current size. The size is returned only upon success.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned with flen set.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

2.1.6 oofs_open()

```
oofsFileDesc oofs_open( const char *filename,            // In
                        oofsFileOpenMode open_mode,      // In
                        oofsFileCreateMode create_mode   // In
                        oofsInfoStruct &info,            // In
                        oofsCredStruct &cred,            // In
                        oofsErrorStruct &einfo)          // Out
```

Function

Open a file; optionally creating it.

Parameters

filename

is the name of the file that is to be opened. Typically, a fully qualified path is given.

open_mode

indicates how the file is to be opened (i.e., read, write, or update). If open_mode indicate that the file is to be created, the file is also opened in update mode. The following are valid mode values:

OOFS_O_RDONLY - open file for reading
OOFS_O_WRONLY - open the file for writing
OOFS_O_RDWR - open the file for reading and writing (i.e., update)
OOFS_O_CREAT - create the file and open with **OOFS_O_RDWR**

create_mode

holds the access mode bits to be assigned to the file when open_mode indicates that the file is to be created. The mode must be specified in POSIX format (.e.g., 744 corresponds to rwx--r—mode).

info

is a pointer to a structure describing opaque information that may be of use during the operation. Opaque information is only meaningful to **oofs()** routines and is passed with modification from the client. A null pointer or a zero length opaque information indicate that opaque information was not supplied.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

A non-null filehandle is returned. The filehandle is to be used for subsequent file operations.

Failure

Null (0) is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

Notes

1) If the file already exists and **OOFS_O_CREAT** is specified, an error is returned.

2.1.7 oofs_opendir()

```
oofsReadDirDesc oofs_opendir( const char *dirname,       // In
                              oofsCredStruct &cred,      // In
                              oofsErrorStruct &einfo)    // Out
```

Function

Open a directory for reading.

Parameters

dirname

is the name of the directory that is to be opened. Typically, a fully qualified path is given.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

A non-null dirhandle is returned. The dirhandle is to be used for subsequent directory operations.

Failure

Null (0) is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

2.1.8 oofs_read()

```
oofsXferSize oofs_read( oofsFileDesc fd,                  // In
                        oofsFileOffset offset,            // In
                        char *buffer,                     // Out
                        oofs XferSize buffer_size,        // In
                        oofsCredStruct &cred,             // In
                        ofsErrorStruct &einfo)            // Out
```

Function

Read zero or more bytes from an open file.

Parameters

fd

is the filehandle returned by **oofs_open()** associated with the file to be read.

offset

is the absolute offset, origin 0, into the file where the read is to begin.

buffer

is a pointer to a buffer that is to hold the contents of the file after the read completes

buffer_size

is the size of the actual buffer. No more than the specified number of bytes are actually read from the file.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

The actual number of bytes that were read.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

2.1.9 oofs_readdir()

```
int oofs_readdir( oofsFileDesc fd,                  // In
                  oofsFileNameBuf &buffer,          // Out
                  oofsCredStruct &cred,             // In
                  oofsErrorStruct &einfo)           // Out
```

Function

Read the next entry in an open directory.

Parameters

fd

is the filehandle returned by **oofs_opendir()** associated with the directory to be read.

buffer

is a pointer to the buffer that is to hold the null-terminated contents of the next entry in the directory. The buffer must be large enough to hold the longest possible entry plus one for the ending null character (i.e., **OOFS_MAX_FILE_NAME_LEN**). The buffer is set only upon success (see the notes).

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

Notes

1) When no more directory entries are left, **OOFS_READDIR_LAST** (a non-zero value) is returned.

2.1.10 oofs_remove()

```
int oofs_remove( const char *filename,       // In
                 oofsCredStruct &cred,       // In
                 oofsErrorStruct &einfo)     // Out
```

Function

Remove a file.

Parameters

filename

is the filename of the file to be removed. Normally, a fully qualified path is specified. Asterisks are not allowed in the filename.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

2.1.11 oofs_rename()

```
int oofs_rename( const char *old_filename,         // In
                 const char *new_filename,         // In
                oofsCredStruct &cred,              // In
                oofsErrorStruct &einfo)            // Out
```

Function

Rename a file.

Parameters

old_filename

is the existing name of a file. Normally, a fully qualified path is specified.

new_filename

is the name that the file is to have. It must be different from old_filename.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

2.1.12 oofs_set()

```
int oofs_set(   oofsFileDesc fd,            // In
                const char *options,        // In
                oofsErrorStruct &einfo)     // Out
```

Function

Set **oofs()** options for a file or for the system..

Parameters

fd

is the file descriptor returned by **oofs_open()** or **oofs_opendir()** to which the options are to apply. If fd is null, then the options apply globally and affect all future file descriptors.

options

is a null terminated string of single letter, space-separated, option letters. Valid letters are:

- **l** - log data transfer statistics into SYSLOG when **oofs_close()** or **oofs_closedir()** is called.
- **p** - print data transfer statistics on STDERR when **oofs_close()** or **oofs_closedir()** is called.
- **t** - race execution on STDERR.

Each letter may be optionally preceded by a plus (the default) to turn on the option or a minus, to turn off the option.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

Notes

1) The **oofs_set()** interface is not part of the **AMS**-defined set of interfaces. It is provided to enable externally linked programs additional control over the interface for debugging and performance monitoring purposes.
2) Default options can be provided on a global level when the following environmental variables are set to a value of one (1):
   `oofs_STATLOG` - log data transfer statistics into SYSLOG when **oofs_close()** or **oofs_closedir()** is called.
   `oofs_STATLOG` - print data transfer statistics on STDERR when **oofs_close()** or **oofs_closedir()** is called.
   `oofs_TRACE` - trace execution on STDERR
3) Setting global options (i.e., when fd is null) is not a thread-safe operation. Global option must be set by a single or interlocking threads.

2.1.13 oofs_sync()

```
int oofs_sync(  oofsFileDesc fd,            // In
                oofsCredStruct &cred,       // In
                oofsErrorStruct &einfo)     // Out
```

Function

Verify that all data is committed to permanent media.

Parameters

fd

is the filehandle returned by **oofs_open()** associated with the file to be synchronized.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

2.1.14 oofs_truncate()

```
int oofs_truncate(oofsFileDesc fd,                // In
                  oofsFileOffset file_size,       // In
                  oofsCredStruct &cred,           // In
                  oofsErrorStruct &einfo)         // Out
```

Function

Set the size of a file.

Parameters

fd

is the filehandle returned by **oofs_open()** associated with the file whose size is to be changed.

file_size

is the new size of the file. If the new size is smaller than the current size, the file is reduced in size and the excess bytes are discarded. If the new size if greater than the current size, the file is extended to the new size with binary zeroes.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

2.1.15 oofs_write()

```
oofsXferSize oofs_write(oofsFileDesc fd,                // In
                        oofsFileOffset offset,          // In
                        const char *buffer,             // In
                        oofs XferSize buffer_size,      // In
                        oofsCredStruct &cred,           // In
                        ofsErrorStruct &einfo)          // Out
```

Function

Write zero or more bytes into an open file.

Parameters

fd

is the filehandle returned by **oofs_open()** associated with the file to be written.

offset

is the absolute offset, origin 0, into the file where the write is to begin.

buffer

is a pointer to a buffer that is to hold the data to be written into the file.

buffer_size

is the size of the actual buffer. No more than the specified number of bytes are actually written to the file.

cred

are the credentials authenticating the remote caller. A null pointer or zero-length credentials indicate that no credentials were supplied.

einfo

is the error information structure used when an error occurs.

Success

The actual number of bytes that were written.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

Notes

1) Indication of errors may be delayed until some future operation on the file.

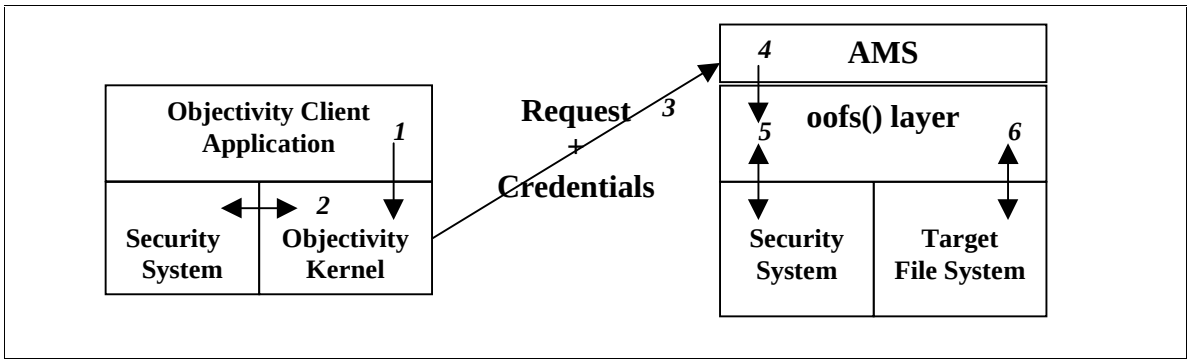
3 Generic Authentication Protocol

Generic Authentication Protocol (**GAP**) allows security information (e.g., credentials) to be transparently transferred from an Objectivity **AMS** client to the **AMS** server using a generic mechanism that can be easily modified to use site-specific security. **GAP** is supported only for **AMS** mediated connections and is suitable for private-key mechanisms such as Kerberos and Windows NT, and public-key mechanisms such as PGP[1]. It consists of eight site-replaceable client-side functions and a new client-side interface.

The following general steps comprise **GAP**:

- the installation makes available a set of security modules that can be used by an application to retrieve authentication credentials,
- the modules are linked with an application program,
- the application program registers these modules with the **OCSK** using the **oofs_Register_Security()** interface (1),
- prior to each **AMS** request, the **OCSK** calls the registered modules to retrieve authentication credentials (2),
- the credentials are sent by the **OCSK** to **AMS** (3),
- the **AMS** forwards the credentials to the **oofs()** interface (4),
- the **oofs()** interface decodes the credentials using an installation supplied mechanism (5), and
- the **oofs()** routines use the credentials to control access to the database (6).

The following diagram shows the basic sequence, numbered as above.



[1] GAP does not currently support challenge / response security protocols even though the interfaces to support such protocols can be provided.

3.1 Application Steps in GAP

1. The application program must initialize the **oofsSecRoutinesDesc** structure.

2. The application program must then call **oofs_Register_Security()** prior to opening any federated databases. Any unneeded routines may have their address in the vector set to zero. An example follows.

```
Struct oofsSecRoutinesDesc oofsSecRoutines =
     {OOFS_SEC_SYSTEM_DESC_VERSION,
     my_CreateSec,
     0,              // No need for a data GetCred
     my_GetCred,
     my_DeleteSec,
     0,              // AuthCred not supported
     0,              // Encryption not supported,
     0               // So no need for FreeBuff
     };

if (oofs_RegisterSecurity(oofsSecRoutines)) Fatal_Error();
```

3. Once the routines have been registered, the **OCSK** will call the routines whenever needed in support of **GAP**.

3.2 OCSK Steps in GAP

1. The **OCSK** calls the supplied routines in support of **GAP**. The following table indicates when each routine is called. Shaded cells indicate interfaces and functions that are currently not supported.

`CreateSec`	Called whenever a database is opened.
`GetCred_D`	Called to obtain credentials for a data operation.
`GetCred_M`	Called to obtain credentials for a meta-data operation.
`DeleteSec`	Called whenever a database is closed.
`AuthCred`	Called to authenticate server supplied credentials.
`Crypt`	Called to encrypt and decrypt a data stream.
`FreeBuff`	Called to release an encryption buffer.

2. The call to each routine is made in a timely manner and as close as possible to the time information is transmitted to the **AMS**. This is because certain security protocols place small lifetimes (approximately 15 seconds) on the credentials and large delays between obtaining the credentials and transmitting them may cause the credentials to become obsolete. A lengthy retransmission delay usually requires that new credentials be obtained.

3.3 AMS Steps in GAP

1. Upon receipt of a request, the **AMS** determines if a credentials structure was passed.

2. If none was passed. **AMS** either initializes the credentials structure with **credentials.cred_len** and **credentials.cred_data** both set to zero or passes a null pointer instead of a pointer to a structure.

3. If credentials were passed, **AMS** recreates the credentials structure as it existed at the **OSCK** after the ***GetCred_D()** or ***GetCred_M()** call.

4. A pointer to the credentials structure, which may be null, is passed to the appropriate **oofs()** routine.

5. After the **AMS** completes all **oofs()** calls, it can free the credentials structure in the appropriate way.

3.4.1 oofs_Register_Security()

```
oofsStatus oofs_Register_Security(oofsSecRoutines *Sec_List); // In
```

Function

Register security callback functions.

Parameters

Sec_List

is the list of routines to be used to provide security. Supplying a NULL address for the routine indicates unused routines.

Success

Zero is returned.

Failure

A negative one (-1) is returned.

Notes

4) The **oofs_Register_Security()** routine is part of the **OCSK**. It must be called by the application, prior to opening a federated database, in order to register the security interface.
5) The security interfaces are defined in subsequent sections.

3.4.1.1 *CreateSec()

```
oofsSecHandle (*CreateSec)(const ooHandle(ooDDObj) &odH, // In
                           oofsErrorStruct &einfo);     // Out
```

Function

Create a security context for a database.

Parameters

odH

is the database handle that would have been returned by **ooRefHandle(ooDBObj)::open()**.

einfo

is the error information structure used when an error occurs.

Success

A non-zero opaque handle to the security context is returned.

Failure

A NULL (i.e., zero) pointer is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

Notes

6) A zero for ***CreateSec()** in the **oofsSecRoutinesDesc** structure suppresses the call to this routine. A null context handle is then used for all subsequent database calls that require the associated security handle.
7) ***CreateSec()** is called whenever **ooRefHandle(ooDBObj)::open()** is effectively called to open a database.

3.4.1.2 *DeleteSec()

```
oofsStatus (*DeleteSec)(oofsSecHandle secHandle,        // In
                        oofsErrorStruct &einfo);        // Out
```

Function

Delete a security context for a database.

Parameters

secHandle

is the handle returned by ***CreateSec()** for the database corresponding to the one that now being closed.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

Notes

1) A zero for ***DeleteSec()** in the **oofsSecRoutinesDesc** structure suppresses the call to this routine.
2) ***DeleteSec()**is called whenever **ooRefHandle(ooDBObj)::close()** is effectively called to close a database.

3.4.1.3 *GetCred_D()

```
oofsCredStruct *(*GetCred_D)(oofsSecHandle secHandle,    // In
                             const char *buffer,         // In
                             oofsXferSize buffer_size,   // In
                             oofsErrorStruct &einfo);    // Out
```

Function

Obtain credentials for a data operation.

Parameters

secHandle

is the handle returned by ***CreateSec()** for the database corresponding to the one that now requires data operation credentials.

buffer

points to the buffer holding the data that will be sent to the **AMS**. If no data is being sent or if data is being read from the **AMS**, the pointer is null.

buffer_size

contains the number of valid bytes in the buffer. If no data is being sent or if data is being read from the **AMS**, the value is zero.

einfo

is the error information structure used when an error occurs.

Success

A non-zero pointer to a credentials structure is returned. The data in the structure is sent to the **AMS** along with the request.

Failure

A NULL (i.e., zero) pointer is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

Notes

1) A zero for ***GetCred_D()** in the **oofsSecRoutinesDesc** structure suppresses the call to this routine. Null credentials are then used for the data operation.
2) ***GetCred_D()** is called for the following AMS data operations:

```
oofs_read()          oofs_sync()          oofs_write()
oofs_readdir()       oofs_truncate())
```

3.4.1.4 *GetCred_M()

```
oofsCredStruct *(*GetCred_M)(oofsSecHandle secHandle,    // In
                             oofsErrorStruct &einfo);    // Out
```

Function

Obtain credentials for a meta-data operation.

Parameters

secHandle

is the handle returned by ***CreateSec()** for the database corresponding to the one that now requires meta-data operation credentials.

einfo

is the error information structure used when an error occurs.

Success

A non-zero pointer to a credentials structure is returned. The data in the structure is sent to the **AMS**.

Failure

A NULL (i.e., zero) pointer is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

Notes

1) A zero for ***GetCredOp()** in the **oofsSecRoutinesDesc** structure suppresses the call to this routine. Null credentials are context are then used for the meta-data operation.
2) ***GetCredOp()** is called for the following **AMS** meta-data operations:

`oofs_close()`	`oofs_getsize()`	`oofs_opendir()`
`oofs_closedir()`	`oofs_getmode()`	`oofs_remove()`
`oofs_exists()`	`oofs_open()`	`oofs_rename()`

3.4.1.5 *AuthCred()

```
oofsStatus (*AuthCred)(oofsSecHandle secHandle,          // In
                       oofsCredStruct &server_cred,      // In
                       const char *buffer,               // In
                       oofsXferSize buffer_size,         // In
                       oofsErrorStruct &einfo);          // Out
```

Function

Create a security context for a database.

Parameters

secHandle

is the handle returned by ***CreateSec()** for the database associated with the server interaction.

server_cred

are the credentials sent by the server.

buffer

points to the buffer holding the data that will be sent to the **AMS**. If no data is being sent or if data is being read from the **AMS**, the pointer is null.

buffer_size

contains the number of valid bytes in the buffer. If no data is being sent or if data is being read from the **AMS**, the value is zero.

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

Notes

1) A zero for ***AuthCred()** in the **oofsSecRoutinesDesc** structure suppresses the call to this routine
2) ***AuthCred()** ***is not currently supported*** and will not be called if it is supplied.

3.4.1.6 *Crypt()

```
oofsStatus (*Crypt)(oofsSecHandle secHandle,            // In
                    const ooInt direction,              // In
                    const char *buffer,                 // In
                    oofsXferSize buffer_size,           // In
                    char **out_buffer,                  // Out
                    oofsXferSize *out_buffsize,         // Out
                    oofsErrorStruct &einfo);            // Out
```

Function

Encrypt or decrypt a data buffer.

Parameters

secHandle

is the handle returned by ***CreateSec()** for the database associated with the data buffer.

direction

describes the operation to be performed:

`OOFS_Seal` - encrypt the data in the buffer

`OOFS_UnSeal` - decrypt the data in the buffer

buffer

a pointer to a buffer that holds data either to be sent or received from an **AMS**. If **`OOFS_Seal`** is set, the buffer holds the data that will be sent to the **AMS**. If **`OOFS_UnSeal`** is set, the buffer holds the data that was received from the **AMS**.

buffer_size

contains the number of valid bytes in the buffer

out_buffer

upon success return, will hold a pointer to a buffer containing encrypted or decrypted data, depending on direction.

outbuff_size

contains the number of valid bytes in the out_buffer

einfo

is the error information structure used when an error occurs.

Success

Zero (0) is returned.

Failure

OOFS_ERROR is returned and **einfo.code** contains the actual error code while **einfo.message** contains an optional null terminated string describing the nature of the error.

Notes

1) ***Crypt()** is called with **oofs_Seal** whenever data is sent to the **AMS**. This allows all networked data to be encrypted.
2) ***Crypt()** is called with **oofs_UnSeal** whenever encrypted data is received from the **AMS**.
3) The out_buffer must be released by a call to ***FreeBuff()**.
4) A zero for ***AuthCred()** in the **oofsSecRoutinesDesc** structure suppresses the call to this routine
5) ***Crypt()** ***is not currently supported*** and will not be called if it is supplied.

3.4.1.7 *FreeBuff()

```
void (*FreeBuff)(char *buffer);       // In
```

Function

Release a buffer allocated by ***Crypt()**.

Parameters

buffer

points to the buffer that is to be released.

Success

n/a

Failure

n/a

Notes

1. A zero for ***FreeBuff()** in the **oofsSecRoutinesDesc** structure suppresses the call to this routine. This is not recommended if ***Crypt()** is defined.
2. ***FreeBuff()** ***is not currently supported*** and will not be called if it is supplied.

4 Opaque Information Protocol

Opaque Information Protocol (**OIP**) allows arbitrary information (e.g., performance hints) to be transparently transferred from an Objectivity **AMS** client to the **AMS** server. **OIP** is supported only for **AMS** mediated connections. It consists of one new client-side function, **oofs_set_info()**. Opaque information is sent to the **oofs_open()** function on the **AMS** side.

4.1 Application Steps in OIP

1. The application program must initialize a character array[2] with information appropriate to the **AMS**-side filesystem being used, along with an **oofsInfoStruct** structure (i.e., a pointer to the character array must be set in the **oofsInfoStruct** structure along with the effective length of the array).

2. The application program must then call **oofs_set_info()**, passing it oofsInfoStruct, prior to opening the database(s) associated with the information. The application has an option of associated a single (**OOFS_SOI_ONCE**) database open with the information or all subsequent opens (**OOFS_SOI_ALWAYS**), using the flags (i.e., second) parameter. An example follows.

```
char info[256];
struct oofsInfoStruct InfoStruct;
•
• // Fill in the info array, as needed
•
InfoStruct.info_len  = sizeof(info);
InfoStruct.indo_data = info;
if (oofs_set_info(InfoStruct, OOFS_SOI_ONCE))
   Fatal_Error();
•
• // Open objectivity database, sending it the info.
•
```

[2] Any kind o structure may be used. However, since the information is considered as opaque by the **OCSK**, it is treated as a character array.

3. Information set by **oofs_set_info()** is used only for the first explicit database open when **`OOFS_SOI_ONCE`** is passed. Otherwise, the same information is used for all subsequent database opens.

4. The information is considered current until replaced by a another call to **oofs_set_info()** (i.e., the supplied information is copied to an internal buffer). Any number of calls may be made to **oofs_set_info()** with information not exceeding OOFS_MAX_INFO_LEN bytes.

4.2 OCSK Steps in OIP

1. Opaque information is only support for open operations, regardless of cause.

2. Prior to sending an **open()** request to **AMS**, **OCSK** checks to see if any **oofs_set_info()** information exists (i.e., an **oofsInfoStruct** exists and has a length is greater than zero with a non-null pointer to a character string).

3. If opaque information exists, the length and data are marshaled as (int), and (char), respectively, on to the **AMS** protocol stream in a suitable way.

4. Opaque information is passed to the **AMS** on any request that would result in an **AMS oofs_open()** call, regardless of the reasons for that call.

5. If **`OOFS_SOI_ONCE`** is associated with the opaque information, the information is discarded after it has been sent (i.e., it is only sent once). Otherwise, an internal copy is maintained for subsequent transmission.

4.3 AMS Steps in OIP

1. Upon receipt of a request, the **AMS** determines if **oofs_set_info()** information was passed.

2. If any was passed, it recreates an **oofsInfoStruct** structure with that information and passes it to the requested **oofs_open()** call(s) using the fourth parameter. Otherwise, a null pointer is passed as the fourth parameter.

3. Opaque information is passed to every **oofs_open()** initiated by the incoming request.

4. After the **AMS** completes all **oofs_open()** calls, it discards the information.

4.3.1 oofs_set_info()

```
oofsStatus oofs_set_info(oofsInfoStruct &InfoStruct,     // In
                         const ooInt flags);,            // In
```

Function

Set opaque information for subsequent transmission to the **AMS**.

Parameters

InfoStruct

is a reference to the structure that contains the length of and pointer to the opaque information. If either the pointer is NULL, **`InfoStruct.info_len`** contains zero, or **`InfoStruct.info_data`** is NULL, the current opaque information, if any, is discarded and no information is subsequently sent.

flags

indicates how the opaque information is to be handled:

`OOFS_SOI_ALWAYS` - always send the information on subsequent open requests.

`OOFS_SOI_ONCE` - send the information on the subsequent open request and discard the information afterwards (i.e., only send it once).

Success

Zero is returned.

Failure

A negative one (-1) is returned.

Notes

1) The **oofs_set_info()** routine is part of the **OCSK**. It must be called by the application in order to establish the information that is to be sent.
2) The **oofs_set_info()** routine may be called at any time. A new invocation simply replaces opaque information established by the previous call, if any.
3) Opaque information is only supplied to the **AMS**-side **oofs_open()** routine.

5 Defer Request Protocol

Defer Request Protocol (**DRP**) allows an **AMS** server to control the timeout of a client's request. It is included as part of the **AMS/OCSK** protocol to allow the use of hierarchical filesystems with highly variable latencies (e.g., microseconds to several minutes).

5.1 AMS Steps in DRP

1. At the end of any failing **oofs()** operation, the **oofsErrorStruct** parameter contains the reason for the failure, represented as an error code and an optional ASCII character string. Failure indication is defined by each **oofs()** function as follows:

Function	Failure Return	Function	Failure Return
`oofs_close()`	`-1`	`oofs_read()`	`-1`
`oofs_closedir()`	`-1`	`oofs_readdir()`	`-1`
`oofs_exists()`	`-1`	`oofs_remove()`	`-1`
`oofs_getmode()`	`-1`	`oofs_rename()`	`-1`
`oofs_getsize()`	`-1`	`oofs_sync()`	`-1`
`oofs_open()`	`0`	`oofs_truncate()`	`-1`
`oofs_opendir()`	`0`	`oofs_write()`	`-1`

2 If the **oofsErrorStruct.code** is zero and the text string in **oofsErrorStruct.message** starts with the string "**!wait**", then the function is requesting that the client retry the operation after a certain amount of time. The number of seconds to wait follows **!wait**, separated by a single space. For example:

   !wait 300

   indicates that the client should wait 300 seconds then reissue the *same* request to the *same* **AMS**.

3. The server uses this information to provide the appropriate indication to the client. The client must retry the request on the same **AMS** after waiting the specified number of seconds.

5.2 OCSK Steps in DRP

1. The **OCSK** must be prepared to receive a retry request following any interaction with the **AMS**. Therefore, all data sent to the **AMS** must be kept until an actual indication of success or failure is received.

2. Should a retry request be received, the **OCSK** must wait the specified number of seconds and then send the same request to the same **AMS**.

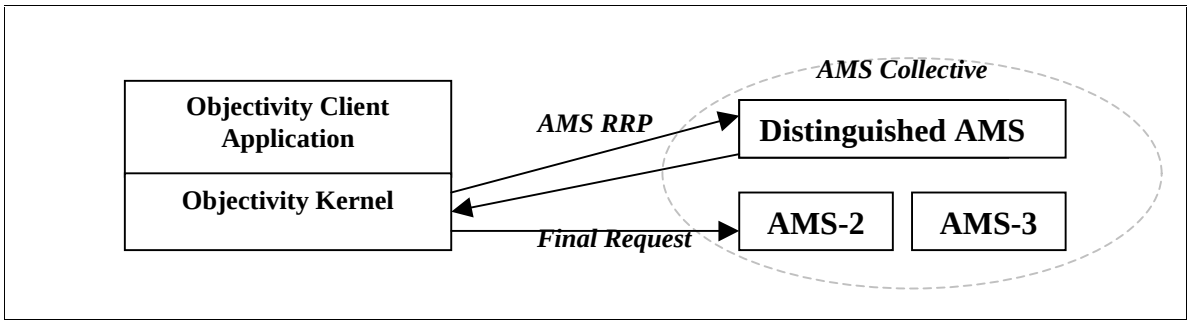
3. The **OCSK** should treat the retry request as an error if any of the following occurs:
   a) The number of seconds is less than 1.
   b) The number of seconds is greater than 9999.
   c) The number of seconds is not a whole number.

4. Retry errors are to be treated as **AMS** errors and standard error recovery procedures are to be applied.

6 Request Redirection Protocol

Request Redirection Protocol (**RRP**) allows cooperating **AMS**'s to perform dynamic load balancing using a simple redirection scheme. A group of cooperating **AMS**'s, or an **AMS** Collective (**AMSC**), is logically treated as a single **AMS** by the client. There is one distinguished member of the collective and all database catalogue information is tied to the distinguished member (i.e., server). This minimizes the administrative impact on client operations and database descriptions. Therefore, a client need not know the composition of the collective and, indeed, that composition is free to change at any time without impacting any database operations that the client may perform.

The following figure illustrates an AMS collective and a simple redirection interaction.



Redirection allows an **AMS** to direct a client to a more suitable **AMS** for processing the client's request. Because the server redirects the request, it is possible to implement a highly flexible and scalable dynamic load-balancing scheme than would otherwise be possible by other mechanisms. For instance, the database load may be balanced using one or more of the following criteria, among others:

- Number of clients,
- Available real memory,
- Available disk space,
- Network link performance, and
- Service level agreements.

RRP also allows the **AMS** to dynamically replicate databases on demand. For instance, should a databases become highly used, the **AMS** could replicate the database at one or more other sites and redirect clients to other copies. Once the database becomes less heavily used, the extra copies can be eliminated to save space.

Dynamic load balancing does not alter any current multi-**AMS** protocols such as the Data Replication Option (**DRO**) or the Fault Tolerant Option (**FTO**). This is because the collective is defined outside the scope of such protocols. Furthermore, each **AMSC** member is effectively interchangeable in the context of the collective. Nevertheless, it is possible to implement collectives in ways that render **DRO** and **FTO** unusable.

The collective is responsible for providing update synchronization should any **AMSC** members allow write operations. Should a client wish to update a database and a modifiable database replica exists outside of the collective, it is the application's responsibility to explicitly indicate that write operations will occur before performing any transactions against the database.

Redirection of requests is only supported for operations against a closed database (i.e., operations that use a filename instead of a filehandle). This includes the **open()** and **opendir()** requests.

6.1 AMS Steps in RRP

1. At the end of any failing **oofs()** operation that takes a filename as an argument, the **oofsErrorStruct** parameter contains the reason for the failure, represented as an error code and an optional ASCII character string. Failure indication is defined by each applicable **oofs()** function as follows:

Function	Failure Return	Function	Failure Return
`oofs_exists()`	`-1`	`oofs_remove()`	`-1`
`oofs_open()`	`0`	`oofs_rename()`	`-1`
`oofs_opendir()`	`0`		

2 If the **oofsErrorStruct.code** is zero and the text string in **oofsErrorStruct.message** starts with the string "**!try**", then the function is requesting that the client retry the operation at another AMS. The location of the alternate **AMS** follows **!try**, separated by a single space, and is either the DNS name or the IP address of the alternate AMS. For example:

!try abh.slac.stanford.edu

indicates that the client should reissue the *same* request to the **AMS** located on host "**abh.slac.stanford.edu**".

3. The server may use this information to provide the appropriate indication to the client or may simply pass the error text-string to the client for resolution. The mechanism actually used is immaterial as long as the redirection occurs.

6.2 OCSK Steps in RRP

1. The **OCSK** must be prepared to receive a redirect request following any interaction with the **AMS**. Therefore, all data sent to the **AMS** must be kept until an actual indication of success or failure is received.

2. Should a redirect request be received, the **OCSK** must send the same request to the specified **AMS**.

3. The **OCSK** should treat the redirect request as an error if any of the following occurs:
   a) The redirect specifies the same **AMS**.
   b) More than 255 consecutive redirect requests have been received.
   c) The specified **AMS** does not exist or refuses a connection.

4. Redirection errors are to be treated as **AMS** errors and standard error recovery procedures are to be applied.