CAVEComm

Virtual reality has become an increasingly familiar part of the science of visualization and communication of information. This, combined with the increase in connectivity of remote sites via high-speed networks, allows for the development of a collaborative distributed virtual environment. Such an environment enables the development of supercomputer simulations with virtual reality visualizations that can be displayed at multiple sites, with each site interacting, viewing, and communicating about the results being discovered.

MOVIE

Available Information on CAVEComm Library

RECEIVED**MAR 27 1996****OSTI**

- ☐ CAVEComm Library Manual
 - ☐ Distribution Readme
 - ☐ FTP a version of the library (Newest Version as of November 10, 1995)
 - ☐ Bug List
-

Publications:

Sharing Visualization Experiences among Remote Virtual Environments, in the proceedings of the **International Workshop on High Performance Computing for Computer Graphics and Visualization**, Swansea, Wales in July 1995.

Collaborators

- ☐ Terry Disz
- ☐ Michael Papka
- ☐ Michael Pellegrino
- ☐ Rick Stevens

Mathematics and Computer Science Division, Argonne National Laboratory

This work was supported by the U.S. Dept. of Energy under Contract W-31-109-Eng-38.

Contact

Terry Disz

Argonne National Laboratory

Mathematics and Computer Science Division

9700 South Cass Avenue

Building 221, Room D-256

Argonne, IL 60439-4825

disz@mcs.anl.gov

[CAVE Applications](#) | [Virtual Environments](#) | [Futures Lab](#) | [MCS Home Page](#)

CAVE is a trademark of the Board of Trustees at the University of Illinois

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER

CAVEComm library 0.3

NOTICE

NEW Things With Version 0.3

- ☐ The biggest change in v0.3 is the move to `nexus_lite`. This should improve the stability of CAVEcomm on the SGI machines. A new function needed to be written to do this, it is `c2cUpdate()` and it needs to be placed within the computation update. Example
- ☐ With regards to the hostname not be expanded problem, the nexus group has changed their format, from `.nexus-database` file to a `.resource_database` file. See examples of this file in `CAVEcomm/examples/Client` and `CAVEcomm/bin`

Some changes in function prototypes have occurred in this release.

New or Changed Functions

- ☐ `int c2cSendTracker(CAVEId sourceid, float x, float y, float z, float a, float e, float r, int stream)`
 - ☐ `int c2cSendWandButtons(CAVEId sourceid, int b1, int b2, int b3, int b4)`
 - ☐ `int c2cSendWandJoystick(CAVEId sourceid, float joyx, float joyy)`
 - ☐ `int c2cSendUser(CAVEId sourceid, c2cUser *user)`
-

NEW Things With Version 0.2

Some changes in function prototypes have occurred in this release.

New or Changed Functions

- ☐ `c2cSendStream`
 - ☐ `c2cRegisterStream`
 - ☐ `c2cWorldSendStream`
-

DISCLAIMER

This document is currently under construction so please bear with the type-o's and other errors.

Bug reports, comments, corrections, and suggestions should be sent to: CAVEComm support

Table of Contents

1. Introduction

1.1 Library model

1.2 Communications transport

2. Library structure

2.1 Application types

2.2 Communications setup

2.3 Streaming data

2.4 Receiving data

2.5 Callback prototypes

2.6 Programming levels

2.7 Debugging

3. CAVEComm library

3.1 General routines

3.2 Broker routines

3.3 Data routines

3.4 World routines

3.5 Global variables

3.6 Data structures

4. CAVEComm configuration file

5. Sample applications

5.1 CAVE sample application

5.2 Simulation sample application

6. References

7. Credits

1. Introduction

The CAVEComm library is a set of routines designed to generalize the communications between virtual environments and supercomputers.

1.1 Library model

The model for the library is that of a client/server. Somewhere at a known URL is a broker whose purpose is to mediate the communications between various client applications (CAVE, simulations, etc.).

The broker

The broker maintains tables of all clients and of all sessions (applications) that are registered with it.

The table of clients maintains such elementary information as the client's capabilities (machine type,

model, etc.) as well as basic application information (CAVE application, IBM simulation, etc.). Clients can query the broker for this information.

The table of sessions maintains information on all sessions (applications) that any registered client can run. The table maintains such information as executable names, data files, command line arguments, etc. Of course, a client cannot attach to sessions outside of the realm of their physical machine. For example, a client cannot run a CAVE application unless they have CAVE functionality on their machine. Clients can query the broker for this session information.

The client

Each CAVEComm client is an application (CAVE, simulation, etc.) that has the power to attach to any other application on the broker. A simulation that is registered with the broker can be accessed by any other application on the broker such as visualization applications or virtual environments.

1.2 Communications transport

The CAVEcomm library utilizes the Nexus communications library developed at the Mathematics and Computer Science Department at Argonne National Laboratory.

While Nexus is the communications transport, the user will not require any prerequisite knowledge in using it. All Nexus communications calls are encapsulated in the library; thus, the user is unaware of the Nexus presence.

By using Nexus, the CAVEcomm library is able to support various communications protocols such as TCP/IP sockets, UDP (reliable and unreliable), ATM, AAL5, multicast, as well as various message passing libraries.

2. Library structure

2.1. Application types

There are basically two types of applications as seen by the CAVEComm library, those that have CAVE library dependence (need the CAVE library for execution which includes CAVEs, Simulators, and Immersadesks), and those that do not have CAVE library dependence. The library reacts differently at higher programming levels based on what type of application is using it. At the lowest levels of programming with the CAVEComm library, it doesn't matter what type of application is being run. At the higher programming levels, the application type defines particular actions that are executed (transparent to the user).

CAVE applications

A CAVE application traditionally has a master process, a tracking process, and some number of drawing processes based on how many walls are active in the CAVE. The CAVEComm library changes this paradigm slightly by adding a thread to the above scenario. This thread is responsible for streaming the six pre-defined CAVE streams (S_CAVE_USER, S_HEAD_TRACKER, S_WAND_TRACKER, S_WAND_BUTTONS, S_WAND_JOYSTICK, and S_WORLD_POSITION) continually as a CAVE application goes through its normal operations.

Non-CAVE applications

A non-CAVE application will look the same as if it did not have CAVEComm functionality. There are no threads created by the library nor is there any default streaming of data.

2.2 Communications setup

All communication among users of the CAVEComm library happen asynchronously. Explicit sending of data is required, however, receipt of data happens asynchronously via stream subscription and user callbacks.

2.3 Streaming data

All data streaming is handled by generic broadcast mechanisms that manage sending data to all subscribed to it. The user merely calls a general data stream function and the library broadcasts the data to all subscribed to it. If no user is subscribed to the sending stream, then the routine merely returns without any sending. For a stream to be cast, it must first be registered so that the library can allow others to subscribe and as well as for management purposes.

Users can explicitly create and send streams packed as they so desire. The user initializes a data buffer, packs it, sends it along, and finally, frees the initialized buffer.

Here's an example of some different stream sends:

```
#define STREAM_ID 10
void SendData(void)
{ /* SendData */
    c2cBuffer *databuffer;          /* Data buffer */
    int x;                          /* Some data to send */
    float y;

    /* User data send */
    c2cInitPackBuffer(&databuffer); /* Initialize a buffer */
    c2cPackInt(&databuffer,&x,1);    /* Pack an integer */
    c2cPackFloat(&databuffer,&y,1);  /* Pack a float */
    c2cSendStream(myid,&databuffer,STREAM_ID); /* Send the generic stream */
    c2cFreePackBuffer(&databuffer); /* Free the buffer */

    /* Predefined stream send */
    c2cSendWandJoystick(CAVE_JOYSTICK_X,CAVE_JOYSTICK_Y); /* Send joystick stream */
} /* SendData */
```

2.4 Receiving data

All data is received asynchronously, thus the user never makes an explicit read for data nor does the user ever block waiting for incoming data. All data when received is routed to the user via callbacks that are executed when data is ready.

The CAVEComm process is interrupted whenever data is received and a callback is executed while the processor is interrupted. The length of this interrupt lasts the duration of the complete execution of the callback. While the process is in a state of interrupt, incoming communications are blocked until the callback has completed execution. This gives the user two requirements when coding the callbacks. The first requirement would be to make the callback as efficient and compact as possible. The longer the callback takes to execute, the longer incoming data is blocked for processing. The second requirement is that CAVEComm communications programming within the interrupt (ie: coded into the callback) is a

poor practice and a process can easily deadlock if any part of the communications performs a wait for incoming data.

The user goes through a four step process to receive a remote data stream. The first thing to do is subscribe to the stream. The next step happens without the aid of the user. Data is received and the callback is executed. It is then up to the user to unpack the data (step three) and finally, free the data buffer (step four).

2.5 Callback prototypes

All data transfer within the CAVEComm library is asynchronous so when an application receives data, a callback is executed and the data is passed as parameters to that callback. The number and types of parameters of the callback is based on what type of data stream is being received. The breakdown is as follows:

S_CAVE_USER

*void callbackname(CAVEUser *user,CAVEId id);*

The parameter *user* is a structure containing all tracking, button, joystick, and world information of the remote CAVE. *Id* is the CAVEId of the sending CAVE as known to both of them on the broker.

S_HEAD_TRACKER

void callbackname(float x,float y,float z,float a,float e,float r,CAVEId id);

The parameters *x,y,z* refer to the location of the head tracker in CAVE space and the parameters *a,e,r* refer to the azimuth, elevation, and roll angles of the tracker. *Id* is the CAVEId of the sending CAVE as known to both of them on the broker.

S_WAND_TRACKER

void callbackname(float x,float y,float z,float a,float e,float r,CAVEId id);

The parameters *x,y,z* refer to the location of the wand tracker in CAVE space and the parameters *a,e,r* refer to the azimuth, elevation, and roll angles of the tracker. *Id* is the CAVEId of the sending CAVE as known to both of them on the broker.

S_WAND_BUTTON

void callbackname(int b1,int b2,int b3,int b4,CAVEId id);

The parameters *b1,b2,b3,b4* refer to the state of the wand buttons on a given CAVE. *Id* is the CAVEId of the sending CAVE as known to both of them on the broker.

S_WAND_JOYSTICK

void callbackname(float joyx,float joyy,CAVEId id);

The parameters *joyx,joyy* refer to the state of the joystick valuator on a given CAVE. *Id* is the CAVEId of the sending CAVE as known to both of them on the broker.

S_WORLD_POSITION

void callbackname(float x,float y,float z,float a,float e,float r,CAVEId id);

The parameters x,y,z refer to the location of the remote CAVE in world coordinate space and the parameters a,e,r refer to the azimuth, elevation, and roll angles of the CAVE in that space. Id is the CAVEId of the sending CAVE as known to both of them on the broker.

User Streams

*void callbackname(void *databuffer,CAVEId id);*

The parameter *databuffer* points to a buffer containing user data (to be unpacked with *c2cGetn* calls). *Id* is the CAVEId of the sending CAVE as known to both of them on the broker.

There is also a callback that can be executed upon the subscription to a stream as specified by *c2cRegisterStream*. The callback prototype is as follows:

Stream Subscribe Callback

void callbackname(StreamType stream,CAVEId id);

The *stream* parameter specifies what type of stream was subscribed to to initiate the callback and *id* is the identifier of the subscriber on the broker.

2.6. Programming levels

The CAVEComm library provides user different levels of functionality. At the highest level, a user can track a remote user with a minimum of five function calls. While this level provides the most ease in use, it doesn't take full advantage of all the libraries features nor does it provide the most power or flexibility. Access to other library features and increased power and flexibility is accomplished via calling lower level routines. Here is an example of a program that does the same thing, however, implemented at both the high and low level:

Low level

```
void Datacallback(CAVEUser *user,CAVEId id)
{ /* Datacallback */
    Usercallback(user,id); /* Call the user callback for drawing */
} /* Datacallback */

c2cConfig config;          /* Configuration file buffer */
HostId hid;                /* Id of the host to connect to */
CAVEId cid;                /* Id of our application on broker */

c2cReadConfigFile(NULL,&config); /* Read config file */
c2cInit(argc,argv,&config.env); /* Start c2c */
hid = c2cBrokerAttach(config.broker); /* Attach to broker */
cid = c2cRegister(hid,&c2cEnv); /* Register on broker */
c2cSubscribe(hid,cid,S_CAVE_USER,Datacallback); /* Subscribe to data */
```

High level

```
c2cWorldInit(argc,argv); /* Start c2c */
c2cTrackUserInit("Myself",Usercallback); /* Subscribe */
c2cDrawUser("Myself"); /* Draw the user */
```

Both programs subscribe to themselves for complete CAVE user information. Both functions will execute the drawing function *Usercallback* for drawing the remote user when data is ready. The higher level greatly simplifies the coding for such an action.

2.7 Debugging

Debugging parallel and networking applications can be quite difficult especially using standard debugging tools. The CAVEComm library provides users with one mechanism for debugging their source programs. The application debugging tool described is that of a layered text printing paradigm.

Users put function calls in their code that mimic the standard printf calls (include a format string and any parameters). Also included is an integer stating what *layer* at which the print is to occur. At run-time, the library checks the *layer* of the print statement and prints text to standard output. The user specifies the maximum level to print at run-time. All prints with a layer less than or equal to this maximum will be printed.

There are three ways for the user to specify the maximum debugging level for their application. The first way would be to put it in their configuration file. The next method of setting the debugging level would be to include the command line parameter `-c2cdbg` followed by an integer *level* (`-c2cdbg level`). The last method would be to explicitly set it in the application via a call to `c2cSetDebugLevel`.

This type of debugger gives the user a simple tool already built in to trace their application's operations on various levels. By adding these print statements to application code (using the `c2cPrintf` call), the user can easily check for correct program operation.

The CAVEComm library uses this debugging tool itself. All the CAVEComm library debugging statements start at level 100. To view the debugging prints of the CAVEComm library, use a maximum debugging level of 140. All CAVEComm routines are logged to standard output.

Here's an example of some CAVEComm library debugging code:

```
c2cPrintf(30,"Testing c2c debugging tool\n");
c2cPrintf(100,"starting c2cWorldInit\n");
c2cPrintf(100,"reading configuration file\n");
c2cReadConfigFile(NULL,&config); /* Get setup information */
c2cPrintf(100,"starting c2c functionality\n");
c2cInit(argc,argv,&config.env); /* Start c2c functionality */
c2cPrintf(100,"attaching to broker\n");
c2cWorld.hostid = c2cBrokerAttach(config.broker); /* Attach to broker */
c2cPrintf(100,"registering with broker\n");
c2cWorld.caveid = c2cRegister(c2cWorld.hostid,&c2cEnv); /* Register with broker */
c2cPrintf(30,"finished registering with id of %d\n",c2cWorld.caveid);
```

Here's the result of the above CAVEComm library debugging code with a maximum debugging level of 100:

```
c2cdbg 30: Testing c2c debugging tool
c2cdbg 100: starting c2cWorldInit
c2cdbg 100: reading configuration file
c2cdbg 100: starting c2c functionality
c2cdbg 100: attaching to broker
c2cdbg 100: registering with broker
c2cdbg 30: finished registering with id of 4
```

Here's the result of the above CAVEComm library debugging code with a maximum debugging level of 30:

```
c2cdbg 30: Testing c2c debugging tool
c2cdbg 30: finished registering with id of 4
```


3. CAVEComm library

All functions in the CAVEComm library start with the prefix **c2c**. The CAVEComm library contains functions are broken down into the following sections:

3.1 General routines

Basic routines for CAVEComm functionality

3.2 Broker routines

Broker routines handle operations that are specific to dealing broker.

3.3 Data routines

Data routines handle the management of various data streams.

3.4 World routines

World routines are abstractions of many of the CAVEComm calls providing the user with minimal program modification and CAVEComm library knowledge.

3.5 Global variables

Global variables that the user can access maintained within the CAVEComm library.

3.1 General routines

int c2cGetDebugLevel(void);

Get the current maximum debugging level in an application.

*void c2cInit(int *argc, char *argv[], CaveEnv *env);*

Start all communications functionality. Its arguments are the argument count, argument vector, and a structure describing the application's operating environment. It **MUST** be the first CAVEComm library call made (except for c2cReadConfigFile or if the world routines are utilized). If the CAVEComm library is used with the CAVE library, the call to c2cInit must occur **AFTER** the call to *CAVEInit* is made.

*void c2cPrintf(int level, char *format, ...);*

This function is very much similar to the printf function found in the standard I/O C library. The *level* parameter tells the library at what level to print the text. The text is of the standard printf setup consisting of a character string describing how to format the text and any applicable variables contained in the format.

*int c2cReadConfigFile(char *filename, c2cConfig *config);*

Read *filename* for application definitions. The definitions are placed into *config* which is a structure containing all known application definitions. If *filename* is NULL, the default file *.c2cConfig* is attempted to be opened. If *filename* or *.c2cConfig* cannot be opened, *E_OPEN_CONFIG_FILE* is returned, otherwise, *E_SUCCESS* is returned.

void c2cSetDebugLevel(int level);

Set the current maximum debugging level in an application.

void c2cTerminate(void);

End all communications functionality. It is the last CAVEComm library call made (unless the world routines are utilized).

3.2 Broker routines

HostId c2cBrokerAttach(URL url);

Attach to the broker listed by *url*. The URL must be of the format **c2cBroker://{ip hostname}:{ip port}**/. Supplying a URL of a different format will give unpredictable results. A HostId is returned uniquely describing that broker.

int c2cBrokerDetach(HostId host);

Detach from the broker specified by *host*. No further references can be made to *host* after it is detached. E_INVALID_HOST is returned if HostId doesn't match a host that was previously attached to with c2cBrokerAttach, otherwise, E_SUCCESS is returned.

int c2cBrokerKill(HostId host);

Terminate the broker specified by *host*. All database information on the broker is disposed of and all programs connected are terminated. Most programs will never issue this call. E_INVALID_HOST is returned if HostId doesn't match a host that was previously attached to with c2cBrokerAttach, otherwise, E_SUCCESS is returned.

*CAVEId c2cGetClientId(HostId host,char *name);*

Query the broker *host* and return the assigned CAVEId of the client *name*. The CAVEId is returned if the client name is found, otherwise, E_INVALID_CLIENT is returned. If the host is invalid, E_INVALID_HOST is returned.

*int c2cGetClients(HostId host,int *num_clients,CaveEnv **client_list);*

Retrieve a list of all clients currently registered on broker *host*. The number of clients registered is returned in *num_clients* along with an array of all the registered CaveEnv structures.

E_INVALID_HOST is returned if HostId doesn't match a host that was previously attached to with c2cBrokerAttach, otherwise, E_SUCCESS is returned.

*SessionId c2cGetSessionId(HostId host,char *name);*

Query the broker *host* and return the assigned SessionId of the session *name*. The SessionId is returned if the session name is found, otherwise, E_INVALID_SESSION is returned. If the host is invalid, E_INVALID_HOST is returned.

*int c2cGetSessions(HostId host,int *num_sessions,CaveSession **session_list);*

Retrieve a list of all sessions currently registered on broker *host*. The number of sessions registered is returned in *num_sessions* along with an array of all the registered CaveSession structures.

E_INVALID_HOST is returned if HostId doesn't match a host that was previously attached to with c2cBrokerAttach, otherwise, E_SUCCESS is returned.

int c2cKillSession(HostId host,SessionId session,CAVEId cave);

Not yet implemented.

*CAVEId c2cRegister(HostId host,CaveEnv *env);*

Register your application on broker *host* with the environment *env*. All subsequent queries to the broker with respect to clients will reflect your application's presence. E_INVALID_HOST is returned if HostId doesn't match a host that was previously attached to with c2cBrokerAttach.

E_CLIENT_EXISTS is returned if your client name is already used on the broker. If there are no errors, a CAVEId (0 or higher) is returned.

int c2cSessionAttach(HostId host,SessionId session, CAVEId cave);

Not yet implemented.

*SessionId c2cSessionCreate(HostId host,CaveSession *ses);*

Create session on broker *host* with session *ses*. All subsequent queries to the broker with respect to sessions will reflect your application session's presence. E_INVALID_HOST is returned if HostId doesn't match a host that was previously attached to with c2cBrokerAttach. E_SESSION_EXISTS

is returned if your session name is already used on the broker. If all goes well, E_SUCCESS is returned.

int c2cSessionDetach(HostId host,CAVEId cave);

Not yet implemented.

*int c2cSubscribe(HostId host,CAVEId datasource,StreamType stream,void *callback);*

Inform broker *host* that your application would like receive stream data of type *stream* from *datasource* and call *callback* when any incoming data is to be processed. The remote application will then start streaming the requested data to your applicaiton. E_INVALID_HOST is returned if HostId doesn't match a host that was previously attached to with c2cBrokerAttach.

E_INVALID_CAVE_ID is returned if *datasource* is invalid on *host*. E_INVALID_STREAM is returned *datasource* doesn't have *stream* available. E_SUCCESS is returned upon successful subscription to *stream*.

int c2cUnregister(HostId host,CAVEId cave);

Tell the broker *host* to remove your application from the client list. All subsequent queries to the broker with respect to clients will not reflect your applications presence. E_INVALID_HOST is returned if HostId doesn't match a host that was previously attached to with c2cBrokerAttach.

E_INVALID_CAVE_ID is returned if *cave* is invalid on *host*. E_SUCCESS is returned upon successful subscription to *stream*.

*int c2cUnsubscribe(HostId host,CAVEId datasource,StreamType stream,void *callback);*

Inform broker *host* that your application would like cancel the streaming of data of type *stream* from *datasource* with the matching callback *callback*. The remote application no longer streams data to your application. E_INVALID_HOST is returned if HostId doesn't match a host that was previously attached to with c2cBrokerAttach. E_INVALID_CAVE_ID is returned if *datasource* is invalid on *host*. E_INVALID_STREAM is returned if *datasource* doesn't have *stream* available. E_INVALID_CALLBACK is returned if a callback is given that wasn't used for a subscription to *stream*. E_SUCCESS is returned upon successful unsubscription to *stream*.

3.3 Data routines

*void c2cFreeDataBuffer(void *buffer);*

Free the data buffer *buffer* (accessed with c2cGetn calls) from memory.

*void c2cFreePackBuffer(c2cBuffer **buffer);*

Free the data buffer *buffer* (accessed with c2cPackn calls) from memory.

*void c2cGetChar(void *buffer,char *data,int size);*

Get *size* characters from *buffer* and put them into *data*.

*void c2cGetDouble(void *buffer,double *data,int size);*

Get *size* doubles from *buffer* and put them into *data*.

*void c2cGetFloat(void *buffer,float *data,int size);*

Get *size* floats from *buffer* and put them into *data*.

*void c2cGetInt(void *buffer,int *data,int size);*

Get *size* integers from *buffer* and put them into *data*.

*void c2cGetLong(void *buffer,long *data,int size);*

Get *size* long integers from *buffer* and put them into *data*.

*void c2cInitPackBuffer(c2cBuffer **buffer);*

Initialize data buffer *buffer* for future packing.

*void c2cPackChar(c2cBuffer **buffer,char *data,int size);*

Add *size* blocks of character *data* to *buffer* for broadcast later.

*void c2cPackDouble(c2cBuffer **buffer,double *data,int size);*

Add *size* blocks of double *data* to *buffer* for broadcast later.

*void c2cPackFloat(c2cBuffer **buffer,float *data,int size);*

Add *size* blocks of float *data* to *buffer* for broadcast later.

*void c2cPackInt(c2cBuffer **buffer,int *data,int size);*

Add size blocks of integer *data* to *buffer* for broadcast later.

*void c2cPackLong(c2cBuffer **buffer,long *data,int size);*

Add size blocks of long *data* to *buffer* for broadcast later.

*int c2cRegisterStream(StreamType stream,void *subscribecallback,void *unsubscribecallback);*

Register a stream in your application. No streams can be subscribed to until they are registered. If *subscribecallback* is supplied (it is not NULL), that callback will be executed every time the stream is subscribed to. If *unsubscribecallback* is supplied (it is not NULL), that callback will be executed every time the stream is unsubscribed to. E_STREAM_REGISTERED is returned if the *stream* is already registered. E_SUCCESS is returned otherwise.

*int c2cSendStream(CAVEId sourceid, c2cBuffer **buffer,StreamType stream);*

Cast a buffer of data *buffer* (previously packed with *c2cPackn* routines) of stream type *stream* to all applications subscribed to that stream with a source identifier of *sourceid*. The source identifier tells the remote processes who sent them the stream. E_INVALID_STREAM is returned if the stream wasn't registered, otherwise, E_SUCCESS is returned.

int c2cSendHeadTracker(float x,float y,float z,float a,float e,float r);

Cast local CAVE head tracker information (passed via parameters) to all applications subscribed to the stream S_HEAD_TRACKER. E_INVALID_STREAM is returned if the stream wasn't registered, otherwise, E_SUCCESS is returned.

*int c2cSendUser(CAVEId sourceid, CAVEUser *user);*

Cast local CAVE user (all trackers, buttons, joystick and world info) to all applications subscribed to the stream S_CAVE_USER. E_INVALID_STREAM is returned if the stream wasn't registered, otherwise, E_SUCCESS is returned.

int c2cSendWandButtons(CAVEId sourceid, int b1,int b2,int b3,int b4);

Cast local CAVE wand button information (passed via parameters) to all applications subscribed to the stream S_WAND_BUTTON. E_INVALID_STREAM is returned if the stream wasn't registered, otherwise, E_SUCCESS is returned.

int c2cSendWandJoystick(CAVEId sourceid, float joyx,float joyy);

Cast local CAVE wand joystick information (passed via parameters) to all applications subscribed to the stream S_WAND_JOYSTICK. E_INVALID_STREAM is returned if the stream wasn't registered, otherwise, E_SUCCESS is returned.

int c2cSendWandTracker(float x,float y,float z,float a,float e,float r);

Cast local CAVE wand tracker information (passed via parameters) to all applications subscribed to the stream S_WAND_TRACKER. E_INVALID_STREAM is returned if the stream wasn't registered, otherwise, E_SUCCESS is returned.

int c2cSendWorldPosition(float x,float y,float z,float a,float e,float r);

Cast local CAVE world position (passed via parameters) to all applications subscribed to the stream S_WORLD_POSITION. E_INVALID_STREAM is returned if the stream wasn't registered, otherwise, E_SUCCESS is returned.

int c2cUnregisterStream(StreamType stream)

Unregister a stream with your application. No remote applications can subscribe to a stream once it is unregistered. E_INVALID_STREAM is returned if the stream wasn't registered, otherwise, E_SUCCESS is returned.

3.4 World routines

void c2cDrawAllUsers(void);

Draw all users that are tracked via c2cTrackUserInit.

*void c2cDrawSomeUsers(int count,char *users[]);*

Draw a list of users that are tracked via c2cTrackUserInit. The number of users in the list as well as the application names of the users to draw are passed as parameters.

*int c2cDrawUser(char *user);*

Draw a user that is tracked via *c2cTrackUserInit*. The user to draw is specified by giving his application name for *user*. *E_INVALID_CLIENT* is returned if an invalid client name is given. *E_SUCCESS* is returned otherwise.

*void c2cTrackUserInit(char *user,void *callback);*

Start tracking *user* (specified by his application name). The rendering of the user can be user defined by supplying a callback. Whenever a draw command is issued, this callback will be executed. If *NULL* is given as the callback, a default stick-man representation of the user will be rendered. *E_INVALID_CLIENT* is returned if *user* is invalid. *E_INVALID_STREAM* is returned if *user* doesn't have his tracking available. *E_SUCCESS* is returned upon successful tracking initialization.

*int c2cTrackUserExit(char *user,void *callback);*

Cancel the tracking of *user* (specified by his application name). The callback of the tracked user **MUST** match that given when initiated. *E_INVALID_CLIENT* is returned if *user* is invalid. *E_INVALID_STREAM* is returned if *user* doesn't have his tracking available. *E_SUCCESS* is returned upon successful tracking exiting.

void c2cWorldDataInit(void);

Initializes necessary data structures needed for world communications. This **MUST** be the first CAVEComm library call issued in an application (except for *c2cReadConfigFile*). If the application running is a CAVE application, it **MUST** occur before the call to *CAVEInit* is made. In addition, if any CAVEComm application wishes to utilize the user tracking facilities, they **MUST** make a call to *c2cWorldDataInit* (whether the application is CAVE based or not).

*void c2cWorldDataSubscribe(char *user,StreamType stream,void *callback);*

Subscribe to *user* (specified by his application name) for data stream *stream* and to call *callback* when the data is received. *E_INVALID_CLIENT* is returned if *user* is invalid. *E_INVALID_STREAM* is returned if *user* doesn't have *stream* available. *E_SUCCESS* is returned upon successful data subscription.

*void c2cWorldDataUnsubscribe(char *user,StreamType stream,void *callback);*

Unsubscribe to the data stream *stream* that was previously subscribed to from *user* (specified by his application name) with *callback*. *E_INVALID_CLIENT* is returned if *user* is invalid. *E_INVALID_STREAM* is returned if *user* doesn't have *stream* available. *E_INVALID_CALLBACK* is returned if a callback is given that wasn't used for a subscription to *stream*. *E_SUCCESS* is returned upon successful data unsubscription.

void c2cWorldExit(void);

Terminate all World functionality. This **MUST** be the last CAVEComm library call issued in an application.

*CAVEId c2cWorldGetClientId(char *name);*

Query the broker and return the assigned CAVEId of the client *name*. The CAVEId is returned if the client name is found, otherwise, *E_INVALID_CLIENT* is returned.

*SessionId c2cWorldGetSessionId(char *name);*

Query the broker and return the assigned SessionId of the session *name*. The SessionId is returned if the session name is found, otherwise, *E_INVALID_SESSION* is returned.

*void c2cWorldInit(int *argc,char *argv[]);*

Initiate all World functionality. If the CAVEComm library is used with the CAVE library, it must occur **AFTER** the call to *CAVEInit* is made.

*int c2cWorldSendStream(c2cBuffer **buffer,StreamType stream);*

Cast a buffer of data *buffer* (previously packed with *c2cPackn* routines) of stream type *stream* to all applications subscribed to that stream. *E_INVALID_STREAM* is returned if the stream wasn't registered, otherwise, *E_SUCCESS* is returned.

3.5 Global variables

The CAVEComm library maintains global variables that the user can access. They are as follows:

CaveEnv c2cEnv

This variable maintains all local environment information that was registered with c2cInit.

3.6 Data structures

The CAVEComm library has data structures that are accessible by the user. They are as follows:

CaveEnv

```
/* Structure defining CAVE environment */
typedef struct caveenv
{
    char name[C2C_NAME_SIZE];      /* Unique name for this CAVE */
    int type;                      /* Type of session this is (CAVE, I-Desk, etc...) */
    int id;                       /* Id of the client */
}CaveEnv;
```

CaveSession

```
/* Structure describing a CAVE session */
typedef struct cavesession
{
    CAVEId owner;                 /* Who owns the session on the broker */
    char name[C2C_NAME_SIZE];     /* Unique name describing the session */
    char pathname[C2C_PATH_SIZE]; /* Where to find the application */
    char execname[C2C_EXE_SIZE];  /* Executable name */
    char args[C2C_ARGS_SIZE];     /* Command line arguments */
    int id;                      /* Id of the session */
}CaveSession;
```

CAVEUser

```
/* Structure defining CAVE user */
typedef struct caveuser
{
    float headx,heady,headz,      /* Head tracker location */
          heada,heade,headr,      /* Head tracker orientation */
          wandx,wandy,wandz,      /* Wand tracker location */
          wanda,wande,wandr,      /* Wand tracker orientation */
          worldx,worldy,worldz,    /* World location */
          worlda,worldr,worldr;    /* World orientation */
    int   but1,but2,              /* Button information */
          but3,but4;
    float joyx,joyy;              /* joystick information */
}CAVEUser;
```

c2cConfig

```
/* Structure defining a CAVEComm session */
typedef struct c2cconfig
{
    URL broker;                  /* URL to connect to broker with */
    CaveEnv env;                 /* Local environment parameters */
    CaveSession session;         /* Session info for this CAVE app */
}c2cConfig;
```

4. CAVEComm configuration file

The CAVEComm library looks for a configuration file (.c2cConfig by default) that defines certain aspects of the application. The format for the configuration file is *keyword [option]*. Keywords are not case sensitive. Application definitions are as follows:

Broker "*Broker URL*"

A known URL of the broker that you are to register and attach to. The broker URL **MUST** be of the format **c2cBroker://{ip hostname}:{ip port}/**.

DebugLevel *level*

Set the level for debugging the application. Debugging will be set at *level* as soon as the parser finishes processing the DebugLevel keyword.

ClientName "*Client application name*"

The name of the application that is used when registering on the broker.

ClientAppType *type*

The application type describes what type of application it is (CAVE, IMMERSADESK, SIMULATION).

SessionName "*Session name*"

The name of the session given to the broker upon session creation.

SessionPath "*Pathname to application*"

The pathname of the executable for the session. It is given to the broker upon session creation.

SessionExe "*Session executable name*"

The executable name of the application for the session. It is given to the broker upon session creation.

SessionArgs "*Command line arguments*"

The command line arguments needed to run the application. They are given to the broker upon session creation.

5. Sample applications

5.1 CAVE sample application

Below is a sample application using the CAVEComm library in a CAVE application:

```
/*
  Simple CAVE-to-CAVE application with CAVE functionality

  This program contacts the CAVE application called "Argonne CAVE" and
  tracks its user. You will see the remote user navigating through the
  coordinate space with the default (stick-man) user representation
*/
#include                /* CAVE library */
#include                /* CAVE-to-CAVE library */

void UserDraw(void);    /* Prototype for drawing function */
int connection;

void main(int argc, char *argv[])
{ /* main */
    int rc;              /* Return Code */

    c2cWorldDataInit();  /* Perform PreInitialization */
    CAVEConfigure(&argc, argv, NULL); /* Get CAVE configuration info */
    CAVEInit();          /* Start CAVE application */
    CAVEDisplay(UserDraw, 0); /* Assign drawing function */
    c2cWorldInit(&argc, argv); /* Start CAVE-to-CAVE functionality */
}
```

```

rc = -1;
connection = 0;
while (rc != E_SUCCESS)
{
    printf("No One to ATTACH to yet\n");
    rc = c2cTrackUserInit("Argonne CAVE2",NULL);/* Track Argonne user with defa
    sleep(1);
}/* End of WHILE Loop */
connection = 1;

while(!getbutton(ESCKEY))          /* Wait for escape key */
{ /* While */
    /* Do computation here or spin endlessly */
    c2cUpdate(); /* <----- NEW FUNCTION */
} /* While */
c2cTrackUserExit("Argonne CAVE2",NULL);/* Stop tracking Argonne user */
c2cWorldExit();          /* End CAVE-to-CAVE functionality */
CAVEExit();              /* End the CAVE application */
} /* main */

/*
User drawing function with CAVE-to-CAVE functionality
*/
void UserDraw(void)
{ /* UserDraw */
    float bg[3] = {0,0,0};

    c3f(bg);
    clear();          /* Clear frame buffers */
    zclear();
    c2cDrawAllUsers(); /* Draw remote user */
} /* UserDraw */

```

Here is its corresponding .c2cConfig file:

```

c2cBroker://cavesound.mcs.anl.gov:1743/
ClientApptype CAVE
Clientname "CAVEComm Test"

```

5.2 Simulation sample application

Below is an example of a client/server CAVEComm application. The server casts its application name to everyone subscribed to it (in this case, everyone subscribed to stream APP_NAME_STREAM.

Server program:

```

/*
Simple CAVE-to-CAVE application without CAVE functionality

This server sends a stream of data (its app name) to everyone
subscribed to it
*/
#include          /* CAVE-to-CAVE library */
#include ;        /* Standard I/O library */

#define APP_NAME_STREAM 10          /* Requested stream id */

void main(int argc,char *argv[])
{ /* main */
    c2cBuffer *databuffer;          /* Data buffer to pack */
    long longdata = 12345678;

```



```

float floatdata = 9876.543;
double doubledata = 5738.5674;

c2cWorldInit(&argc,argv);          /* Start CAVE-to-CAVE functionality */
c2cRegisterStream(APP_NAME_STREAM,NULL,NULL); /* Register stream to cast */
while(1)                          /* Wait for escape key */
{ /* While */
    c2cInitPackBuffer(&databuffer); /* Initialize pack buffer */
    c2cPackChar(&databuffer,&c2cEnv.name,/* Pack the sending buffer */
                C2C_NAME_SIZE);
    c2cPackInt(&databuffer,&c2cEnv.type,1);
    c2cPackLong(&databuffer,&longdata,1);
    c2cPackFloat(&databuffer,&floatdata,1);
    c2cPackDouble(&databuffer,&doubldata,1);
    c2cWorldSendStream(&databuffer,APP_NAME_STREAM); /* Broadcast your stream */
    c2cFreePackBuffer(&databuffer); /* Free sending data buffer */
    c2cUpdate();
} /* While */
c2cWorldExit();                  /* End CAVE-to-CAVE functionality */
} /* main */

```

Here is the server's corresponding .c2cConfig file:

```

Broker c2cBroker://cavesound.mcs.anl.gov:1743/
ClientApptype SIMULATION
Clientname "Argonne Simulation Server"

```

Client program:

```

/*
Simple CAVE-to-CAVE application without CAVE functionality

This client receives a stream and processes it in UserData callback
*/
#include                /* CAVE-to-CAVE library */
#include                /* Standard I/O */

#define APP_NAME_STREAM 10          /* Requested stream id */

void UserData(void *data, CAVEId id); /* Callback prototype */

void main(int argc,char *argv[])
{ /* main */
    c2cWorldInit(&argc,argv);          /* Start CAVE-to-CAVE functionality */
    c2cWorldDataSubscribe("Argonne Simulation Server",. /* Subscribe to stream */
                          APP_NAME_STREAM,UserData);
    while(1)                          /* Spin endlessly */
    { /* While */
        c2cUpdate();                  /* Do some computation here or spin endlessly */
    } /* While */
    c2cWorldExit();                  /* End CAVE-to-CAVE functionality */
} /* main */

/*
This is the callback called when the stream data is received
*/
void UserData(void *databuffer, CAVEId id)
{ /* UserData */
    char remotedata[C2C_NAME_SIZE]; /* Data buffers */
    int type;
    long longdata;
    float floatdata;
    double doubledata;

```

```

c2cGetChar(databuffer,remotedata,C2C_NAME_SIZE); /* Read the data */
c2cGetInt(databuffer,&type,1);
c2cGetLong(databuffer,&longdata,1);
c2cGetFloat(databuffer,&floatdata,1);
c2cGetDouble(databuffer,&doubledata,1);
c2cFreeGetBuffer(databuffer); /* Get rid of data buffer */
printf("Received data character %s\n",remotedata);
printf("            integer %d\n",type);
printf("            long %d\n",longdata);
printf("            float %f\n",floatdata);
printf("            double %f\n",doubledata);
printf("from source %d\n",id);
} /* UserData */

```

Here is the client's corresponding .c2cConfig file:

```

Broker c2cBroker://cavesound.mcs.anl.gov:1743/
ClientApptype SIMULATION
Clientname "Argonne Simulation Client"

```

6. References

1. T. Disz, M. Papka, M. Pellegrino, R. Stevens, *Sharing visualization experiences among remote virtual environments. Proceedings of the International Workshop on High Performance Computing for Computer Graphics and Visualization, Swansea, Wales in July 1995.*

7. Credits

The CAVEComm library is based on the Nexus communications library written and supported by Argonne's Nexus Group and the P2P shared memory library supported by Argonne's MPI Group.

Last edited October 1, 1995

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.