Inter-Client Exchange (ICE) Protocol

X Consortium Standard

Robert Scheifler
Jordan Brown
Quarterdeck Office Systems

Inter-Client Exchange (ICE) Protocol: X Consortium Standard

by Robert Scheifler
Jordan Brown
Quarterdeck Office Systems

X Version 11, Release 6.4

Version 1.0
Copyright © 1993 X Consortium
Copyright © 1994 X Consortium

Abstract

There are numerous possible protocols that can be used for communication among clients. They have many similarities and common needs, including authentication, version negotiation, data typing, and connection management. The *Inter-Client Exchange* (ICE) protocol is intended to provide a framework for building such protocols. Using ICE reduces the complexity of designing new protocols and allows the sharing of many aspects of the implementation.

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.

Table of Contents

Chapter 1. Purpose and Goals

In discussing a variety of protocols -- existing, under development, and hypothetical -- it was noted that they have many elements in common. Most protocols need mechanisms for authentication, for version negotiation, and for setting up and taking down connections. There are also cases where the same two parties need to talk to each other using multiple protocols. For example, an embedding relationship between two parties is likely to require the simultaneous use of session management, data transfer, focus negotiation, and command notification protocols. While these are logically separate protocols, it is desirable for them to share as many pieces of implementation as possible.

The *Inter-Client Exchange* (ICE) protocol provides a generic framework for building protocols on top of reliable, byte-stream transport connections. It provides basic mechanisms for setting up and shutting down connections, for performing authentication, for negotiating versions, and for reporting errors. The protocols running within an ICE connection are referred to here as *subprotocols.* ICE provides facilities for each subprotocol to do its own version negotiation, authentication, and error reporting. In addition, if two parties are communicating using several different subprotocols, ICE will allow them to share the same transport layer connection.

Chapter 2. Overview of the Protocol

Through some mechanism outside ICE, two parties make themselves known to each other and agree that they would like to communicate using an ICE subprotocol. ICE assumes that this negotation includes some notion by which the parties will decide which is the *Qoriginating*U party and which is the *Qanswering*U party. The negotiation will also need to provide the originating party with a name or address of the answering party. Examples of mechanisms by which parties can make themselves known to each other are the X selection mechanism, environment variables, and shared files.

The originating party first determines whether there is an existing ICE connection between the two parties. If there is, it can re-use the existing connection and move directly to the setup of the subprotocol. If no ICE connection exists, the originating party will open a transport connection to the answering party and will start ICE connection setup.

The ICE connection setup dialog consists of three major parts: byte order exchange, authentication, and connection information exchange. The first message in each direction is a `ByteOrder` message telling which byte order will be used by the sending party in messages that it sends. After that, the originating party sends a `ConnectionSetup` message giving information about itself (vendor name and release number) and giving a list of ICE version numbers it is capable of supporting and a list of authentication schemes it is willing to accept. Authentication is optional. If no authentication is required, the answering party responds with a `ConnectionReply` message giving information about itself, and the connection setup is complete.

If the connection setup is to be authenticated, the answering party will respond with an `AuthenticationRequired` message instead of a `ConnectionReply` message. The parties then exchange `AuthenticationReply` and `AuthenticationNextPhase` messages until authentication is complete, at which time the answering party finally sends its `ConnectionReply` message.

Once an ICE connection is established (or an existing connection reused), the originating party starts subprotocol negotiation by sending a `ProtocolSetup` message. This message gives the name of the subprotocol that the parties have agreed to use, along with the ICE major opcode that the originating party has assigned to that subprotocol. Authentication can also occur for the subprotocol, independently of authentication for the connection. Subprotocol authentication is optional. If there is no subprotocol authentication, the answering party responds with a `ProtocolReply` message, giving the ICE major opcode that it has assigned for the subprotocol.

Subprotocols are authenticated independently of each other, because they may have differing security requirements. If there is authentication for this particular subprotocol, it takes place before the answering party emits the `ProtocolReply` message, and it uses the `AuthenticationRequired` `AuthenticationReply` and `AuthenticationNextPhase` messages, just as for the connection authentication. Only when subprotocol authentication is complete does the answering party send its `ProtocolReply` message.

When a subprotocol has been set up and authenticated, the two parties can communicate using messages defined by the subprotocol. Each message has two opcodes: a major opcode and a minor opcode. Each party will send messages using the major opcode it has assigned in its `ProtocolSetup` or `ProtocolReply` message. These opcodes will, in general, not be the same. For a particular subprotocol, each party will need to keep track of two major opcodes: the major opcode it uses when it sends messages, and the major opcode it expects to see in messages it receives. The minor opcode values and semantics are defined by each individual subprotocol.

Each subprotocol will have one or more messages whose semantics are that the subprotocol is to be shut down. Whether this is done unilaterally or is performed through negotiation is defined by each subprotocol. Once a subprotocol is shut down, its major opcodes are removed from use; no further messages on this subprotocol should be sent until the opcode is reestablished with `ProtocolSetup`

ICE has a facility to negotiate the closing of the connection when there are no longer any active subprotocols. When either party decides that no subprotocols are active, it can send a `WantToClose` message. If the other party agrees to close the connection, it can simply do so. If the other party wants to keep the connection open, it can indicate its desire by replying with a `NoClose` message.

It should be noted that the party that initiates the connection isn't necessarily the same as the one that initiates setting up a subprotocol. For example, suppose party A connects to party B. Party A will issue the `ConnectionSetup` message and party B will respond with a `ConnectionReply` message. (The authentication steps are omitted here for brevity.) Typically, party A will also issue the `ProtocolSetup` message and expect a `ProtocolReply` from party B. Once the connection is established, however, either party may initiate the negotiation of a subprotocol. Continuing this example, party B may decide that it needs to set up a subprotocol for communication with party A. Party B would issue the `ProtocolSetup` message and expect a `ProtocolReply` from party A.

Chapter 3. Data Types

ICE messages contain several types of data. Byte order is negotiated in the initial connection messages; in general data is sent in the sender's byte order and the receiver is required to swap it appropriately. In order to support 64-bit machines, ICE messages are padded to multiples of 8 bytes. All messages are designed so that fields are *Qnaturally*U aligned on 16-, 32-, and 64-bit boundaries. The following formula gives the number of bytes necessary to pad E bytes to the next multiple of b:

$$\mathrm{pad}(E, b) = (b - (E \bmod b)) \bmod b$$

Primitive Types

Type Name	**Description**
CARD8	8-bit unsigned integer
CARD16	16-bit unsigned integer
CARD32	32-bit unsigned integer
BOOL	`False` or `True`
LPCE	A character from the X Portable Character Set in Latin Portable Character Encoding

Complex Types

Type Name	**Type**
VERSION	[Major, minor: CARD16]
STRING	LISTofLPCE

LISTof<type> denotes a counted collection of <type>. The exact encoding varies depending on the context; see the encoding section.

Message Format

All ICE messages include the following information:

Field Type	**Description**
CARD8	protocol major opcode
CARD8	protocol minor opcode
CARD32	length of remaining data in 8-byte units

The fields are as follows:

Protocol major opcode	This specifies what subprotocol the message is intended for. Major opcode 0 is reserved for ICE control messages. The major opcodes of other subproto-

	cols are dynamically assigned and exchanged at protocol negotiation time.
Protocol minor opcode	This specifies what protocol-specific operation is to be performed. Minor opcode 0 is reserved for Errors; other values are protocol-specific.
Length of data in 8-byte units	This specifies the length of the information following the first 8 bytes. Each message-type has a different format, and will need to be separately length-checked against this value. As every data item has either an explicit length, or an implicit length, this can be easily accomplished. Messages that have too little or too much data indicate a serious protocol failure, and should result in a `BadLength` error.

Chapter 4. Overall Protocol Description

Every message sent in a given direction has an implicit sequence number, starting with 1. Sequence numbers are global to the connection; independent sequence numbers are *not* maintained for each protocol.

Messages of a given major-opcode (i.e., of a given protocol) must be responded to (if a response is called for) in order by the receiving party. Messages from different protocols can be responded to in arbitrary order.

Minor opcode 0 in every protocol is for reporting errors. At most one error is generated per request. If more than one error condition is encountered in processing a request, the choice of which error is returned is implementation-dependent.

`Error`

offending-minor-opcode:	CARD8
severity:	{CanContinue, `FatalToProtocol` `FatalToConnection`
sequence-number:	CARD32
class:	CARD16
value(s):	<dependent on major/minor opcode and class>

This message is sent to report an error in response to a message from any protocol. The `Error` message exists in all protocol major-opcode spaces; it is minor-opcode zero in every protocol. The minor opcode of the message that caused the error is reported, as well as the sequence number of that message. The severity indicates the sender's behavior following the identification of the error. `CanContinue` indicates the sender is willing to accept additional messages for this protocol. `FatalToProcotol` indicates the sender is unwilling to accept further messages for this protocol but that messages for other protocols may be accepted. `FatalToConnection` indicates the sender is unwilling to accept any further messages for any protocols on the connection. The sender is required to conform to specified severity conditions for generic and ICE (major opcode 0) errors; see the section called “Generic Error Classes” and the section called “ICE Error Classes”. . The class defines the generic class of error. Classes are specified separately for each protocol (numeric values can mean different things in different protocols). The error values, if any, and their types vary with the specific error class for the protocol.

Chapter 5. ICE Control Subprotocol -- Major Opcode 0

Each of the ICE control opcodes is described below. Most of the messages have additional information included beyond the description above. The additional information is appended to the message header and the length field is computed accordingly.

In the following message descriptions, *QExpected errors*U indicates errors that may occur in the normal course of events. Other errors (in particular `BadMajor` `BadMinor` `BadState` `BadLength` `BadValue` `ProtocolDuplicate` and `MajorOpcodeDuplicate` might occur, but generally indicate a serious implementation failure on the part of the errant peer.

`ByteOrder`

byte-order:	{MSBfirst, `LSBfirst`

Both parties must send this message before sending any other, including errors. This message specifies the byte order that will be used on subsequent messages sent by this party.

Note

Note: If the receiver detects an error in this message, it must be sure to send its own `ByteOrder` message before sending the `Error`.

`ConnectionSetup`

versions:	LISTofVERSION
must-authenticate:	BOOL
authentication-protocol-names:	LISTofSTRING
vendor:	STRING
release:	STRING
Responses:	`ConnectionReply`, `AuthenticationRequired` (See note)
Expected errors:	`NoVersion`, `SetupFailed`, `NoAuthentication`, `AuthenticationRejected`, `AuthenticationFailed`

The party that initiates the connection (the one that does the "connect()") must send this message as the second message (after `ByteOrder` on startup.

Versions gives a list, in decreasing order of preference, of the protocol versions this party is capable of speaking. This document specifies major version 1, minor version 0.

If must-authenticate is `True` the initiating party demands authentication; the accepting party *must* pick an authentication scheme and use it. In this case, the only valid response is `AuthenticationRequired`

If must-authenticate is `False` the accepting party may choose an authentication mechanism, use a host-address-based authentication scheme, or skip authentication. When must-authenticate is `False` `ConnectionReply` and `AuthenticationRequired` are both valid responses. If a host-address-based authentication scheme is used, `AuthenticationRejected` and `AuthenticationFailed` errors are possible.

Authentication-protocol-names specifies a (possibly null, if must-authenticate is `False` list of authentication protocols the party is willing to perform. If must-authenticate is `True` presumably the party will offer only authentication mechanisms allowing mutual authentication.

Vendor gives the name of the vendor of this ICE implementation.

Release gives the release identifier of this ICE implementation.

`AuthenticationRequired`

authentication-protocol-index:	CARD8
data:	<specific to authentication protocol>
Response:	`AuthenticationReply`
Expected errors:	`AuthenticationRejected`, `AuthenticationFailed`

This message is sent in response to a `ConnectionSetup` or `ProtocolSetup` message to specify that authentication is to be done and what authentication mechanism is to be used.

The authentication protocol is specified by a 0-based index into the list of names given in the `ConnectionSetup` or `ProtocolSetup` Any protocol-specific data that might be required is also sent.

`AuthenticationReply`

data:	<specific to authentication protocol>
Responses:	`AuthenticationNextPhase`, `ConnectionReply`, `ProtocolReply`
Expected errors:	`AuthenticationRejected`, `AuthenticationFailed`, `SetupFailed`

This message is sent in response to an `AuthenticationRequired` or `AuthenticationNextPhase` message, to supply authentication data as defined by the authentication protocol being used.

Note that this message is sent by the party that initiated the current negotiation -- the party that sent the `ConnectionSetup` or `ProtocolSetup` message.

`AuthenticationNextPhase` indicates that more is to be done to complete the authentication. If the authentication is complete, `ConnectionReply` is appropriate if the current authentication handshake is the result of a `ConnectionSetup` and a `ProtocolReply` is appropriate if it is the result of a `ProtocolSetup`.

`AuthenticationNextPhase`

data: <specific to authentication protocol>

Response: `AuthenticationReply`

Expected errors: `AuthenticationRejected`, `AuthenticationFailed`

This message is sent in response to an `AuthenticationReply` message, to supply authentication data as defined by the authentication protocol being used.

`ConnectionReply`

version-index: CARD8

vendor: STRING

release: STRING

This message is sent in response to a `ConnectionSetup` or `AuthenticationReply` message to indicate that the authentication handshake is complete.

Version-index gives a 0-based index into the list of versions offered in the `ConnectionSetup` message; it specifies the version of the ICE protocol that both parties should speak for the duration of the connection.

Vendor gives the name of the vendor of this ICE implementation.

Release gives the release identifier of this ICE implementation.

`ProtocolSetup`

protocol-name: STRING

major-opcode: CARD8

versions: LISTofVERSION

vendor: STRING

release: STRING

must-authenticate: BOOL

authentication-protocol-names: LISTofSTRING

Responses: `AuthenticationRequired`, `ProtocolReply`

Expected errors: `UnknownProtocol`, `NoVersion`, `SetupFailed`, `NoAuthentication`, `AuthenticationRejected`, `AuthenticationFailed`

This message is used to initiate negotiation of a protocol and establish any authentication specific to it.

Protocol-name gives the name of the protocol the party wishes to speak.

Major-opcode gives the opcode that the party will use in messages it sends.

Versions gives a list of version numbers, in decreasing order of preference, that the party is willing to speak.

Vendor and release are identification strings with semantics defined by the specific protocol being negotiated.

If must-authenticate is `True`, the initiating party demands authentication; the accepting party *must* pick an authentication scheme and use it. In this case, the only valid response is `AuthenticationRequired`

If must-authenticate is `False`, the accepting party may choose an authentication mechanism, use a host-address-based authentication scheme, or skip authentication. When must-authenticate is `False`, `ProtocolReply` and `AuthenticationRequired` are both valid responses. If a host-address-based authentication scheme is used, `AuthenticationRejected` and `AuthenticationFailed` errors are possible.

Authentication-protocol-names specifies a (possibly null, if must-authenticate is `False` list of authentication protocols the party is willing to perform. If must-authenticate is `True` presumably the party will offer only authentication mechanisms allowing mutual authentication.

`ProtocolReply`

major-opcode:	CARD8
version-index:	CARD8
vendor:	STRING
release:	STRING

This message is sent in response to a `ProtocolSetup` or `AuthenticationReply` message to indicate that the authentication handshake is complete.

Major-opcode gives the opcode that this party will use in messages that it sends.

Version-index gives a 0-based index into the list of versions offered in the `ProtocolSetup` message; it specifies the version of the protocol that both parties should speak for the duration of the connection.

Vendor and release are identification strings with semantics defined by the specific protocol being negotiated.

`Ping`

Response: `PingReply`

This message is used to test if the connection is still functioning.

`PingReply`

This message is sent in response to a `Ping` message, indicating that the connection is still functioning.

`WantToClose`

Responses: `WantToClose`, `NoClose`, `ProtocolSetup`

This message is used to initiate a possible close of the connection. The sending party has noticed that, as a result of mechanisms specific to each protocol, there are no active protocols left. There are four possible scenarios arising from this request:

1. The receiving side noticed too, and has already sent a `WantToClose` On receiving a `WantToClose` while already attempting to shut down, each party should simply close the connection.
2. The receiving side hasn't noticed, but agrees. It closes the connection.
3. The receiving side has a `ProtocolSetup` "in flight," in which case it is to ignore `WantToClose` and the party sending `WantToClose` is to abandon the shutdown attempt when it receives the `ProtocolSetup`
4. The receiving side wants the connection kept open for some reason not specified by the ICE protocol, in which case it sends `NoClose`

See the state transition diagram for additional information.

`NoClose`

This message is sent in response to a `WantToClose` message to indicate that the responding party does not want the connection closed at this time. The receiving party should not close the connection. Either party may again initiate `WantToClose` at some future time.

Generic Error Classes

These errors should be used by all protocols, as applicable. For ICE (major opcode 0), `FatalToProtocol` should be interpreted as `FatalToConnection`.

`BadMinor`

offending-minor-opcode:	<any>
severity:	`FatalToProtocol` or `CanContinue` (protocol's discretion)
values:	(none)

Received a message with an unknown minor opcode.

`BadState`

offending-minor-opcode:	<any>
severity:	`FatalToProtocol` or `CanContinue` (protocol's discretion)
values:	(none)

Received a message with a valid minor opcode which is not appropriate for the current state of the protocol.

`BadLength`

offending-minor-opcode: <any>

severity: `FatalToProtocol` or `CanContinue` (protocol's discretion)

values: (none)

Received a message with a bad length. The length of the message is longer or shorter than required to contain the data.

`BadValue`

offending-minor-opcode: <any>

severity: `CanContinue`

values: CARD32 Byte offset to offending value in offending message. CARD32 Length of offending value. <varies> Offending value

Received a message with a bad value specified.

ICE Error Classes

These errors are all major opcode 0 errors.

`BadMajor`

offending-minor-opcode: <any>

severity: `CanContinue`

values: CARD8 Opcode

The opcode given is not one that has been registered.

`NoAuthentication`

offending-minor-opcode: `ConnectionSetup`, `ProtocolSetup`

severity: `ConnectionSetup` \(-> `FatalToConnection` `ProtocolSetup` \(-> `FatalToProtocol`

values: (none)

None of the authentication protocols offered are available.

`NoVersion`

offending-minor-opcode: `ConnectionSetup`, `ProtocolSetup`

severity: `ConnectionSetup` \(-> `FatalToConnection` `ProtocolSetup` \(-> `FatalToProtocol`

values: (none)

None of the protocol versions offered are available.

`SetupFailed`

offending-minor-opcode: `ConnectionSetup`, `ProtocolSetup`, `AuthenticationReply`

severity: `ConnectionSetup` \(-> `FatalToConnection` `ProtocolSetup` \(-> `FatalToProtocol` `AuthenticationReply` \(-> `FatalToConnection` if authenticating a connection, otherwise `FatalToProtocol`

values: STRING reason

The sending side is unable to accept the new connection or new protocol for a reason other than authentication failure. Typically this error will be a result of inability to allocate additional resources on the sending side. The reason field will give a human-interpretable message providing further detail on the type of failure.

`AuthenticationRejected`

offending-minor-opcode: `AuthenticationReply`, `AuthenticationRequired`, `AuthenticationNextPhase`

severity: `FatalToProtocol`

values: STRING reason

Authentication rejected. The peer has failed to properly authenticate itself. The reason field will give a human-interpretable message providing further detail.

`AuthenticationFailed`

offending-minor-opcode: `AuthenticationReply`, `AuthenticationRequired`, `AuthenticationNextPhase`

severity: `FatalToProtocol`

values: STRING reason

Authentication failed. `AuthenticationFailed` does not imply that the authentication was rejected, as `AuthenticationRejected` does. Instead it means that the sender was unable to complete the authentication for some other reason. (For instance, it may have been unable to contact an authentication server.) The reason field will give a human-interpretable message providing further detail.

`ProtocolDuplicate`

offending-minor-opcode: `ProtocolSetup`

severity: `FatalToProtocol` (but see note)

values: STRING protocol name

The protocol name was already registered. This is fatal to the "new" protocol being set up by `ProtocolSetup` but it does not affect the existing registration.

`MajorOpcodeDuplicate`

offending-minor-opcode:	`ProtocolSetup`
severity:	`FatalToProtocol` (but see note)
values:	CARD8 opcode

The major opcode specified was already registered. This is fatal to the *Qnew*U protocol being set up by `ProtocolSetup` but it does not affect the existing registration.

`UnknownProtocol`

offending-minor-opcode:	`ProtocolSetup`
severity:	`FatalToProtocol`
values:	STRING protocol name

The protocol specified is not supported.

Chapter 6. State Diagrams

Here are the state diagrams for the party that initiates the connection:

start:
  connect to other end, send `ByteOrder` `ConnectionSetup` -> *conn_wait*

conn_wait:
  receive `ConnectionReply` -> *stasis*
  receive `AuthenticationRequired` -> *conn_auth1*
  receive `Error` -> *quit*
  receive <other>, send `Error` -> *quit*

conn_auth1:
  if good auth data, send `AuthenticationReply` -> *conn_auth2*
  if bad auth data, send `Error` -> *quit*

conn_auth2:
  receive `ConnectionReply` -> *stasis*
  receive `AuthenticationNextPhase` -> *conn_auth1*
  receive `Error` -> *quit*
  receive <other>, send `Error` -> *quit*

Here are top-level state transitions for the party that accepts connections.

listener:
  accept connection -> *init_wait*

init_wait:
  receive `ByteOrder` `ConnectionSetup` -> *auth_ask*
  receive <other>, send `Error` -> *quit*

auth_ask:
  send `ByteOrder` `ConnectionReply`
-> *stasis*

  send `AuthenticationRequired` -> *auth_wait*

  send `Error` -> *quit*

auth_wait:
  receive `AuthenticationReply` -> *auth_check*

  receive <other>, send `Error` -> *quit*

auth_check:
  if no more auth needed, send `ConnectionReply` -> *stasis*
  if good auth data, send `AuthenticationNextPhase` -> *auth_wait*
  if bad auth data, send `Error` -> *quit*

Here are the top-level state transitions for all parties after the initial connection establishment subprotocol.

Note

Note: this is not quite the truth for branches out from stasis, in that multiple conversations can be interleaved on the connection.

stasis:
- send `ProtocolSetup` -> *proto_wait*
- receive `ProtocolSetup` -> *proto_reply*
- send `Ping` -> *ping_wait*
- receive `Ping` send `PingReply` -> *stasis*
- receive `WantToClose` -> *shutdown_attempt*
- receive <other>, send `Error` -> *stasis*
- all protocols shut down, send `WantToClose` -> *close_wait*

proto_wait:
- receive `ProtocolReply` -> *stasis*
- receive `AuthenticationRequired` -> *give_auth1*
- receive `Error` give up on this protocol -> *stasis*
- receive `WantToClose` -> *proto_wait*

give_auth1:
- if good auth data, send `AuthenticationReply` -> *give_auth2*
- if bad auth data, send `Error` give up on this protocol -> *stasis*
- receive `WantToClose` -> *give_auth1*

give_auth2:
- receive `ProtocolReply` -> *stasis*
- receive `AuthenticationNextPhase` -> *give_auth1*
- receive `Error` give up on this protocol -> *stasis*
- receive `WantToClose` -> *give_auth2*

proto_reply:
- send `ProtocolReply` -> *stasis*
- send `AuthenticationRequired` -> *take_auth1*
- send `Error` give up on this protocol -> *stasis*

take_auth1:
- receive `AuthenticationReply` -> *take_auth2*
- receive `Error` give up on this protocol -> *stasis*

take_auth2:
- if good auth data \(-> *take_auth3*
- if bad auth data, send `Error` give up on this protocol -> *stasis*

take_auth3:
- if no more auth needed, send `ProtocolReply` -> *stasis*
- if good auth data, send `AuthenticationNextPhase` -> *take_auth1*
- if bad auth data, send `Error` give up on this protocol -> *stasis*

ping_wait:
- receive `PingReply` -> *stasis*

quit:

-> close connection

Here are the state transitions for shutting down the connection:

shutdown_attempt:
if want to stay alive anyway, send `NoClose` -> *stasis*
else -> *quit*

close_wait:
receive `ProtocolSetup` -> *proto_reply*
receive `NoClose` -> *stasis*
receive `WantToClose` -> *quit*
connection close -> *quit*

Chapter 7. Protocol Encoding

In the encodings below, the first column is the number of bytes occupied. The second column is either the type (if the value is variable) or the actual value. The third column is the description of the value (e.g., the parameter name). Receivers must ignore bytes that are designated as unused or pad bytes.

This document describes major version 1, minor version 0 of the ICE protocol.

LISTof<type> indicates some number of repetitions of <type>, with no additional padding. The number of repetitions must be specified elsewhere in the message.

Primitives

Type Name	**Length (bytes)**	**Description**
CARD8	1	8-bit unsigned integer
CARD16	2	16-bit unsigned integer
CARD32	4	32-bit unsigned integer
LPCE	1	A character from the X Portable Character Set in Latin Portable Character Encoding

Enumerations

Type Name	**Value**	**Description**
BOOL	0	False
	1	True

Compound Types

Type Name	**Length (bytes)**	**Type**	**Description**
VERSION			
	2	CARD16	Major version number
	2	CARD16	Minor version number
STRING			
	2	CARD16	length of string in bytes
	n	LISTofLPCE	string
	p		unused, p = pad(n+2, 4)

ICE Minor opcodes

Message Name	**Encoding**
Error	0
ByteOrder	1
ConnectionSetup	2
AuthenticationRequired	3
AuthenticationReply	4
AuthenticationNextPhase	5
ConnectionReply	6
ProtocolSetup	7
ProtocolReply	8
Ping	9
PingReply	10
WantToClose	11
NoClose	12

Message Encoding

```
Error
     1     CARD8         major-opcode
     1     0             Error
     2     CARD16        class
     4     (n+p)/8+1     length
     1     CARD8         offending-minor-opcode
     1                   severity:
           0               CanContinue
           1               FatalToProtocol
           2               FatalToConnection
     2                   unused
     4     CARD32        sequence number of erroneous message
     n     <varies>     value(s)
     p                   pad, p = pad(n,8)
```

```
ByteOrder
     1     0     ICE
     1     1     ByteOrder
     1           byte-order:
           0        LSBfirst
           1        MSBfirst
     1           unused
     4     0     length
```

```
ConnectionSetup
```

```
     1     0                   ICE
     1     2                   ConnectionSetup
     1     CARD8               Number of versions offered
     1     CARD8               Number of authentication protocol names offered
     4     (i+j+k+m+p)/8+1     length
     1     BOOL                must-authenticate
     7                         unused
     i     STRING              vendor
     j     STRING              release
     k     LISTofSTRING        authentication-protocol-names
     m     LISTofVERSION       version-list
     p                         unused, p = pad(i+j+k+m,8)


AuthenticationRequired
     1     0             ICE
     1     3             AuthenticationRequired
     1     CARD8         authentication-protocol-index
     1                   unused
     4     (n+p)/8+1     length
     2     n             length of authentication data
     6          unused
     n     <varies>     data
     p                   unused, p = pad(n,8)


AuthenticationReply
     1     0             ICE
     1     4             AuthenticationReply
     2                   unused
     4     (n+p)/8+1     length
     2     n             length of authentication data
     6                   unused
     n     <varies>     data
     p                   unused, p = pad(n,8)


AuthenticationNextPhase
     1     0             ICE
     1     5             AuthenticationNextPhase
     2                   unused
     4     (n+p)/8+1     length
     2     n             length of authentication data
     6                   unused
     n     <varies>     data
     p                   unused, p = pad(n,8)


ConnectionReply
     1     0             ICE
     1     6             ConnectionReply
     1     CARD8         version-index
     1                   unused
     4     (i+j+p)/8     length
     i     STRING        vendor
```

```
     j     STRING        release
     p                   unused, p = pad(i+j,8)


ProtocolSetup
     1     0                      ICE
     1     7                      ProtocolSetup
     1     CARD8                  major-opcode
     1     BOOL                   must-authenticate
     4     (i+j+k+m+n+p)/8+1      length
     1     CARD8                  Number of versions offered
     1     CARD8                  Number of authentication protocol names offered
     6                            unused
     i     STRING                 protocol-name
     j     STRING                 vendor
     k     STRING                 release
     m     LISTofSTRING           authentication-protocol-names
     n     LISTofVERSION          version-list
     p                            unused, p = pad(i+j+k+m+n,8)


ProtocolReply
     1     0             ICE
     1     8             ProtocolReply
     1     CARD8         version-index
     1     CARD8         major-opcode
     4     (i+j+p)/8     length
     i     STRING        vendor
     j     STRING        release
     p                   unused, p = pad(i+j, 8)


Ping
     1     0     ICE
     1     9     Ping
     2     0     unused
     4     0     length


PingReply
     1     0     ICE
     1     10    PingReply
     2     0     unused
     4     0     length


WantToClose
     1     0     ICE
     1     11    WantToClose
     2     0     unused
     4     0     length


NoClose
     1     0     ICE
```

```
1     12    NoClose
2     0     unused
4     0     length
```

Error Class Encoding

Generic errors have classes in the range 0x8000-0xFFFF, and subprotocol-specific errors are in the range 0x0000-0x7FFF.

Generic Error Class Encoding

Class	**Encoding**
BadMinor	0x8000
BadState	0x8001
BadLength	0x8002
BadValue	0x8003

ICE-specific Error Class Encoding

Class	**Encoding**
BadMajor	0
NoAuthentication	1
NoVersion	2
SetupFailed	3
AuthenticationRejected	4
AuthenticationFailed	5
ProtocolDuplicate	6
MajorOpcodeDuplicate	7
UnknownProtocol	8

Appendix A. Modification History

Release 6 to Release 6.1

Release 6.1 added the ICE X rendezvous protocol (Appendix B) and updated the document version to 1.1.

Release 6.1 to Release 6.3

Release 6.3 added the listen on well known ports feature.

Appendix B. ICE X Rendezvous Protocol

Introduction

The ICE X rendezvous protocol is designed to answer the need posed in Section 2 for one mechanism by which two clients interested in communicating via ICE are able to exchange the necessary information. This protocol is appropriate for any two ICE clients who also have X connections to the same X server.

Overview of ICE X Rendezvous

The ICE X Rendezvous Mechanism requires clients willing to act as ICE originating parties to pre-register the ICE subprotocols they support in an ICE_PROTOCOLS property on their top-level window. Clients willing to act as ICE answering parties then send an ICE_PROTOCOLS X `ClientMessage` event to the ICE originating parties. This `ClientMessage` event identifies the ICE network IDs of the ICE answering party as well as the ICE subprotocol it wishes to speak. Upon receipt of this message the ICE originating party uses the information to establish an ICE connection with the ICE answering party.

Registering Known Protocols

Clients willing to act as ICE originating parties preregister the ICE subprotocols they support in a list of atoms held by an ICE_PROTOCOLS property on their top-level window. The name of each atom listed in ICE_PROTOCOLS must be of the form ICE_INITIATE_*pname* where *pname* is the name of the ICE subprotocol the ICE originating party is willing to speak, as would be specified in an ICE `ProtocolSetup` message.

Clients with an ICE_INITIATE_*pname* atom in the ICE_PROTOCOLS property on their top-level windows must respond to `ClientMessage` events of type ICE_PROTOCOLS specifying ICE_INITIATE_ *pname*. If a client does not want to respond to these client message events, it should remove the ICE_INITIATE_*pname* atom from its ICE_PROTOCOLS property or remove the ICE_PROTOCOLS property entirely.

Initiating the Rendezvous

To initiate the rendezvous a client acting as an ICE answering party sends an X `ClientMessage` event of type ICE_PROTOCOLS to an ICE originating party. This ICE_PROTOCOLS client message contains the information the ICE originating party needs to identify the ICE subprotocol the two parties will use as well as the ICE network identification string of the ICE answering party.

Before the ICE answering party sends the client message event it must define a text property on one of its windows. This text property contains the ICE answering party's ICE network identification string and will be used by ICE originating parties to determine the ICE answering party's list of ICE network IDs.

The property name will normally be ICE_NETWORK_IDS, but may be any name of the ICE answering party's choosing. The format for this text property is as follows:

Field	**Value**
type	XA_STRING
format	8
value	comma-separated list of ICE network IDs

Once the ICE answering party has established this text property on one of its windows, it initiates the rendezvous by sending an ICE_PROTOCOLS `ClientMessage` event to an ICE originating party's top-level window. This event has the following format and must only be sent to windows that have pre-registered the ICE subprotocol in an ICE_PROTOCOLS property on their top-level window.

Field	**Value**
message_type	Atom = "ICE_PROTOCOLS"
format	32
data.l[0]	Atom identifying the ICE subprotocol to speak
data.l[1]	Timestamp
data.l[2]	ICE answering party's window ID with ICE network IDs text property
data.l[3]	Atom naming text property containing the ICE answering party's ICE network IDs
data.l[4]	Reserved. Must be 0.

The name of the atom in data.l[0] must be of the form ICE_INITIATE_*pname*, where *pname* is the name of the ICE subprotocol the ICE answering party wishes to speak.

When an ICE originating party receives a `ClientMessage` event of type ICE_PROTOCOLS specifying ICE_INITIATE_*pname* it can initiate an ICE connection with the ICE answering party. To open this connection the client retrieves the ICE answering party's ICE network IDs from the window specified in data.l[2] using the text property specified in data.l[3].

If the connection attempt fails for any reason, the client must respond to the client message event by sending a return `ClientMessage` event to the window specified in data.l[2]. This return event has the following format:

Field	**Value**
message_type	Atom = "ICE_INITIATE_FAILED"
format	32
data.l[0]	Atom identifying the ICE subprotocol requested
data.l[1]	Timestamp

Field	Value
data.l[2]	Initiating party's window ID (holding ICE_PROTOCOLS)
data.l[3]	int: reason for failure
data.l[4]	Reserved, must be 0

The values of data.l[0] and data.l[1] are copied directly from the client message event the client received.

The value in data.l[2] is the id of the window to which the ICE_PROTOCOLS.ICE_INITIATE_*pname* client message event was sent.

Data.l[3] has one of the following values:

Value	Encoding	Description
OpenFailed	1	The client was unable to open the connection (e.g. a call to IceOpenConnection() failed). If the client is able to distinguish authentication or authorization errors from general errors, then the preferred reply is `AuthenticationFailed` for authorization errors.
AuthenticationFailed	2	Authentication or authorization of the connection or protocol setup was refused. This reply will be given only if the client is able to distinguish it from `OpenFailed` otherwise `OpenFailed` will be returned.
SetupFailed	3	The client was unable to initiate the specified protocol on the connection (e.g. a call to IceProtocolSetup() failed).
UnknownProtocol	4	The client does not recognize the requested protocol. (This represents a semantic error on the part of the answering party.)
Refused	5	The client was in the process of removing ICE_INITIATE_*pname* from its ICE_PROTOCOLS list when the client message was sent; the client

Value	Encoding	Description
		no longer is willing to establish the specified ICE communication.

Note

Clients willing to act as ICE originating parties must update the ICE_PROTOCOLS property on their top-level windows to include the ICE_INITIATE_*pname* atom(s) identifying the ICE subprotocols they speak. The method a client uses to update the ICE_PROTOCOLS property to include ICE_INITIATE_*pname* atoms is implementation dependent, but the client must ensure the integrity of the list to prevent the accidental omission of any atoms previously in the list.

When setting up the ICE network IDs text property on one of its windows, the ICE answering party can determine its comma-separated list of ICE network IDs by calling IceComposeNetworkIdList() after making a call to IceListenForConnections(). The method an ICE answering party uses to find the top-level windows of clients willing to act as ICE originating parties is dependent upon the nature of the answering party. Some may wish to use the approach of requiring the user to click on a client's window. Others wishing to find existing clients without requiring user interaction might use something similar to the XQueryTree() method used by several freely-available applications. In order for the ICE answering party to become automatically aware of new clients willing to originate ICE connections, the ICE answering party might register for SubstructureNotify events on the root window of the display. When it receives a SubstructureNotify event, the ICE answering party can check to see if it was the result of the creation of a new client top-level window with an ICE_PROTOCOLS property.

In any case, before attempting to use this ICE X Rendezvous Mechanism ICE answering parties wishing to speak ICE subprotocol *pname* should check for the ICE_INITIATE_*pname* atom in the ICE_PROTOCOLS property on a client's top-level window. A client that does not include an ICE_INITIATE_*pname* atom in a ICE_PROTOCOLS property on some top-level window should be assumed to ignore `ClientMessage` events of type ICE_PROTOCOLS specifying ICE_INITIATE_*pname* for ICE subprotocol *pname*.

ICE Subprotocol Versioning

Although the version of the ICE subprotocol could be passed in the client message event, ICE provides more a flexible version negotiation mechanism than will fit within a single `ClientMessage` event. Because of this, ICE subprotocol versioning is handled within the ICE protocol setup phase.

Note

Clients wish to communicate with each other via an ICE subprotocol known as "RAP V1.0". In RAP terminology one party, the "agent", communicates with other RAP-enabled applications on demand. The user may direct the agent to establish communication with a specific application by clicking on the application's window, or the agent may watch for new application windows to be created and automatically establish communication.

During startup the ICE answering party (the agent) first calls IceRegisterForProtocolReply() with a list of the versions (i.e., 1.0) of RAP the agent can speak. The answering party then calls IceListenForConnections() followed by IceComposeNetworkIdList() and stores the resulting ICE network IDs string in a text property on one of its windows.

When the answering party (agent) finds a client with which it wishes to speak, it checks to see if the ICE_INITIATE_RAP atom is in the ICE_PROTOCOLS property on the client's top-level window. If it is present the agent sends the client's top-level window an ICE_PROTOCOLS client message event as described above. When the client receives the client message event and is willing to originate an ICE connection using RAP, it performs an IceRegisterForProtocolSetup() with a list of the versions of RAP the client can speak. The client then retrieves the agent's ICE network ID from the property and window specified by the agent in the client message event and calls IceOpenConnection(). After this call succeeds the client calls IceProtocolSetup() specifying the RAP protocol. During this process, ICE calls the RAP protocol routines that handle the version negotiation.

Note that it is not necessary for purposes of this rendezvous that the client application call any ICElib functions prior to receipt of the client message event.