CCDS User's Guide*

Soeren P. Olesen
Vaidy S. Sunderam

Department of Mathematics and Computer Science
Emory University, Atlanta, GA 30322, USA
{olesen,vss}@mathcs.emory.edu

January 7, 1998

*Research supported in part by NSF grant ASC-9527186

Contents

1 Introduction

The Collaborative Computing Data Space (CCDS) implements a shared object space for collaborative computing systems.

With CCDS application programs access shared objects through familiar programming interfaces.

This user's guide contains an overview of the CCDS architecture, information on how to program with CCDS, programming examples, and details of the CCDS programming library.

2 Architecture

The CCDS architecture (fig. 1) is based on a set of distributed data servers known as `ccds_s`. Together, the data servers implement a global name space for objects in the data space. Each data server serves as a centralized server for the objects on that server; hence, concurrency control is simply achieved by serializing requests to a centralized server. A data server is instantiated on any machine where one or more clients is accessing the data space. Several reasons motivated us to chose this hybrid model; using a centralized scheme immediately implies a simple and robust concurrency model, as well as simplifying other operations, such as latecomer support. Furthermore, with distributed servers, locality of access can be exploited which means that data objects can be located nearest, in terms of communication cost, to where the object is accessed most often. The distributed servers can also serve as a framework for a future extension where data objects are fragmented across servers. However, for a tightly coupled synchronous access pattern where many clients access one object, a replicated architecture would be more appropriate.

From a clients perspective the physical location of an object is transparent and objects can migrate from one server to another in order to achieve better performance. A client library handles client communication with data servers and provides applications with a simple programming interface.

The data servers communicate on a well known *server channel* to maintain a consistent global naming space, data servers also join a well known *data space channel*. When a client joins the data space, it joins the *data space channel*. The underlying communication functionality for reliable and atomic

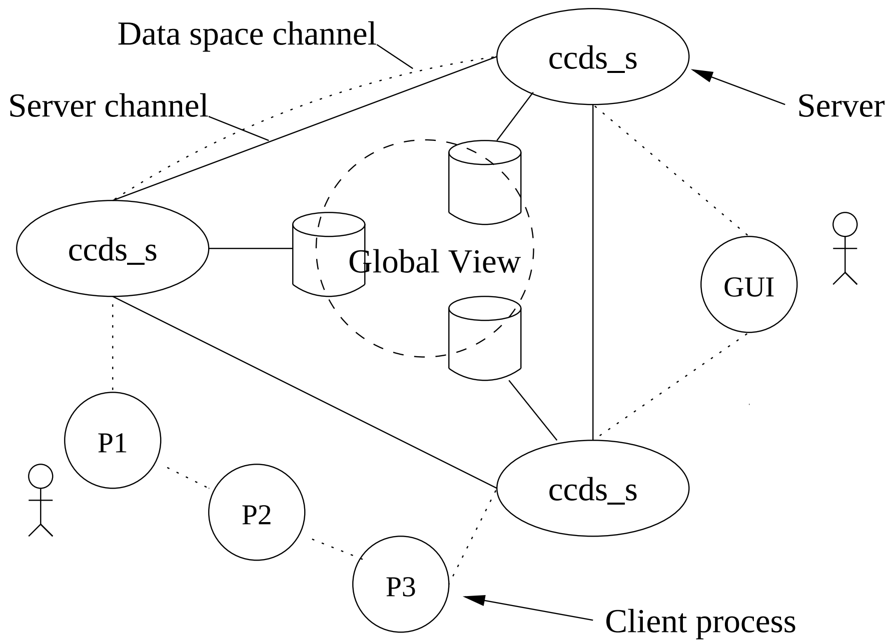

Figure 1: CCDS architecture.

communication semantics is handled by the Collaborative Communication Transport Layer (CCTL) [1]. The CCTL also provide whitepage services for naming and membership operations. To access an object initially, for example to open a data file, the client library broadcasts a request on the *data space channel*, every data server then responds with a positive or negative reply. From then on, all accesses to the object occur in point to point mode with the particular server that serves the object.

The objects on a particular data server are physically stored on a storage medium that is accessible from the data server via the operating system. Ideally, for best performance, the storage is local to the machine where the data server is running.

The current implementation of CCDS supports a file object type. Operations provided by the CCDS programming interface for the file object type include standard file operations known from operating systems, such as: open, close, read, write, segment lock, unlock, and seek. In addition to these operations, the API includes calls for joining and leaving the CCDS, for importing a file object from a local filesystem to CCDS, conversely, unimporting from CCDS to local filesystem, calls for listing of CCDS objects, as well as other management functions. CCDS also has functionality for migrating an object from a remote server to a local server. With a local server we refer to a server running on the same machine as the client. The advantage of migrating an object to a local server is lower latency and higher throughput resulting from less communication.

In addition to the familiar single process file object access semantics, CCDS facilitates collaborative access to CCDS file objects through several shared file offset pointers. A shared file pointer is a file pointer that is shared among several processes that have the file object opened, pointer update semantics guarantee that every process has the same view of the shared file pointer. Currently, one shared read pointer and one shared writer pointer per file are supported. In addition to shared pointers, a process can open a file object with a local file pointer that is accessible only to that particular process.

Clients can request asynchronous notifications of data space content changes from the CCDS data servers. Notification events are passed to a client via a callback function implemented by the client. Currently, notification events are generated when a new object is created in the data space, or when a object is removed from the data space. We use the notification events to implement a graphical user interface client for the CCDS that displays the data space content, and which also has functions for importing, unimporting, and viewing objects.

3 Programming with CCDS

The following components are needed to create and run applications with CCDS:

- CCDS library (`libccds.a`)
- CCDS server binary (`ccds_s`)
- CCTL library (`libcctl.a`)
- CCTL whitepage server (`wp`)

CCDS requires that the following environment variables are set:

- `CCF_BIN`: path to `ccds_s` binary. Example:
  `CCF_BIN=/us-home/ccf/TEST/bin/SunOS-5.6`
- `CCTL_WP_HOST_NAME`: name of machine where `wp` is running. Example:
  `CCTL_WP_HOST_NAME=emily.mathcs.emory.edu`

CCDS data spaces are identified by a *session name*. A client that wishes to access a particular data space does so by calling the CCDS library function `ccds_init` with a session name as a parameter. During initialization CCDS searches for a `/tmp/ccds_server_file_info.[session name]` file. If this file exists, then a `ccds_s` server is already running on the machine for that session; otherwise, if the file is not found, a `ccds_s` server is spawned. A client can only access one data space at a time; however, two or more clients can run on the same machine and access different data spaces.

By default CCDS uses the `/tmp` path for permanent storage. For each session a `/tmp/[session name]` directory is created for that session. There is several ways to override the default value, listed here in order of selection:

1. provide path as option to `ccds_init` call
2. provide path in `.ccfrc` file in users home directory
3. set optional environment variable: `CCF_CCDSPATH`

Clients should call `ccds_leave` before terminating. This ensures that resources are freed and also lets the `ccds_s` servers terminate when no clients are accessing the data space.

4 Notes

If a client is killed or exits before calling `ccds_leave`, then the `ccds_s` server on that machine will not terminate. In this case the user has to manually kill `ccds_s`, remove the `/tmp/ccds_server_file_info.[session name]` file, and the `/tmp/[session name]` directory.

5 Examples

Programming with CCDS consists of issuing calls to the CCDS programming library functions. The following example show how to get started.

The program in figure 2 accesses the data space given by the *-session* argument in *argv[]*, then opens the file *myfile* for reading and writing with a local file offset pointer. If *myfile* does not exist in the data space, then it is created. See also `ccds_open` in appendix A. The first 11 bytes segment of *myfile* is locked, and a message is written. Finally, *myfile* is unimported to a file named *test*.

An example makefile to make this program is shown in figure 3.

The program in figure 4 shows how to enable the notification feature of CCDS.

References

[1] I. Rhee, S. Y. Cheung, P. Hutto, and V. Sunderam, "Group communication support for distributed multimedia and CSCW systems," in *Proc. 17th ICDCS*, (Baltimore, MD), May 1997.

```
/* This test program first accesses the CCDS data space given
 * on the commandline, then opens a file (file is created if
 * it doesn't exist), locks a segment, and writes a message
*/
#include <stdio.h>
#include <ccds.h>
#include <ccds_errno.h>
void main( int argc, char *argv[] )
{
     int retval;
     char message[ 11 ] = "Hello World";
     int fdes;

     if( ( retval = ccds_init( argc, argv ) ) != CCDS_OK ) {
             fprintf( stderr, "INIT Error: %d\n", retval );
             exit(1);
     }
     if( ( fdes = ccds_open( "myfile",
                         CCDS_RDWR|CCDS_CREAT,
                         0,
                         CCDS_LOCAL,
                         CCDS_UNRESTR ) ) < CCDS_OK ) {
             fprintf( stderr, "OPEN Error: %d \n", fdes );
             ccds_leave();
             exit(1);
     }
     if( ( retval = ccds_lock( fdes, 11 ) ) != CCDS_OK )
             fprintf( stderr, "LOCK Error: %d\n", retval );
     if( ( retval = ccds_write( fdes, message, 11) ) != 11 )
             fprintf( stderr, "WRITE Error: %d\n", retval );
     ccds_close( fdes );
     if( ( retval = ccds_unimport( "test", "myfile" ) ) != CCDS_OK)
     fprintf( stderr, "UNIMPORT Error: %d\n", retval );
     ccds_leave();
}
```

Figure 2: Example CCDS C program

```
LINKLIBS = -L/us-home/ccf/TEST/lib/SunOS-5.6 -lccds -lcctl \
           -R/usr/ucblib -L/usr/ucblib -lpthread -lnsl -lsocket \
           -lucb -lposix4  -lm
INCLUDE = -I/us-home/ccf/TEST/include -I.

CFLAGS = -g -DSOLARIS $(INCLUDE)
CC = cc

example: example.o
	$(CC) -o example  example.o $(LINKLIBS)

example.o: example.c
	$(CC) $(CFLAGS) -c example.c

clean:
	rm -f *.o example
```

Figure 3: Example Makefile

```c
/*  This program enables a notification call back routine
 *  which prints a message everytime an object is added or
 *  deleted
*/
#include <stdio.h>
#include <ccds.h>
#include <ccds_errno.h>
static void notify_cb( ccds_item_t *msg, int nitem );

void main( int argc, char *argv[] )
{
     int retval;

     if( ( retval = ccds_init( argc, argv ) ) != CCDS_OK ) {
             fprintf( stderr, "INIT Error: %d\n", retval );
             exit(1);
     }
     ccds_notify( notify_cb );
     getchar();
     ccds_leave();
}

static void notify_cb( ccds_item_t *msg, int nitem )
{
     ccds_item_t *ptr;
     int i;

     ptr = msg;
     for( i = 0; i < nitem; i++, ptr++) {
          if( ptr->cflag == CCDS_OBJECT_ADD )
               fprintf( stdout, "%s is added \n", ptr->name );
          else if( ptr->cflag == CCDS_OBJECT_DELETE )
               fprintf( stdout, "%s is deleted \n", ptr->name );
     }
}
```

Figure 4: Example CCDS C program

A CCDS Programming Interface

The following pages contain documentation of the CCDS programming interface functions in alphabetical order.

NAME

ccds_close - close file descriptor

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_close(int fdes);

DESCRIPTION

ccds_close() closes the file descriptor specified by *fdes*.

RETURN VALUES

Upon successful completion, a value of 0 is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned:

CCDS_EFSTM
: error in underlying file system

CCDS_BADF
: not an open file descriptor

CCDS_INVID
: not owner of file descriptor

CCDS_ETPORT
: error in underlying transport system

NAME

ccds_delete - delete a CCDS object

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

void ccds_delete(char *name);

DESCRIPTION

ccds_delete() deletes the CCDS object pointed to by *name*. The object is not actually removed from the CCDS until it has been closed by every process that has opened it. Only the owner of an object can successfully delete an object.

NAME

ccds_export - export an object from CCDS

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_export(char *dstname, char *name);

DESCRIPTION

ccds_export() moves the object pointed to by *name* to the file pointed to by *dstname*.

RETURN VALUES

Upon successful completion, a value of 0 is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_EACCES
: File not found or can not create

CCDS_EFTBL
: server file table full

CCDS_ECFTBL
: client file table full

CCDS_EFSTM
: error in underlying file system

CCDS_ETPORT
: error in underlying transport system

CCDS_ELOCK
: section is locked

CCDS_ESRC
: can't open src

CCDS_ENAME
: bad name

NAME

ccds_import - import an object to CCDS

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_import(char *srcname, char *name, int mode, int type);

DESCRIPTION

ccds_import() imports the object pointed to by *srcname* into CCDS where it is reference to by the name pointed to by *name*. The *mode* argument specifies the access permission bits for the object and is determined as the bitwise OR of: CCDS_IR, CCDS_IW, CCDS_IX, giving read, write, and execute permission respectively for the file to all members of the CCDS. Permissions are currently not supported by CCDS.

Only *type* argument currently supported is

CCDS_NATIVEFILE
: File is accessed via CCDS

RETURN VALUES

Upon successful completion, a value of 0 is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_EACCES
: File not found or can not create

CCDS_EFTBL
: server file table full

CCDS_ECFTBL
: client file table full

CCDS_EFSTM
: error in underlying file system

CCDS_ETPORT
: error in underlying transport system

CCDS_EINSUF
: insufficient system resources

CCDS_ESRC
: can't open src

CCDS_ENAME
: bad name

NAME

ccds_init - initialize ccds

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_init(int argc, char **argv);

DESCRIPTION

ccds_init() performs various initialization functions. *argv* is a pointer to a string of options, *argv* must contain a *-session* option followed by a *session name*. ccds_init() accepts optional *-path* option followed by *path* to local data storage and *-debug* option followed by debug level 0 – 6. Default path is /tmp and default debug level is 0 (no debug output). ccds_init() spawns a data server on the local machine if a server in the named session is not already active.

RETURN VALUES

Upon successful completion, a value of 0 is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_FATAL
: fatal system error

CCDS_ETPORT
: error in underlying transport system

NAME

ccds_leave - leave ccds

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_leave( );

DESCRIPTION

ccds_leave() causes the process to leave the ccds. All open files are closed and all objects owned by the calling process on data servers are deleted. The ccds data servers will terminate when ccds is empty and all processes have called ccds_leave().

RETURN VALUES

Upon successful completion, a value of 0 is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_ERROR
: ccds error

NAME

ccds_lock - lock segment of file

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_lock(int fdes, long size);

int ccds_olock(int fdes, ccds_offt offset, long size);

DESCRIPTION

ccds_lock() allows a segment of the file associated with *fdes* to be locked. Only one lock can be associated with each *fdes*.

ccds_lock() locks *size* bytes from the current file offset pointer position. If part of that segment is already locked, then ccds_lock() returns with an error. The file pointer remains unchanged.

ccds_olock() performs the same action as ccds_lock(), but begins locking from *offset* file position.

RETURN VALUES

Upon successful completion, a value of 0 is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_BADF
: not an open file descriptor

CCDS_INVID
: not owner of file descriptor

CCDS_ELOCK
: section is locked

CCDS_ENLOCK
: number of locks exceeded

NAME

ccds_ls - list content of CCDS

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_ls(ccds_item_t **buf, int *nitem);

DESCRIPTION

ccds_ls() returns a list of content of CCDS. Upon return from ccds_ls() **buf* points to an array of *nitem* items of type ccds_item_t. ccds_ls() allocates memory for the buffer that should be freed by the calling process.

RETURN VALUES

Upon successful completion, a value of 0 is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_ETPORT
: error in underlying transport system

NAME

ccds_lseek - move read/write file pointer

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

ccds_offt ccds_lseek(int fdes, ccds_offt offset, int whence);

DESCRIPTION

ccds_lseek() sets the file pointer associated with *fdes*.

The *whence* argument determines ccds_lseek operation as follows:

CCDS_SEEK_SET
: the pointer is set to *offset* bytes

CCDS_SEEK_CUR
: the pointer is set to its current position plus *offset*

RETURN VALUES

Upon successful completion, the resulting non-negative file pointer is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_BADF
: not an open file descriptor

CCDS_INVID
: not owner of file descriptor

CCDS_ELOCK
: section is locked

CCDS_ETPORT
: error in underlying transport system

NAME

ccds_migrate - Migrate object to local data server

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_migrate(char *name);

DESCRIPTION

ccds_migrate moves the object pointed to by *name* to the local data server for the calling process. Currently an object has to be closed before it can migrate.

RETURN VALUES

Upon successful completion, a value of 0 is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_EACCES
: File not found or can not create

CCDS_EFSTM
: error in underlying file system

CCDS_ETPORT
: error in underlying transport system

CCDS_ERROR
: ccds error

NAME

ccds_notify - enable or disable object notification change callback

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

void ccds_notify(void (*cb)(ccds_item_t *buf, int nitem));

DESCRIPTION

ccds_notify() enables a callback for object notification changes. Three types of object changes will result in a call of *cb*: (1) addition of object to CCDS, (2) deletion of object from CCDS, and (3) change of object properties. Currently only (1) and (2) are implemented. *cb* is a pointer to a user supplied callback function. *cb* has two parameters *buf* and *nitem*.

ccds_item_t buf

pointer to an array of elements. *buf* is freed after returning from *cb*.

nitem

number of elements in *buf*

The type *ccds_item_t* is defined as:

```
typedef struct {
                char name[ CCDS_MAX_OBJECTNAME ];
                int type;
                int owner_id;
                int cflag;
        } ccds_item_t;
```

Where the fields have the following semantics

char name[]

name of object

int type

type of object, currently only CCDS_NATIVEFILE is supported

int owner_id

unique id of owner

int cflag

change flag, can take the values of CCDS_OBJECT_ADD and CCDS_OBJECT_DELETE signifying that the object is added, respectively deleted.

When notification is enabled, *cb* will be called to reflect the current global state of CCDS.

To disable notification use ccds_notify(NULL);

NAME

ccds_open - open a file for reading or writing

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_open(char *name, int oflag, int mode, int view, int lflag);

DESCRIPTION

ccds_open() opens a file descriptor for the file pointed to by *name*. *oflag* is determined by bitwise OR of one of:

CCDS_RDONLY
: open for reading only, and select shared read pointer in shared filepointer operation. See *view* parameter.

CCDS_WRONLY
: open for writing only, and select shared write pointer in shared filepointer operation. See *view* parameter.

CCDS_RDWR
: open for reading and writing, and select shared write pointer in shared filepointer operation. See *view* parameter.

and any combination of:

CCDS_APPEND
: file pointer is set to end of file prior to each write.

CCDS_CREAT
: if the file does not exist, then it is created with access permission set to *mode*.

CCDS_CREATONLY
: same as CCDS_CREATE, but if file already exists then an error value is returned.

The *mode* argument specifies the access permission bits when a file is created, and is determined as the bitwise OR of: CCDS_IR, CCDS_IW, CCDS_IX, giving read, write, and execute permission respectively for the file to all members of the CCDS. The *mode* argument and permissions are currently not supported by CCDS.

The argument *view* sets one of the following file views:

CCDS_SHARED

A single file pointer is shared among all processes that open the file with the same *oflag* parameter. CCDS allows two different shared file pointers per file. One shared file pointer is assigned to processes that open with CCDS_SHARED and *oflag* parameter CCDS_RDONLY set. The second shared file pointer is assigned to processes that open with CCDS_SHARED and *oflag* parameter CCDS_WRONLY or CCDS_RDWR set.

CCDS_LOCAL

The file is accessed via a local file pointer.

Permission to open the file with different oflag and view parameters is controlled by the *lflag* parameter. *lflag* can take one of the following values:

CCDS_UNRESTR

File can be opened by any combination of *oflag* and *view* parameters.

CCDS_RESTR

File can only be opened with one shared pointer. This means that all shared opens must have the same *oflag* parameter. CCDS_RESTR is currently not supported.

CCDS_EXCLUV

File can not be opened by other processes.

NOTE

A file descriptor can not be shared with another process.

RETURN VALUES

Upon successful completion, a file descriptor is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_EACCES

File not found or can not create

CCDS_EFTBL
: server file table full

CCDS_ECFTBL
: client file table full

CCDS_EFSTM
: error in underlying file system

CCDS_ETPORT
: error in underlying transport system

CCDS_ELFLAG
: file cannot be opened with lflag

CCDS_EXIST
: file already exists

CCDS_ENAME
: bad name

NAME

ccds_property - properties of an CCDS object

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_property(char *name, ccds_propt *buf);

DESCRIPTION

ccds_property() returns the object properties for the CCDS object pointed to by *name*.

RETURN VALUES

Upon successful completion, a value of 0 is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_EACCES
: File not found or can not create

CCDS_ETPORT
: error in underlying transport system

CCDS_ENAME
: bad name

NAME

ccds_read - read from a file

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

ccds_ssizet ccds_read(int fdes, char *buf, ccds_sizet nbyte);

ccds_ssizet ccds_oread(int fdes, char *buf, ccds_sizet nbyte, ccds_offt offset);

ccds_ssizet ccds_pread(int fdes, char *buf, ccds_sizet nbyte, ccds_offt offset);

DESCRIPTION

ccds_read() reads *nbyte* bytes from the file specified in *fdes*, starting at the current file pointer position, into the buffer pointed to by *buf*. ccds_read() stops reading bytes if an end-of-file or error condition is encountered. The file pointer is incremented by the number of bytes read.

ccds_oread() performs the same action as ccds_read(), except that the starting offset is specified by the *offset* parameter.

ccds_pread() performs the same action as ccds_oread(), except that the file pointer associated with fdes is unchanged.

RETURN VALUES

Upon successful completion, the number of bytes read is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_EFSTM
: error in underlying file system

CCDS_BADF
: not an open file descriptor

CCDS_INVID
: not owner of file descriptor

CCDS_ETPORT
: error in underlying transport system

CCDS_EINSUF
: insufficient system resources

CCDS_ELOCK
: section is locked

NAME

ccds_unimport - unimport an object from CCDS

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_unimport(char *dstname, char *name);

DESCRIPTION

ccds_unimport() moves the object pointed to by *name* to the file pointed to by *dstname*. The object is removed from CCDS following successful unimport. Only the owner of an object can unimport.

RETURN VALUES

Upon successful completion, a value of 0 is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_EACCES
File not found or can not create

CCDS_EFTBL
server file table full

CCDS_ECFTBL
client file table full

CCDS_EFSTM
error in underlying file system

CCDS_ETPORT
error in underlying transport system

CCDS_ELOCK
section is locked

CCDS_ESRC
can't open src

CCDS_ENAME
bad name

NAME

ccds_unlock - unlock file

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

int ccds_unlock(int fdes);

DESCRIPTION

ccds_unlock() unlocks the segment associated with *fdes* previously locked by ccds_lock() or ccds_olock()

RETURN VALUES

Upon successful completion, a value of 0 is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_BADF

not an open file descriptor

CCDS_ETPORT

error in underlying transport system

NAME

ccds_write - write to a file

SYNOPSIS

#include <ccds.h>

#include <ccds_errno.h>

ccds_ssizet ccds_write(int fdes, char *buf, ccds_sizet nbyte);

ccds_ssizet ccds_owrite(int fdes, char *buf, ccds_sizet nbyte, ccds_offt offset);

ccds_ssizet ccds_pwrite(int fdes, char *buf, ccds_sizet nbyte, ccds_offt offset);

DESCRIPTION

ccds_write() writes *nbyte* bytes to the file specified in *fdes*, beginning at the current file pointer position, from the buffer pointed to by *buf*. ccds_write() stops writing if an error condition is encountered. The file pointer is incremented by the number of bytes written.

ccds_owrite() performs the same action as ccds_write(), except that the starting offset is specified by the *offset* parameter.

ccds_pwrite() performs the same action as ccds_owrite(), except that the file pointer associated with *fdes* is unchanged.

RETURN VALUES

Upon successful completion, the number of bytes written is returned; otherwise, a negative error value is returned.

ERROR VALUES

The following error values can be returned

CCDS_EFSTM

error in underlying file system

CCDS_BADF

not an open file descriptor

CCDS_INVID

not owner of file descriptor

CCDS_ETPORT
: error in underlying transport system

CCDS_EINSUF
: insufficient system resources

CCDS_ELOCK
: section is locked

B CCDS Console

The following pages contain a description of the CCDS console program.

NAME

ccdssh - ccds console

SYNOPSIS

ccdssh -session [session name] -path [pathname] -debug [value]

DESCRIPTION

ccdssh is a shell user interface to the Collaborating Computing Data Space (CCDS) which provides functions for object management. ccdssh spawns a data server on the local machine if a server in the named session is not already active.

OPTIONS

-session

Session name. Default is ccds_session.

-path

Path to local data storage. Default is /tmp.

-debug

Debug output level (0–6). Default is 0 (no debug output)

COMMANDS

ccds_import/cimp srcname name

Import object with name srcname from file system to CCDS where it is named name.

ccds_unimport/cout dstname name

Unimport object (object is removed from CCDS) named name to file system where it is named dstname. Only user who originally imported object can unimport object.

ccds_export/cexp dstname name

Same as ccds_unimport, but object is not removed from CCDS and any user can export object.

ccds_rm/crm name

Remove object named name from CCDS. Only user who originally imported object can remove.

ccds_ls/cls

Object listing.

ccds_cat/ccat name

Display object.

ccds_notify/cnot

Deactivate or activate object notification. An object notification is generated when an object is imported or removed. Default notification is off.

quit

Leave ccdssh. When every participant in the session has quit ccdssh, the local data server also terminates.

ls, cat, more

Shell commands.