Inter-Client Exchange (ICE) Library

Version 1.0

X Consortium Standard

X Version 11, Release 6.1

Ralph Mor

X Consortium

Copyright © 1993, 1994 X Consortium

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ‘‘Software’’), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ‘‘AS IS’’, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE X CONSORTIUM BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of the X Consortium shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from the X Consortium.

X Window System is a trademark of X Consortium, Inc.

1. Acknowledgements

Thanks to Bob Scheifler for his thoughtful input on the design of the ICE library. Thanks also to Jordan Brown, Larry Cable, Donna Converse, Clive Feather, Stephen Gildea, Vania Joloboff, Kaleb Keithley, Stuart Marks, Hiro Miyamoto, Ralph Swick, Jim VanGilder, and Mike Wexler.

2. Overview of ICE

There are numerous possible "inter-client" protocols, with many similarities and common needs - authentication, version negotiation, byte order negotiation, etc. The ICE protocol is intended to provide a framework for building such protocols, allowing them to make use of common negotiation mechanisms and to be multiplexed over a single transport connection.

3. The ICE Library - A "C" Language Interface to ICE

A client that wishes to utilize ICE must first register the protocols it understands with the ICE library. Each protocol is dynamically assigned a major opcode ranging from 1-255 (two clients can use different major opcodes for the same protocol). The next step for the client is to either open a connection with another client, or to wait for connections made by other clients. Authentication may be required. A client can both initiate connections with other clients and be waiting for clients to connect to itself (a nested session manager is an example). Once an ICE connection is established between the two clients, one of the clients needs to initiate a *Protocol Setup* in order to "activate" a given protocol. Once the other client accepts the *Protocol Setup* (once again, authentication may be required), the two clients are ready to start passing messages specific to that protocol to each other. Multiple protocols may be active on a single ICE connection. Clients are responsible for notifying the ICE library when a protocol is no longer active on an ICE connection, although ICE does not define how each sub-protocol triggers a protocol shutdown.

The ICE library utilizes callbacks to process incoming messages. Using callbacks allows *Protocol Setups* and authentication to happen "behind the scenes." An additional benefit is that messages never need to be buffered up by the library when the client "blocks" waiting for a particular message.

4. Intended Audience

This document is intended primarily for implementors of protocol libraries layered on top of ICE. Typically, applications that wish to utilize ICE will make calls into individual protocol libraries rather than directly make calls into the ICE library. However, some applications will have to make some initial calls into the ICE library in order to accept ICE connections (for example, a session manager accepting connections from clients). But in general, protocol libraries should be designed to hide the inner details of ICE from applications.

5. Header Files and Library Name

The header file *<X11/ICE/ICElib.h>* defines all of the ICElib data structures and function prototypes. *ICElib.h* includes the header file *<X11/ICE/ICE.h>* which defines all of the ICElib constants. Protocol libraries that need to read and write messages should include the header file *<X11/ICE/ICEmsg.h>*.

Applications should link against ICElib using *-lICE*.

6. Note on prefixes

The following name prefixes are used in the library to distinguish between a client that initiates a *Protocol Setup* and a client which responds with a *Protocol Reply*:

- *Ice Protocol Originator*

- *Ice Protocol Acceptor*

7. Protocol Registration

In order for two clients to exchange messages for a given protocol, each side must register the protocol with the ICE library. The purpose of registration is for each side to obtain a major opcode for the protocol, and to provide callbacks for processing messages and handling authentication. There are two separate registration functions - one to handle the side that does a *Protocol Setup*, and one to handle the side that responds with a *Protocol Reply*.

It is recommended that protocol registration occur before the two clients establish an ICE connection. If protocol registration occurs after an ICE connection is created, there can be a brief interval of time in which a *Protocol Setup* is received, but the protocol is not registered. If it is not possible to register a protocol before the creation of an ICE connection, proper precautions should be taken to avoid the above race condition.

The function should be called for the client that initiates a *Protocol Setup*.

```
int IceRegisterForProtocolSetup(protocol_name, vendor, release,
                version_count, version_recs, auth_count, auth_names, auth_procs, io_error_proc)
    char *protocol_name;
    char *vendor;
    char *release;
    int version_count;
    IcePoVersionRec *version_recs;
    int auth_count;
    char **auth_names;
    IcePoAuthProc *auth_procs;
    IceIOErrorProc io_error_proc;
```

protocol_name	A string specifying the name of the protocol to register.
vendor	A vendor string with semantics specified by the protocol.
release	A release string with semantics specified by the protocol.
version_count	The number of different versions of the protocol supported.
version_recs	List of versions and associated callbacks.
auth_count	The number of authentication methods supported.
auth_names	The list of authentication methods supported.
auth_procs	The list of authentication callbacks, one for each authentication method.
io_error_proc	IO Error handler.

returns the major opcode reserved, or -1 if an error occurred. In order to actually activate the protocol, the function needs to be called with this major opcode. Once the protocol is activated, all messages for the protocol should be sent using this major opcode.

A protocol library may support multiple versions of the same protocol. *version_recs* specifies a list of supported versions of the protocol, prioritized in decreasing order of preference. Each version record consists of a major and minor version of the protocol, as well as a callback to be used for processing incoming messages.

```
typedef struct {
        int major_version;
        int minor_version;
        IcePoProcessMsgProc process_msg_proc;
} IcePoVersionRec;
```

The callback is responsible for processing the set of messages that can be received by the client that initiated the *Protocol Setup*. The details of how this callback works is described in the section titled *Callbacks for*

Processing Messages.

Authentication may be required before the protocol can become active. The protocol library must register the authentication methods that it supports with the ICE library. *auth_names* and *auth_procs* are a list of authentication names and callbacks, prioritized in decreasing order of preference. The details of how the callback works is described in the section titled *Authentication Methods*.

The callback is invoked if the ICE connection unexpectedly breaks. Pass for *io_error_proc* if not interested in being notified. See the section titled *Error Handling* for more details on this callback.

The function should be called for the client that responds to a *Protocol Setup* with a *Protocol Reply*.

```
int IceRegisterForProtocolReply(protocol_name, vendor, release, version_count, version_recs,
                auth_count, auth_names, auth_procs, host_based_auth_proc,
                protocol_setup_proc, protocol_activate_proc, io_error_proc)
    char *protocol_name;
    char *vendor;
    char *release;
    int version_count;
    IcePaVersionRec *version_recs;
    int auth_count;
    char **auth_names;
    IcePaAuthProc *auth_procs;
    IceHostBasedAuthProc host_based_auth_proc;
    IceProtocolSetupProc protocol_setup_proc;
    IceProtocolActivateProc protocol_activate_proc;
    IceIOErrorProc io_error_proc;
```

protocol_name A string specifying the name of the protocol to register.

vendor A vendor string with semantics specified by the protocol.

release A release string with semantics specified by the protocol.

version_count The number of different versions of the protocol supported.

version_recs List of versions and associated callbacks.

auth_count The number of authentication methods supported.

auth_names The list of authentication methods supported.

auth_procs The list of authentication callbacks, one for each authentication method.

host_based_auth_proc
Host based authentication callback.

protocol_setup_proc
A callback to be invoked when authentication has succeeded for a *Protocol Setup*, *before* the *Protocol Reply* is sent.

protocol_activate_proc
A callback to be invoked *after* the *Protocol Reply* is sent.

io_error_proc IO Error handler.

returns the major opcode reserved, or -1 if an error occurred. The major opcode should be used in all subsequent messages sent for this protocol.

A protocol library may support multiple versions of the same protocol. *version_recs* specifies a list of supported versions of the protocol, prioritized in decreasing order of preference. Each version record consists of a major and minor version of the protocol, as well as a callback to be used for processing incoming messages.

```
typedef struct {
        int major_version;
        int minor_version;
        IcePaProcessMsgProc process_msg_proc;
} IcePaVersionRec;
```

The callback is responsible for processing the set of messages that can be received by the client that accepted the *Protocol Setup*. The details of how this callback works is described in the section titled *Callbacks for Processing Messages*.

Authentication may be required before the protocol can become active. The protocol library must register the authentication methods that it supports with the ICE library. *auth_names* and *auth_procs* are a list of authentication names and callbacks, prioritized in decreasing order of preference. The details of how the callback works is described in the section titled *Authentication Methods*.

If authentication fails and the client attempting to initiate the *Protocol Setup* has not required authentication, the callback is invoked with the host name of the originating client. If the callback returns the *Protocol Setup* will succeed, even though the original authentication failed. Note that authentication can effectively be disabled by registering an which always returns If no host based authentication is allowed, pass for *host_based_auth_proc*.

typedef Bool (*IceHostBasedAuthProc) (); Bool HostBasedAuthProc(*host_name*)
   char **host_name*;

host_name   The host name of the client that sent the *Protocol Setup*.

host_name is a string of the form "protocol/hostname", where *protocol* is one of {tcp, decnet, local}.

Since *Protocol Setups* and authentication happen "behind the scenes" via callbacks, the protocol library needs some way of being notified when the *Protocol Setup* has completed. This occurs in two phases. In the first phase, the callback is invoked after authentication has successfully completed, *before* the ICE library sends a *Protocol Reply*. Any resources required for this protocol should be allocated at this time. If the returns a successful status, the ICE library will send the *Protocol Reply* and then invoke the callback. Otherwise, an error will be sent to the other client in response to the *Protocol Setup*.

The is an optional callback, and should be registered only if the protocol library intends to generate a message immediately following the *Protocol Reply*. Pass for *protocol_activate_proc* if not interested in this callback.

typedef Status (*IceProtocolSetupProc) (); Status ProtocolSetupProc(*ice_conn*, *major_version*, *minor_version*, *vendor*, *release*,
      client_data_ret, *failure_reason_ret*)
   IceConn *ice_conn*;
   int *major_version*;
   int *minor_version*;
   char **vendor*;
   char **release*;
   IcePointer **client_data_ret*;
   char ***failure_reason_ret*;

ice_conn   The ICE connection object.

major_version   The major version of the protocol.

minor_version   The minor version of the protocol.

vendor   The vendor string registered by the protocol originator.

release   The release string registered by the protocol originator.

client_data_ret	Client data to be set by callback.
failure_reason_ret	Failure reason returned.

The pointer stored in the *client_data_ret* argument will be passed to the callback whenever a message has arrived for this protocol on the ICE connection.

The *vendor* and *release* strings should be freed with free() when they are no longer needed.

If a failure occurs, the should return a zero status, as well as allocate and return a failure reason string in *failure_reason_ret*. The ICE library will be responsible for freeing this memory.

The discussed above is defined as follows:

typedef void (*IceProtocolActivateProc)();

void ProtocolActivateProc(*ice_conn*, *client_data*)
  IceConn *ice_conn*;
  IcePointer *client_data*;

ice_conn	The ICE connection object.
client_data	The client data set in the callback.

The callback is invoked if the ICE connection unexpectedly breaks. Pass for *io_error_proc* if not interested in being notified. See the section titled *Error Handling* for more details on this callback.

7.1. Callbacks for Processing Messages

When an application detects that there is a new data to read on an ICE connection (via select), it calls the function (discussed in the section titled *Processing Messages*). When reads an ICE message header with a major opcode other than zero (reserved for the ICE protocol), it needs to call a function which will read the rest of the message, unpack it, and process it accordingly.

If the message arrives at the client which initiated the *Protocol Setup*, the callback is invoked.

typedef void (*IcePoProcessMsgProc)(); void PoProcessMsgProc(*ice_conn*, *client_data*, *opcode*, *length*, *swap*, *reply_wait*, *reply_ready_ret*)
  IceConn *ice_conn*;
  IcePointer *client_data*;
  int *opcode*;
  unsigned long *length*;
  Bool *swap*;
  IceReplyWaitInfo **reply_wait*;
  Bool **reply_ready_ret*;

ice_conn	The ICE connection object.
client_data	Client data associated with this protocol on the ICE connection.
opcode	The minor opcode of the message.
length	The length (in 8 byte units) of the message beyond the ICE header.
swap	A flag which indicates if byte swapping is necessary.
reply_wait	Indicates if the invoking client is waiting for a reply.
reply_ready_ret	If set to a reply is ready.

If the message arrives at the client which accepted the *Protocol Setup*, the callback is invoked.

typedef void (*IcePaProcessMsgProc)(); void PaProcessMsgProc(*ice_conn*, *client_data*, *opcode*, *length*, *swap*)
    IceConn *ice_conn*;
    IcePointer *client_data*;
    int *opcode*;
    unsigned long *length*;
    Bool *swap*;

ice_conn	The ICE connection object.
client_data	Client data associated with this protocol on the ICE connection.
opcode	The minor opcode of the message.
length	The length (in 8 byte units) of the message beyond the ICE header.
swap	A flag which indicates if byte swapping is necessary.

In order to read the message, both of the above callbacks should use the macros defined in the section of this document titled *Reading ICE Messages*. Note that byte swapping may be necessary. As a convenience, the length field in the ICE header will be swapped by ICElib if necessary.

In both of the above callbacks, *client_data* is a pointer to client data that was registered at *Protocol Setup* time. In the case of the client data was set in the call to In the case of the client data was set in the callback.

The callback needs to check the *reply_wait* argument. If *reply_wait* is the ICE library expects the function to pass the message to the client via a callback. For example, if this is a Session Management *Save Yourself* message, this function should notify the client of the *Save Yourself* via a callback. The details of how such a callback would be defined is implementation dependent.

However, if *reply_wait* is not then the client is waiting for a reply or an error for a message it previously sent. *reply_wait* is of type

```
typedef struct {
        unsigned long sequence_of_request;
        int major_opcode_of_request;
        int minor_opcode_of_request;
        IcePointer reply;
} IceReplyWaitInfo;
```

contains the major/minor opcodes and sequence number of the message for which a reply is being awaited. It also contains a pointer to the reply message to be filled in (the protocol library should cast this to the appropriate reply type). In most cases, the reply will have some fixed-size part, and the client waiting for the reply will have provided a pointer to a structure to hold this fixed-size data. If there is variable-length data, it would be expected that the callback will have to allocate additional memory and store pointer(s) to that memory in the fixed-size structure. If the entire data is variable length (e.g., a single variable-length string), then the client waiting for the reply would probably just pass a pointer to fixed-size space to hold a pointer, and the callback would allocate the storage and store the pointer. It is the responsibility of the client receiving the reply to free any memory allocated on its behalf.

If *reply_wait* is not and has a reply or error to return in response to this *reply_wait* (i.e. no callback was generated), then the *reply_ready_ret* argument should be set to Note that an error should only be returned if it corresponds to the reply being waited for. Otherwise, the should either handle the error internally, or invoke an error handler for its library.

If *reply_wait* is then care must be taken not to store any value in *reply_ready_ret* since this pointer may also be

The callback, on the other hand, should always pass the message to the client via a callback. For example, if this is a Session Management *Interact Request* message, this function should notify the client of the *Interact Request* via a callback.

The reason the callback does not have a *reply_wait* like does, is because a process that is acting as a "server" should never block for a reply (infinite blocking can occur if the connecting client does not act properly, denying access to other clients).

7.2. Authentication Methods

As discussed earlier, a protocol library must register the authentication methods that it supports with the ICE library. For each authentication method, there are two callbacks that may be registered - one to handle the side that initiates a *Protocol Setup*, and one to handle the side that accepts or rejects this request.

is the callback invoked for the client that initiated the *Protocol Setup*. This callback must be able to respond to the initial *Authentication Required* message or subsequent *Authentication Next Phase* messages sent by the other client.

typedef IcePoAuthStatus (*IcePoAuthProc)(); IcePoAuthStatus PoAuthProc(*ice_conn*, *auth_state_ptr*, *clean_up*, *swap*,
auth_datalen, *auth_data*, *reply_datalen_ret*, *reply_data_ret*, *error_string_ret*)
IceConn *ice_conn*;
IcePointer **auth_state_ptr*;
Bool *clean_up*;
Bool *swap*;
int *auth_datalen*;
IcePointer *auth_data*;
int **reply_datalen_ret*;
IcePointer **reply_data_ret*;
char ***error_string_ret*;

ice_conn	The ICE connection object.
auth_state_ptr	A pointer to state for use by the authentication callback procedure.
clean_up	If authentication is over, and the function should clean up any state it was maintaining. The last 6 arguments should be ignored.
swap	If the *auth_data* may have to be byte swapped (depending on its contents).
auth_datalen	The length (in bytes) of the authenticator data.
auth_data	The data from the authenticator.
reply_datalen_ret	The length (in bytes) of the reply data returned.
reply_data_ret	The reply data returned.
error_string_ret	If the authentication procedure encounters an error during authentication, it should allocate and return an error string.

Authentication may require several phases, depending on the authentication method. As a result, the may be called more than once when authenticating a client, and some state will have to be maintained between each invocation. At the start of each *Protocol Setup*, **auth_state_ptr* is and the function should initialize its state and set this pointer. In subsequent invocations of the callback, the pointer should be used to get at any state previously stored by the callback.

If needed, the network ID of the client accepting the *Protocol Setup* can be obtained by calling the function.

ICElib will be responsible for freeing the *reply_data_ret* and *error_string_ret* pointers with free().

The *auth_data* pointer may point to a volatile block of memory. If the data must be kept beyond this invocation of the callback, be sure to make a copy of it.

The should return one of four values:

lw(2i) lw(5i). T{ T} T{ A reply is available T}

T{ T} T{ Authentication rejected T}

T{ T} T{ Authentication failed T}

T{ T} T{ Done cleaning up T}

is the callback invoked for the client that received the *Protocol Setup*.

typedef IcePaAuthStatus (*IcePaAuthProc) (); IcePaAuthStatus PaAuthProc(*ice_conn*, *auth_state_ptr*, *swap*,
auth_datalen, *auth_data*, *reply_datalen_ret*, *reply_data_ret*, *error_string_ret*)
IceConn *ice_conn*;
IcePointer **auth_state_ptr*;
Bool *swap*;
int *auth_datalen*;
IcePointer *auth_data*;
int **reply_datalen_ret*;
IcePointer **reply_data_ret*;
char ***error_string_ret*;

ice_conn	The ICE connection object.
auth_state_ptr	A pointer to state for use by the authentication callback procedure.
swap	If the *auth_data* may have to be byte swapped (depending on its contents).
auth_datalen	The length (in bytes) of the protocol originator authentication data.
auth_data	The authentication data from the protocol originator.
reply_datalen_ret	The length of the authentication data returned.
reply_data_ret	The authentication data returned.
error_string_ret	If authentication is rejected or fails, an error string is returned.

Authentication may require several phases, depending on the authentication method. As a result, the may be called more than once when authenticating a client, and some state will have to be maintained between each invocation. At the start of each *Protocol Setup*, *auth_datalen* is zero, **auth_state_ptr* is and the function should initialize its state and set this pointer. In subsequent invocations of the callback, the pointer should be used to get at any state previously stored by the callback.

If needed, the network ID of the client accepting the *Protocol Setup* can be obtained by calling the function.

The *auth_data* pointer may point to a volatile block of memory. If the data must be kept beyond this invocation of the callback, be sure to make a copy of it.

ICElib will be responsible for freeing the *reply_data_ret* and *error_string_ret* pointers with free().

The should return one of four values:

lw(2i) lw(5i). T{ T} T{ Continue (or start) authentication T}

T{ T} T{ Authentication accepted T}

T{ T} T{ Authentication rejected T}

T{ T} T{ Authentication failed T}

8. ICE Connections

In order for two clients to establish an ICE connection, one client has to be "waiting" for connections, and the other client has to initiate the connection. Most clients will initiate connections, so we discuss that first.

8.1. Opening an ICE Connection

In order to open an ICE connection with another client (that is waiting for connections), call the function.

IceConn IceOpenConnection(*network_ids_list, context*, *must_authenticate*, *major_opcode_check*,
error_length, *error_string_ret*)
char **network_ids_list*;
IcePointer *context*;
Bool *must_authenticate*;
int *major_opcode_check*;
int *error_length*;
char **error_string_ret*;

network_ids_list	Specifies the network ID(s) of the other client.
context	A pointer to an opaque object, or NULL. Used to determine if an ICE connection can be shared (see below).
must_authenticate	If the other client may not bypass authentication.
major_opcode_check	Used to force a new ICE connection to be created (see below).
error_length	Length of the *error_string_ret* argument passed in.
error_string_ret	Returns a null terminated error message, if any. *error_string_ret* points to user supplied memory. No more than *error_length* bytes are used.

returns an opaque ICE connection object if it succeeds, otherwise.

network_ids_list contains a list of network IDs separated by commas. An attempt will be made to use the first network ID. If that fails, an attempt will be made using the second network ID, and so on. Each network ID has the form...

lw(0.25i) lw(2.5i) lw(1i). tcp/<hostname>:<portnumber>or decnet/<hostname>::<objname> or local/<hostname>:<path>

Most protocol libraries will have some sort of "open" function which should internally make a call into When is called, it may be possible to use a previously opened ICE connection (if the target client is the same). However, there are cases in which shared ICE connections are not desired.

The *context* argument is used to determine if an ICE connection can be shared. If *context* is then the caller is always willing to share the connection. If *context* is not then the caller is not willing to use a previously opened ICE connection that has a different non-NULL context associated with it.

In addition, if *major_opcode_check* contains a non-zero major opcode value, a previously created ICE connection will be used only if the major opcode is not active on the connection. This can be used to force multiple ICE connections between two clients for the same protocol.

Any authentication requirements are handled internally by the ICE library. The method by which the authentication data is obtained is implementation dependent.†

† The X Consortium’s ICElib implementation uses an .ICEauthority file (see Appendix A).

After is called, the client is ready to send a *Protocol Setup* (provided that was called), or receive a *Protocol Setup* (provided that was called).

8.2. Listening for ICE Connections

Clients wishing to accept ICE connections must first call so they can listen for connections. A list of opaque "listen" objects are returned, one for each type of transport method that is available (for example, Unix Domain, TCP, DECnet, etc...).

```
Status IceListenForConnections(count_ret, listen_objs_ret, error_length, error_string_ret)
    int *count_ret;
    IceListenObj **listen_objs_ret;
    int error_length;
    char *error_string_ret;
```

count_ret The number of listen objects returned.

listen_objs_ret Returns a list of opaque listen objects.

error_length The length of the *error_string_ret* argument passed in.

error_string_ret Returns a null terminated error message, if any. *error_string_ret* points to user supplied memory. No more than *error_length* bytes are used.

The return value of is zero for failure, and a positive value for success.

Call to close and free the listen objects.

```
void IceFreeListenObjs(count, listen_objs)
    int count;
    IceListenObj *listen_objs;
```

count The number of listen objects.

listen_objs The listen objects.

In order to detect a new connection on a listen object, select() must be called on the descriptor associated with the listen object. To obtain the descriptor, call the function.

```
int IceGetListenConnectionNumber(listen_obj)
    IceListenObj listen_obj;
```

listen_obj The listen object.

To obtain the network ID string associated with a listen object, call the function.

```
char *IceGetListenConnectionString(listen_obj)
    IceListenObj listen_obj;
```

listen_obj The listen object.

A network ID has the form...

lw(0.25i) lw(2.5i) lw(1i). tcp/<hostname>:<portnumber>or
decnet/<hostname>::<objname> or local/<hostname>:<path>

To compose a string containing a list of network IDs separated by commas (the format recognized by call the function.

char *IceComposeNetworkIdList(*count*, *listen_objs*)
    int *count*;
    IceListenObj **listen_objs*;

count The number of listen objects.

listen_objs The listen objects.

8.3. Host Based Authentication for ICE Connections

If authentication fails when a client attempts to open an ICE connection, and the initiating client has not required authentication, a host based authentication procedure may be invoked to provide a last chance for the client to connect. Each listen object has such a callback associated with it, and this callback is set using the function.

void IceSetHostBasedAuthProc(*listen_obj*, *host_based_auth_proc*)
    IceListenObj *listen_obj*;
    IceHostBasedAuthProc *host_based_auth_proc*;

listen_obj The listen object.

host_based_auth_proc The host based authentication procedure.

By default, each listen object has no host based authentication procedure associated with it. Passing for *host_based_auth_proc* turns off host based authentication if it was previously set.

typedef Bool (*IceHostBasedAuthProc) (); Bool HostBasedAuthProc(*host_name*)
char **host_name*;

host_name The host name of the client that tried to open an ICE connection.

host_name is a string of the form "protocol/hostname" where *protocol* is one of {tcp, decnet, local}.

If returns access will be granted, even though the original authentication failed. Note that authentication can effectively be disabled by registering an which always returns

Host based authentication is also allowed at *Protocol Setup* time. The callback is specified in the function discussed earlier.

8.4. Accepting ICE Connections

After a connection attempt is detected on a listen object returned by should be called. This returns a new opaque ICE connection object.

IceConn IceAcceptConnection(*listen_obj*, *status_ret*)
IceListenObj *listen_obj*;
IceAcceptStatus **status_ret*;

listen_obj The listen object on which a new connection was detected.

status_ret Return status information.

The *status_ret* argument is set to one of the following values:

lw(2i) lw(4i). T{ T} T{ The accept operation succeeded. The function returns a new connection object. T}

T{ T} T{ The accept operation failed. The function returns NULL. T}

T{ T} T{ A memory allocation failed. The function returns NULL. T}

In general, in order to detect new connections, the application will call select() on the file descriptors associated with the listen objects. When a new connection is detected, the function should be called. may return a new ICE connection that is in a "pending" state. This is because before the connection can become valid, authentication may be necessary. Since the ICE library cannot block and wait for the connection to become valid (infinite blocking can occur if the connecting client does not act properly), the application must wait for the connection status to become "valid".

The following pseudo-code demonstrates how connections are accepted:

```
new_ice_conn = IceAcceptConnection (listen_obj);
status = IceConnectionStatus (new_ice_conn);
time_start = time_now;

while (status == IceConnectPending)
{
        select() on {new_ice_conn, all open connections}

        for (each ice_conn in the list of open connections)
                if (data ready on ice_conn)
                {
                        status = IceProcessMessages (ice_conn, NULL, NULL);
                        if (status == IceProcessMessagesIOError)
                                IceCloseConnection (ice_conn);
                }

        if (data ready on new_ice_conn)
        {
                /*
                 * IceProcessMessages is called until the connection
                 * is non-pending.  Doing so handles the connection
                 * setup request and any authentication requirements.
                 */

                IceProcessMessages (new_ice_conn, NULL, NULL);
                status = IceConnectionStatus (new_ice_conn);
        }
        else
        {
                if (time_now - time_start > MAX_WAIT_TIME)
                        status = IceConnectRejected;
        }
}

if (status == IceConnectAccepted)
{
        Add new_ice_conn to the list of open connections
}
else
{
        IceCloseConnection (new_ice_conn);
}
```

After is called and the connection has been validated, the client is ready to receive a *Protocol Setup* (provided that was called), or send a *Protocol Setup* (provided that was called).

8.5. Closing ICE Connections

To close an ICE connection created with or call the function.

IceCloseStatus IceCloseConnection(*ice_conn*)
  IceConn *ice_conn*;

ice_conn The ICE connection to close.

In order to actually close an ICE connection, the following conditions must be met:

- The *open reference count* must have reached zero on this ICE connection. When is called, it tries to use a previously opened ICE connection. If it is able to use an existing connection, it increments the *open reference count* on the connection by one. So in order to close an ICE connection, each call to must be matched with a call to The connection can be closed only on the last call to

- The *active protocol count* must have reached zero. Each time a *Protocol Setup* succeeds on the connection the *active protocol count* is incremented by one. When the client no longer expects to use the protocol on the connection, the function should be called, which decrements the *active protocol count* by one (see the *Protocol Setup and Shutdown* section).

- If shutdown negotiation is enabled on the connection, the client on the other side of the ICE connection must agree to have the connection closed.

returns one of the following values:

lw(2i) lw(4i). T{ T} T{ The ICE connection was closed at this time. The watch procedures were invoked and the connection was freed. T}

T{ T} T{ An IO error had occurred on the connection, but is being called within a nested The watch procedures have been invoked at this time, but the connection will be freed as soon as possible (when the nesting level reaches zero and returns a status of T}

T{ T} T{ The connection was not closed at this time because it is being used by other active protocols. T}

T{ T} T{ The connection was not closed at this time and shutdown negotiation started with the client on the other side of the ICE connection. When the connection is actually closed, will return a status of T}

When it is known that the client on the other side of the ICE connection has terminated the connection without initiating shutdown negotiation, the function should be called to turn off shutdown negotiation. This will prevent from writing to a broken connection.

void IceSetShutdownNegotiation(*ice_conn*, *negotiate*)
  IceConn *ice_conn*;
  Bool *negotiate*;

ice_conn A valid ICE connection object.

negotiate If shutdown negotiating will be turned off.

In order to check the shutdown negotiation status of an ICE connection, call the function.

Bool IceCheckShutdownNegotiation(*ice_conn*)
  IceConn *ice_conn*;

ice_conn A valid ICE connection object.

returns if shutdown negotiation will take place on the connection, otherwise. Negotiation is on by default for a connection. It can only be changed with the function.

8.6. Connection Watch Procedures

In order to add a watch procedure which will be called each time ICElib opens a new connection via or or closes a connection via call the function.

Status IceAddConnectionWatch(*watch_proc*, *client_data*)
  IceWatchProc *watch_proc*;
  IcePointer *client_data*;

watch_proc The watch procedure to invoke when ICElib opens or closes a connection.

client_data This pointer will be passed to the watch procedure.

The return value of is zero for failure, and a positive value for success.

Note that several calls to might share the same ICE connection. In such a case, the watch procedure is only invoked when the connection is first created (after authentication succeeds). Similarly, since connections might be shared, the watch procedure is called only if actually closes the connection (right before the IceConn is freed).

The watch procedures are very useful for applications which need to add a file descriptor to a select mask when a new connection is created, and remove the file descriptor when the connection is destroyed. Since connections are shared, knowing when to add and remove the file descriptor from the select mask would be difficult without the watch procedures.

Multiple watch procedures may be registered with the ICE library. No assumptions should be made about their order of invocation.

If one or more ICE connections were already created by the ICE library at the time the watch procedure is registered, the watch procedure will instantly be invoked for each of these ICE connections (with the *opening* flag set to

The watch procedure is of type

typedef void (*IceWatchProc)();

void WatchProc(*ice_conn*, *client_data*, *opening*, *watch_data*)
   IceConn *ice_conn*;
   IcePointer *client_data*;
   Bool *opening*;
   IcePointer **watch_data*;

ice_conn	The opened or closed ICE connection. Call to get the file descriptor associated with this connection.
client_data	Client data specified in the call to
opening	If the connection is being opened. If the connection is being closed.
watch_data	Can be used to save a pointer to client data.

If *opening* is the client should set the **watch_data* pointer to any data it may need to save until the connection is closed and the watch procedure is invoked again with *opening* set to

To remove a watch procedure, call the function.

void IceRemoveConnectionWatch(*watch_proc*, *client_data*)
   IceWatchProc *watch_proc*;
   IcePointer *client_data*;

watch_proc The watch procedure that was passed to

client_data The *client_data* pointer that was passed to

9. Protocol Setup and Shutdown

In order to activate a protocol on a given ICE connection, call the function.

IceProtocolSetupStatus IceProtocolSetup(*ice_conn*, *my_opcode*, *client_data*, *must_authenticate*, *major_version_ret*, *minor_version_ret*, *vendor_ret*, *release_ret*, *error_length*, *error_string_ret*)
   IceConn *ice_conn*;
   int *my_opcode*;
   IcePointer *client_data*;
   Bool *must_authenticate*;
   int **major_version_ret*;
   int **minor_version_ret*;
   char ***vendor_ret*;
   char ***release_ret*;
   int *error_length*;
   char **error_string_ret*;

ice_conn A valid ICE connection object.

my_opcode The major opcode of the protocol to be set up, as returned by

client_data The client data stored in this pointer will be passed to the callback.

must_authenticate If the other client may not bypass authentication.

major_version_ret The major version of the protocol to be used is returned.

minor_version_ret The minor version of the protocol to be used is returned.

vendor_ret The vendor string specified by the protocol acceptor.

release_ret The release string specified by the protocol acceptor.

error_length Specifies the length of the *error_string_ret* argument passed in.

error_string_ret Returns a null terminated error message, if any. *error_string_ret* points to user supplied memory. No more than *error_length* bytes are used.

The *vendor_ret* and *release_ret* strings should be freed with free() when no longer needed.

returns one of the following values:

lw(2i) lw(4i). T{ T} T{ *major_version_ret*, *minor_version_ret*, *vendor_ret*, *release_ret* are set. T}

T{ or
T} T{ Check *error_string_ret* for failure reason. *major_version_ret*, *minor_version_ret*, *vendor_ret*, *release_ret* are NOT set. T}

T{ T} T{ This protocol is already active on this connection. *major_version_ret*, *minor_version_ret*, *vendor_ret*, *release_ret* are NOT set. T}

In order to notify the ICE library when a given protocol will no longer be used on an ICE connection, call the function.

Status IceProtocolShutdown(*ice_conn*, *major_opcode*)
  IceConn *ice_conn*;
  int *major_opcode*;

ice_conn A valid ICE connection object.

major_opcode The major opcode of the protocol to shut down.

The return value of is zero for failure, and a positive value for success.

Failure will occur if the major opcode was never registered OR the protocol of the major opcode was never "activated" on the connection. By "activated" we mean that a *Protocol Setup* succeeded on the connection. Note that ICE does not define how each sub-protocol triggers a protocol shutdown.

10. Processing Messages

In order to process incoming messages on an ICE connection, the function should be called.

IceProcessMessagesStatus IceProcessMessages(*ice_conn*, *reply_wait*, *reply_ready_ret*)
  IceConn *ice_conn*;
  IceReplyWaitInfo **reply_wait*;
  Bool **reply_ready_ret*;

ice_conn A valid ICE connection object.

reply_wait Indicates if a reply is being waited for.

reply_ready_ret If set to on return, a reply is ready.

This function is used in two ways. In the first, a client may generate a message and "block" by calling repeatedly until it gets its reply. In the second case, a client calls with *reply_wait* set to in response to select() showing that there is data to read on the ICE connection. The ICE library may process zero or more complete messages. Note that messages which are not "blocked" for are always processed by invoking callbacks.

contains the major/minor opcodes and sequence number of the message for which a reply is being awaited. It also contains a pointer to the reply message to be filled in (the protocol library should cast this to the appropriate reply type). In most cases, the reply will have some fixed-size part, and the client waiting for the reply will have provided a pointer to a structure to hold this fixed-size data. If there is variable-length data, it would be expected that the callback will have to allocate additional memory and store pointer(s) to that memory in the fixed-size structure. If the entire data is variable length (e.g., a single variable-length string), then the client waiting for the reply would probably just pass a pointer to fixed-size space to hold a pointer, and the callback would allocate the storage and store the pointer. It is the responsibility of the client receiving the reply to free up any memory allocated on its behalf.

```
typedef struct {
        unsigned long sequence_of_request;
        int major_opcode_of_request;
        int minor_opcode_of_request;
        IcePointer reply;
} IceReplyWaitInfo;
```

If *reply_wait* is not and has a reply or error to return in response to this *reply_wait* (i.e. no callback was generated), then the *reply_ready_ret* argument will be set to

If *reply_wait* is then the caller may also pass for *reply_ready_ret* and be guaranteed that no value will be stored in this pointer.

returns one of the following values:

lw(2.5i) lw(4i). T{ T} T{ No error occurred. T}

T{ T} T{ An IO error occurred. The caller must explicitly close the connection by calling T}

T{ T} T{ The ICE connection has been closed (closing of the connection was deferred because of shutdown negotiation, or because the nesting level was not zero). Do not attempt to access the ICE connection at this point, since it has been freed. T}

11. Ping

To send a *Ping* message to the client on the other side of the ICE connection, call the function.

Status IcePing(*ice_conn*, *ping_reply_proc*, *client_data*)
IceConn *ice_conn*;
IcePingReplyProc *ping_reply_proc*;
IcePointer *client_data*;

ice_conn A valid ICE connection object.

ping_reply_proc The callback to invoke when the *Ping* reply arrives.

client_data This pointer will be passed to the callback.

The return value of is zero for failure, and a positive value for success.

When processes the Ping reply, it will invoke the callback.

typedef void (*IcePingReplyProc)();

void PingReplyProc(*ice_conn*, *client_data*)
IceConn *ice_conn*;
IcePointer *client_data*;

ice_conn The ICE connection object.

client_data The client data specified in the call to

12. Informational Functions

IceConnectStatus IceConnectionStatus(*ice_conn*)
IceConn *ice_conn*;

Returns the status of an ICE connection. The possible return values are:

lw(2i) lw(4i). T{ T} T{ The connection is not valid yet (i.e. authentication is taking place). Only relevant to connections created by T}

T{ T} T{ The connection has been accepted. Only relevant to connections created by T}

T{ T} T{ The connection had been rejected (i.e. authentication failed). Only relevant to connections created by T}

T{ T} T{ An IO error has occurred on the connection. T}

char *IceVendor(*ice_conn*)
  IceConn *ice_conn*;

Returns the ICE library vendor identification for the other side of the connection. The string should be freed with a call to free() when no longer needed.

char *IceRelease(*ice_conn*)
  IceConn *ice_conn*;

Returns the release identification of the ICE library on the other side of the connection. The string should be freed with a call to free() when no longer needed.

int IceProtocolVersion(*ice_conn*)
  IceConn *ice_conn*;

Returns the major version of the ICE protocol on this connection.

int IceProtocolRevision(*ice_conn*)
  IceConn *ice_conn*;

Returns the minor version of the ICE protocol on this connection.

int IceConnectionNumber(*ice_conn*)
  IceConn *ice_conn*;

Returns the file descriptor of this ICE connection.

char *IceConnectionString(*ice_conn*)
  IceConn *ice_conn*;

Returns the network ID of the client which accepted this connection. The string should be freed with a call to free() when no longer needed.

unsigned long IceLastSentSequenceNumber(*ice_conn*)
  IceConn *ice_conn*;

Returns the sequence number of the last message sent on this ICE connection.

unsigned long IceLastReceivedSequenceNumber(*ice_conn*)
  IceConn *ice_conn*;

Returns the sequence number of the last message received on this ICE connection.

Bool IceSwapping(*ice_conn*)
  IceConn *ice_conn*;

Returns if byte swapping is necessary when reading messages on the ICE connection.

IcePointer IceGetContext(*ice_conn*)
    IceConn *ice_conn*;

Returns the context associated with a connection created by

13. ICE Messages

All ICE messages have a standard 8 byte header. The ICElib macros which read and write messages rely on the following naming convention for message headers:

```
CARD8     major_opcode;
CARD8     minor_opcode;
CARD8     data[2];
CARD32    length B32;
```

The 3rd and 4th bytes of the message header can be used as needed. The length field is specified in **units of 8 bytes**.

13.1. Sending ICE Messages

The ICE library maintains an output buffer used for generating messages. Protocol libraries layered on top of ICE may choose to batch messages together and flush the output buffer at appropriate times.

If an IO error has occurred on an ICE connection, all write operations will be ignored. Refer to the section titled *Error Handling* for more discussion on handling IO errors.

To get the size of the ICE output buffer, call the function.

int IceGetOutBufSize(*ice_conn*)
    IceConn *ice_conn*;

ice_conn   A valid ICE connection object.

To flush the ICE output buffer, call the function.

IceFlush(*ice_conn*)
    IceConn *ice_conn*;

ice_conn   A valid ICE connection object.

Note that the output buffer may be implicitly flushed if there is insufficient space to generate a message.

The following macros can be used to generate ICE messages:

IceGetHeader(*ice_conn*, *major_opcode*, *minor_opcode*, *header_size*, *<C_data_type>*, *pmsg*)
    IceConn *ice_conn*;
    int *major_opcode*;
    int *minor_opcode*;
    int *header_size*;
    <C_data_type> **pmsg*;

ice_conn   A valid ICE connection object.

major_opcode   The major opcode of the message.

minor_opcode   The minor opcode of the message.

header_size   The size of the message header (in bytes).

<C_data_type>   The actual C data type of the message header.

pmsg The message header pointer. After this macro is called, the library can store data in the message header.

is used to set up a message header on an ICE connection. It sets the major and minor opcodes of the message, and initializes the message's length to the length of the header. If additional variable length data follows, the message's length field should be updated.

IceGetHeaderExtra(*ice_conn*, *major_opcode*, *minor_opcode*, *header_size*, *extra*, *<C_data_type>*, *pmsg*, *pdata*)
    IceConn *ice_conn*;
    int *major_opcode*;
    int *minor_opcode*;
    int *header_size*;
    int *extra*;
    <C_data_type> **pmsg*;
    char **pdata*;

ice_conn A valid ICE connection object.

major_opcode The major opcode of the message.

minor_opcode The minor opcode of the message.

header_size The size of the message header (in bytes).

extra The size of the extra data beyond the header (in 8 byte units).

<C_data_type> The actual C data type of the message header.

pmsg The message header pointer. After this macro is called, the library can store data in the message header.

pdata Returns a pointer to the ICE output buffer which points immediately after the message header. The variable length data should be stored here. If there was not enough room in the ICE output buffer, *pdata* is set to

is used to generate a message with a fixed (and relatively small) amount of variable length data. The complete message must fit in the ICE output buffer.

IceSimpleMessage(*ice_conn*, *major_opcode*, *minor_opcode*)
    IceConn *ice_conn*;
    int *major_opcode*;
    int *minor_opcode*;

ice_conn A valid ICE connection object.

major_opcode The major opcode of the message.

minor_opcode The minor opcode of the message.

is used to generate a message which is identical in size to the ICE header message, and has no additional data.

IceErrorHeader(*ice_conn*, *offending_major_opcode*, *offending_minor_opcode*, *offending_sequence_num*, *severity*, *error_class*, *data_length*)
    IceConn *ice_conn*;
    int *offending_major_opcode*;
    int *offending_minor_opcode*;
    int *offending_sequence_num*;
    int *severity*;
    int *error_class*;
    int *data_length*;

ice_conn A valid ICE connection object.

offending_major_opcode
The major opcode of the protocol in which an error was detected.

offending_minor_opcode
The minor opcode of the protocol in which an error was detected.

offending_sequence_num
The sequence number of the message that caused the error.

severity or

error_class The error class. See below.

data_length Length of data (in 8 byte units) to be written after the header.

sets up an error message header.

Note that the two clients connected by ICE may be using different major opcodes for a given protocol. The *offending_major_opcode* passed to this macro is the major opcode of the protocol for the client sending the error message.

Generic errors which are common to all protocols have classes in the range 0x8000..0xFFFF. See the *Inter-Client Exchange Protocol* document for more details.

lw(1i) lw(1i). T{ T} T{ 0x8000 T}

T{ T} T{ 0x8001 T}

T{ T} T{ 0x8002 T}

T{ T} T{ 0x8003 T}

Per-protocol errors have classes in the range 0x0000-0x7fff.

To write data to an ICE connection, use the macro. If the data fits into the ICE output buffer, it is copied there. Otherwise, the ICE output buffer is flushed and the data is directly sent.

This macro is used in conjunction with and

```
IceWriteData(ice_conn, bytes, data)
    IceConn ice_conn;
    int bytes;
    char *data;
```

ice_conn A valid ICE connection object.

bytes The number of bytes to write.

data The data to write.

To write data as 16 bit quantities, use the macro.

```
IceWriteData16(ice_conn, bytes, data)
    IceConn ice_conn;
    int bytes;
    short *data;
```

ice_conn A valid ICE connection object.

bytes The number of bytes to write.

data The data to write.

To write data as 32 bit quantities, use the macro.

```
IceWriteData32(ice_conn, bytes, data)
    IceConn ice_conn;
    int bytes;
    long *data;
```

ice_conn A valid ICE connection object.

bytes The number of bytes to write.

data The data to write.

To bypass copying data to the ICE output buffer, use the to directly send data over the network connection. If necessary, the ICE output buffer is first flushed.

```
IceSendData(ice_conn, bytes, (char *) data)
    IceConn ice_conn;
    int bytes;
    char *data;
```

ice_conn A valid ICE connection object.

bytes The number of bytes to send.

data The data to send.

To force 32 or 64 bit alignment, use the macro. A maximum of 7 pad bytes can be specified.

```
IceWritePad(ice_conn, bytes)
    IceConn ice_conn;
    int bytes;
```

ice_conn A valid ICE connection object.

bytes The number of pad bytes.

13.2. Reading ICE Messages

The ICE library maintains an input buffer used for reading messages. If the ICE library chooses to perform non-blocking reads (this is implementation dependent), then for every read operation that it makes, zero or more complete messages may be read into the input buffer. As a result, for all of the macros described in this section which "read" messages, an actual read operation will occur on the connection only if the data is not already present in the input buffer.

To get the size of the ICE input buffer, call the function.

```
int IceGetInBufSize(ice_conn)
    IceConn ice_conn;
```

ice_conn A valid ICE connection object.

When reading messages, care must be taken to check for IO errors. If any IO error occurs in reading any part of a message, the message should be thrown out. After using any of the macros described below for reading messages, the macro can be used to check if an IO error occurred on the connection. After an IO error has occurred on an ICE connection, all read operations will be ignored. Refer to the section titled *Error Handling* for more discussion on handling IO errors.

```
Bool IceValidIO(ice_conn)
   IceConn ice_conn;
```

The following macros can be used to read ICE messages:

```
IceReadSimpleMessage(ice_conn, <C_data_type>, pmsg)
    IceConn ice_conn;
    <C_data_type> *pmsg;
```

ice_conn A valid ICE connection object.

<C_data_type> The actual C data type of the message header.

pmsg This pointer is set to the message header.

is used for messages which are identical in size to the 8 byte ICE header, but use the spare 2 bytes in the header to encode additional data. Note that the ICE library always reads in these first 8 bytes so it can obtain the major opcode of the message. simply returns a pointer to these 8 bytes, it does not actually read any data into the input buffer.

For a message with variable length data, there are two ways of reading the message. One method involves reading the complete message in one pass using The second method involves reading the message header (note that this may be larger than the 8 byte ICE header), then reading the variable length data in chunks (see and

```
IceReadCompleteMessage(ice_conn, header_size, <C_data_type>, pmsg, pdata)
    IceConn ice_conn;
    int header_size;
    <C_data_type> *pmsg;
    char *pdata;
```

ice_conn A valid ICE connection object.

header_size The size of the message header (in bytes).

<C_data_type> The actual C data type of the message header.

pmsg This pointer is set to the message header.

pdata This pointer is set to the variable length data of the message.

If the ICE input buffer has sufficient space, will read the complete message into the ICE input buffer. Otherwise, a buffer will be allocated to hold the variable length data. After the call, the *pdata* argument should be checked against to make sure that there was sufficient memory to allocate the buffer.

After calling and processing the message, should be called.

```
IceDisposeCompleteMessage(ice_conn, pdata)
    IceConn ice_conn;
    char *pdata;
```

ice_conn A valid ICE connection object.

pdata The pointer to the variable length data returned in

If a buffer had to be allocated to hold the variable length data (because it didn't fit in the ICE input buffer), it is freed here by ICElib.

```
IceReadMessageHeader(ice_conn, header_size, <C_data_type>, pmsg)
    IceConn ice_conn;
    int header_size;
    <C_data_type> *pmsg;
```

ice_conn A valid ICE connection object.

header_size The size of the message header (in bytes).

<C_data_type> The actual C data type of the message header.

pmsg This pointer is set to the message header.

reads just the message header. The rest of the data should be read with the family of macros. This method of reading a message should be used when the variable length data must be read in chunks.

To read data directly into a user supplied buffer, use the macro.

IceReadData(*ice_conn*, *bytes*, *pdata*)
    IceConn *ice_conn*;
    int *bytes*;
    char **pdata*;

ice_conn	A valid ICE connection object.
bytes	The number of bytes to read.
pdata	The data is read into this user supplied buffer.

To read data as 16 bit quantities, use the macro.

IceReadData16(*ice_conn*, *swap*, *bytes*, *pdata*)
    IceConn *ice_conn*;
    Bool *swap*;
    int *bytes*;
    short **pdata*;

ice_conn	A valid ICE connection object.
swap	If the values will be byte swapped.
bytes	The number of bytes to read.
pdata	The data is read into this user supplied buffer.

To read data as 32 bit quantities, use the macro.

IceReadData32(*ice_conn*, *swap*, *bytes*, *pdata*)
    IceConn *ice_conn*;
    Bool *swap*;
    int *bytes*;
    long **pdata*;

ice_conn	A valid ICE connection object.
swap	If the values will be byte swapped.
bytes	The number of bytes to read.
pdata	The data is read into this user supplied buffer.

To force 32 or 64 bit alignment, use the macro. A maximum of 7 pad bytes can be specified.

IceReadPad(*ice_conn*, *bytes*)
    IceConn *ice_conn*;
    int *bytes*;

ice_conn	A valid ICE connection object.
bytes	The number of pad bytes.

14. Error Handling

There are two default error handlers in ICElib: one to handle typically fatal conditions (for example, a connection dying because a machine crashed) and one to handle ICE-specific protocol errors. These error handlers can be changed to user-supplied routines if you prefer your own error handling and can be changed as often as you like.

To set the ICE error handler, use

IceErrorHandler IceSetErrorHandler(*handler*)
  IceErrorHandler *handler*;

handler The ICE error handler. Pass to restore the default handler.

returns the previous error handler.

The ICE error handler is invoked when an unexpected ICE protocol error (major opcode 0) is encountered. The action of the default handler is to print an explanatory message to stderr and if the severity is fatal, call exit() with a non-zero value. If exiting is undesirable, the application should register its own error handler.

Note that errors in other protocol domains should be handled by their respective libraries (these libraries should have their own error handlers).

An ICE error handler has the type of

typedef void (*IceErrorHandler)(); void ErrorHandler(*ice_conn*, *swap*, *offending_minor_opcode*, *offending_sequence_num*, *error_class*, *severity*, *values*)
  IceConn *ice_conn*;
  Bool *swap*;
  int *offending_minor_opcode*;
  unsigned long *offending_sequence_num*;
  int *error_class*;
  int *severity*;
  IcePointer *values*;

ice_conn The ICE connection object.

swap A flag which indicates if the *values* need byte swapping.

offending_minor_opcode
The ICE minor opcode of the offending message.

offending_sequence_num
The sequence number of the offending message.

error_class The error class of the offending message.

severity or

values Any additional error values specific to the minor opcode and class.

The following error classes are defined at the ICE level. Refer to the *Inter-Client Exchange Protocol* document for more details.

or

To handle fatal I/O errors, use

IceIOErrorHandler IceSetIOErrorHandler(*handler*)
  IceIOErrorHandler *handler*;

handler The I/O error handler. Pass to restore the default handler.

returns the previous IO error handler.

An ICE I/O error handler has the type of typedef void (*IceIOErrorHandler)();

void IOErrorHandler(*ice_conn*)
  IceConn *ice_conn*;

ice_conn The ICE connection object.

There are two ways of handling IO errors in ICElib.

In the first model, the IO error handler does whatever is necessary to respond to the IO error and then returns, but it does not call The ICE connection is given a "bad IO" status, and all future reads and writes to the connection are ignored. The next time is called it will return a status of At that time, the application should call

In the second model, the IO error handler does call and then uses the longjmp() call to get back to the application's main event loop. setjmp() and longjmp() may not work properly on all platforms and special care must be taken to avoid memory leaks, so this second model is less desirable.

Before the application I/O error handler is invoked, protocol libraries that were interested in being notified of I/O errors will have their handlers invoked. This handler is set up in the protocol registration functions (see and and could be used to clean up state specific to the protocol.

typedef void (*IceIOErrorProc)();

void IOErrorProc(*ice_conn*)
  IceConn *ice_conn*;

ice_conn The ICE connection object.

Note that every callback must return. This is required because each active protocol must be notified of the broken connection, and the application IO error handler must be invoked afterwards.

15. Multi-Threading Support

To declare that multiple threads in an application will be using the ICE library, call

Status IceInitThreads()

The function must be the first ICElib function a multi-threaded program calls. It must complete before any other ICElib call is made. IceInitThreads returns a non-zero status if and only if it was able to successfully initialize the threads package. It is safe to call this function more than once, although the threads package will only be initialized once.

Protocol libraries layered on top of ICElib will have to lock critical sections of code that access an ICE connection (for example, when generating messages). Two calls, which are generally implemented as macros, are provided:

IceLockConn(*ice_conn*)
  IceConn *ice_conn*;

IceUnlockConn(*ice_conn*)
  IceConn *ice_conn*;

ice_conn The ICE connection.

To keep an ICE connection locked across several ICElib calls, applications use and

void IceAppLockConn(*ice_conn*)
   IceConn *ice_conn*;

ice_conn   The ICE connection to lock.

The function completely locks out other threads from ICElib until is called. Other threads attempting to use ICElib will block. If the program has not previously called has no effect.

void IceAppUnlockConn(*ice_conn*)
   IceConn *ice_conn*;

ice_conn   The ICE connection to unlock.

The function allows other threads to complete ICElib calls which were blocked by a previous call to from this thread. If the program has not previously called has no effect.

16. Miscellaneous Functions

To allocate scratch space (for example, when generating messages with variable data), use the function. Each ICE connection has one scratch space associated with it. The scratch space starts off as empty and grows as needed. The contents of the scratch space is not guaranteed to be preserved after any ICElib function is called.

char *IceAllocScratch(*ice_conn*, *size*)
   IceConn *ice_conn*;
   unsigned long *size*;

ice_conn   A valid ICE connection object.

size   The number of bytes required.

The memory returned by should *not* be freed by the caller! The ICE library will free the memory when the ICE connection is closed.

Appendix A

Authentication Utility Functions

As discussed in this document, the means by which authentication data is obtained by the ICE library (for *Connection Setups* or *Protocol Setups*) is implementation dependent.†

This appendix describes some utility functions which manipulate an ICE authority file. The authority file can be used to pass authentication data between clients.

The basic operations on the .ICEauthority file are : get file name, lock, unlock, read entry, write entry, and search for entry. These are fairly low level operations, and it is expected that a program like "iceauth" would exist to add, remove, and display entries in the file.

In order to use these utility functions, the *<X11/ICE/ICEutil.h>* header file must be included.

An entry in the .ICEauthority file is defined by the following data structure:

```
typedef struct {
        char *protocol_name;
        unsigned short protocol_data_length;
        char *protocol_data;
        char *network_id;
        char *auth_name;
        unsigned short auth_data_length;
        char *auth_data;
} IceAuthFileEntry;
```

The *protocol_name* is either "ICE" for connection setup authentication, or the sub-protocol name, such as "XSMP". For each entry, protocol specific data can be specified in the *protocol_data* field. This can be used to search for old entries that need to be removed from the file.

network_id is the network ID of the client accepting authentication (for example, the network ID of a session manager). A network ID has the form...

lw(0.25i) lw(2.5i) lw(1i). tcp/<hostname>:<portnumber>or
decnet/<hostname>::<objname> or local/<hostname>:<path>

auth_name is the name of the authentication method. *auth_data* is the actual authentication data, and *auth_data_length* is the number of bytes in the data.

To obtain the default authorization file name, call the function.

```
char *IceAuthFileName()
```

If the ICEAUTHORITY environment variable if set, this value is returned. Otherwise, the default authorization file name is $HOME/.ICEauthority. This name is statically allocated and should not be freed.

In order to synchronously update the authorization file, the file must be locked with a call to This function takes advantage of the fact that the "link" system call will fail if the name of the new link already exists.

† The X Consortium's ICElib implementation assumes the presence of an ICE authority file.

int IceLockAuthFile(*file_name*, *retries*, *timeout*, *dead*)
  char **file_name*;
  int *retries*;
  int *timeout*;
  long *dead*;

file_name The authorization file to lock.

retries The number of retries.

timeout The number of seconds before each retry.

dead If a lock already exists that is *dead* seconds old, it is broken. A value of zero is used to unconditionally break an old lock.

One of three values is returned:

lw(2i) lw(5i). T{ T} T{ The lock succeeded. T}

T{ T} T{ A system error occurred. errno may prove useful. T}

T{ T} T{ *retries* attempts failed. T}

To unlock an authorization file, call the function.

void IceUnlockAuthFile(*file_name*)
  char **file_name*;

file_name The authorization file to unlock.

To read the next entry in an authorization file, call the function.

IceAuthFileEntry *IceReadAuthFileEntry(*auth_file*)
  FILE **auth_file*;

auth_file The authorization file.

Note that it is the responsibility of the application to open the file for reading before calling this function. If an error is encountered, or there are no more entries to read, is returned.

Entries should be free with a call to (see below).

To write an entry in an authorization file, call the function.

Status IceWriteAuthFileEntry(*auth_file*, *entry*)
  FILE **auth_file*;
  IceAuthFileEntry **entry*;

auth_file The authorization file.

entry The entry to write.

Note that it is the responsibility of the application to open the file for writing before calling this function. The function returns a non-zero status if the operation was successful.

To search the default authorization file for an entry matching a given protocol_name/network_id/auth_name tuple, call the function.

IceAuthFileEntry *IceGetAuthFileEntry(*protocol_name*, *network_id*, *auth_name*)
    char ***protocol_name*;
    char ***network_id*;
    char ***auth_name*;

protocol_name	The name of the protocol to search on.
network_id	The network ID to search on.
auth_name	The authentication method to search on.

If fails to find such an entry, is returned.

To free an entry returned by or call the function.

void IceFreeAuthFileEntry(*entry*)
    IceAuthFileEntry ***entry*;

entry	The entry to free.

Appendix B

MIT-MAGIC-COOKIE-1 Authentication

The X Consortium's ICElib implementation supports a simple MIT-MAGIC-COOKIE-1 authentication scheme using the authority file utilities described in Appendix A.

In this model, an application such as a session manager, obtains a magic cookie by calling and then stores it in the user's local .ICEauthority file so that local clients can connect. In order to allow remote clients to connect, some remote execution mechanism should be used to store the magic cookie in the user's .ICEauthority file on a remote machine.

In addition to storing the magic cookie in the .ICEauthority file, the application needs to call the function in order to store the magic cookie in memory. When it comes time for the MIT-MAGIC-COOKIE-1 authentication procedure to accept or reject the connection, it will compare the magic cookie presented by the requestor to the magic cookie in memory.

```
char *IceGenerateMagicCookie(length)
   int length;
```

length The desired length of the magic cookie.

The magic cookie returned will be null terminated. If memory can not be allocated for the magic cookie, the function will return Otherwise, the magic cookie should be freed with a call to free().

In order to store the authentication data in memory, call the function. Currently, this function is only used for MIT-MAGIC-COOKIE-1 authentication, but it may be used for additional authentication methods in the future.

```
void IceSetPaAuthData(num_entries, entries)
   int num_entries;
   IceAuthDataEntry *entries;
```

num_entries The number of authentication data entries.

entries The list of authentication data entries.

Each entry has associated with it a protocol name (e.g. "ICE" for ICE connection setup authentication, "XSMP" for session management authentication), a network ID for the "accepting" client, an authentication name (e.g. MIT-MAGIC-COOKIE-1), and authentication data. The ICE library will merge these entries with previously set entries, based on the (protocol_name, network_id, auth_name) tuple.

```
typedef struct {
        char *protocol_name;
        char *network_id;
        char *auth_name;
        unsigned short auth_data_length;
        char *auth_data;
} IceAuthDataEntry;
```