X Session Management Protocol

X Consortium Standard

Mike Wexler

X Session Management Protocol: X Consortium Standard

by Mike Wexler

X Version 11, Release 7

Version 1.0

Copyright © 1992, 1993, 1994, 2002 The Open Group

Abstract

This document specifies a protocol that facilitates the management of groups of client applications by a session manager. The session manager can cause clients to save their state, to shut down, and to be restarted into a previously saved state. This protocol is layered on top of the X.Org ICE protocol.

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 The Open Group.

Table of Contents

Chapter 1. Acknowledgments

First I would like to thank the entire ICCCM and Intrinsics working groups for the comments and suggestions. I would like to make special thanks to the following people (in alphabetical order), Jordan Brown, Ellis Cohen, Donna Converse, Vania Joloboff, Stuart Marks, Ralph Mor and Bob Scheifler.

Chapter 2. Definitions and Goals

The purpose of the X Session Management Protocol (XSMP) is to provide a uniform mechanism for users to save and restore their sessions. A *session* is a group of clients, each of which has a particular state. The session is controlled by a network service called the *session manager*. The session manager issues commands to its clients on behalf of the user. These commands may cause clients to save their state or to terminate. It is expected that the client will save its state in such a way that the client can be restarted at a later time and resume its operation as if it had never been terminated. A client's state might include information about the file currently being edited, the current position of the insertion point within the file, or the start of an uncommitted transaction. The means by which clients are restarted is unspecified by this protocol.

For purposes of this protocol, a *client* of the session manager is defined as a connection to the session manager. A client is typically, though not necessarily, a process running an application program connected to an X Window System display. However, a client may be connected to more than one X display or not be connected to any X displays at all.

This protocol is layered on top of the X Consortium's ICE protocol and relies on the ICE protocol to handle connection management and authentication.

Chapter 3. Overview of the Protocol

Clients use XSMP to register themselves with the session manager (SM). When a client starts up, it should connect to the SM. The client should remain connected for as long as it runs. A client may resign from the session by issuing the proper protocol messages before disconnecting. Termination of the connection without notice will be taken as an indication that the client died unexpectedly.

Clients are expected to save their state in such a way as to allow multiple instantiations of themselves to be managed independently. A unique value called a *client-ID* is provided by the protocol for the purpose of disambiguating multiple instantiations of clients. Clients may use this ID, for example, as part of a filename in which to store the state for a particular instantiation. The client-ID should be saved as part of the command used to restart this client (the RestartCommand) so that the client will retain the same ID after it is restarted. Certain small pieces of state might also be stored in the RestartCommand. For example, an X11 client might place a '-twoWindow' option in its RestartCommand to indicate that it should start up in two window mode when it is restarted.

The client finds the network address of the SM in a system-dependent way. On POSIX systems an environment variable called `SESSION_MANAGER` will contain a list of network IDs. Each id will contain the transport name followed by a slash and the (transport-specific) address. A TCP/IP address would look like this:

`tcp/`*`hostname`*`:`*`portnumber`*

where the hostname is a fully qualified domain name. A Unix Domain address looks like this:

`local/`*`hostname`*`:`*`path`*

A DECnet address would look like this:

`decnet/`*`nodename`*`::`*`objname`*

If multiple network IDs are specified, they should be separated by commas.

Rationale

There was much discussion over whether the XSMP protocol should use X as the transport protocol or whether it should use its own independent transport. It was decided that it would use an independent protocol for several reasons. First, the Session Manager should be able to manage programs that do not maintain an X connection. Second, the X protocol is not appropriate to use as a general-purpose transport protocol. Third, a session might span multiple displays.

The protocol is connection based, because there is no other way for the SM to determine reliably when clients terminate.

It should be noted that this protocol introduces another single point of failure into the system. Although it is possible for clients to continue running after the SM has exited, this will probably not be the case in normal practice. Normally the program that starts the SM will consider the session to be terminated when the SM exits (either normally or abnormally).

> To get around this would require some sort of rendezvous server that would also introduce a single point of failure. In the absence of a generally available rendezvous server, XSMP is kept simple in the hopes of making simple reliable SMs.

Some clients may wish to manage the programs they start. For example, a mail program could start a text editor for editing the text of a mail message. A client that does this is a session manager itself; it should supply the clients it starts with the appropriate connection information (i.e., the `SESSION_MANAGER` environment variable) that specifies a connection to itself instead of to the top level session manager.

Each client has associated with it a list of properties. A property set by one client is not visible to any other client. These properties are used for the client to inform the SM of the client's current state. When a client initially connects to the SM, there are no properties set.

Chapter 4. Data Types

XSMP messages contain several types of data. Both the SM and the client always send messages in their native byte order. Thus, both sides may need to byte-swap the messages received. The need to do byte-swapping is determined at run-time by the ICE protocol.

If an invalid value is specified for a field of any of the enumerated types, a BadValue error message must be sent by the receiver of the message to the sender of the message.

Type Name	**Description**
BOOL	`False` or `True`
INTERACT_STYLE	`None` `Errors` or `Any`
DIALOG_TYPE	`Error` or `Normal`
SAVE_TYPE	`Global` `Local` or `Both`
CARD8	a one-byte unsigned integer
CARD16	a two-byte unsigned integer
CARD32	a four-byte unsigned integer
ARRAY8	a sequence of CARD8s
LISTofARRAY8	a sequence of ARRAY8s
PROPERTY	a property name (an ARRAY8), a type name, and a value of that type
LISTofPROPERTY	a counted collection of PROPERTYs.

Chapter 5. Protocol Setup and Message Format

To start the XSMP protocol, the client sends the server an ICE `ProtocolSetup` message. All XSMP messages are in the standard ICE message format. The message's major opcode is assigned to XSMP by ICE at run-time. The different parties (client and SM) may be assigned different major opcodes for XSMP. Once assigned, all XSMP messages issued by this party will use the same major opcode. The message's minor opcode specifies which protocol message this message contains.

Chapter 6. Client Identification String

A client ID is a string of XPCS characters encoded in ISO Latin 1 (ISO 8859-1). No null characters are allowed in this string. The client ID string is used in the `RegisterClient` and `RegisterClientReply` messages.

Client IDs consist of the pieces described below. The ID is formed by concatenating the pieces in sequence, without separator characters. All pieces are padded on the left with '`0`' characters so as to fill the specified length. Decimal numbers are encoded using the characters '`0`' through '`9`', and hexadecimal numbers using the characters '`0`' through '`9`' and '`A`' through '`F`'.

- Version. This is currently the character '`1`'.
- Address type and address. The address type will be one of

'`1`'	a 4-byte IPv4 address encoded as 8 hexadecimal digits
'`2`'	a 6-byte DECNET address encoded as 12 hexadecimal digits
'`6`'	a 16-byte IPv6 address encoded as 32 hexadecimal digits

  The address is the one of the network addresses of the machine where the session manager (not the client) is running. For example, the IP address 198.112.45.11 would be encoded as the string "`QC6702D0B`".
- Time stamp. A 13-digit decimal number specifying the number of milliseconds since 00:00:00 UTC, January 1, 1970.
- Process-ID type and process-ID. The process-ID type will be one of

  '`1`' a POSIX process-ID encoded as a 10-digit decimal number.

  The process-ID is the process-ID of the session manager, not of a client.
- Sequence number. This is a four-digit decimal number. It is incremented every time the session manager creates an ID. After reaching "Q9999" it wraps to "Q0000".

Rationale

Once a client ID has been assigned to the client, the client keeps this ID indefinitely. If the client is terminated and restarted, it will be reassigned the same ID. It is desirable to be able to pass client IDs around from machine to machine, from user to user, and from session manager to session manager, while retaining the identity of the client. This, combined with the indefinite persistence of client IDs, means that client IDs need to be globally unique. The construction specified above will ensure that any client ID created by any user, session manager, and machine will be different from any other.

Chapter 7. Protocol

The protocol consists of a sequence of messages as described below. Each message type is specified by an ICE minor opcode. A given message type is sent either from a client to the session manager or from the session manager to a client; the appropriate direction is listed with each message's description. For each message type, the set of valid responses and possible error messages are listed. The ICE severity is given in parentheses following each error class.

`RegisterClient [Client → SM]`

`previous-ID`: `ARRAY8`

`Valid Responses: RegisterClientReply`

`Possible Errors: BadValue (CanContinue)`

The client must send this message to the SM to register the client's existence. If a client is being restarted from a previous session, the *`previous-ID`* field must contain the client ID from the previous session. For new clients, *`previous-ID`* should be of zero length.

If *`previous-ID`* is not valid, the SM will send a BadValue error message to the client. At this point the SM reverts to the register state and waits for another `RegisterClient` The client should then send a `RegisterClient` with a null *`previous-ID`* field.

`RegisterClientReply [Client ← SM]`

`client-ID`: `ARRAY8`

The *`client-ID`* specifies a unique identification for this client. If the client had specified an ID in the *`previous-ID`* field of the `RegisterClient` message, *`client-ID`* will be identical to the previously specified ID. If *`previous-ID`* was null, *`client-ID`* will be a unique ID freshly generated by the SM. The *`client-ID`* format is specified in section 6.

If the client didn't supply a *`previous-ID`* field to the `RegisterClient` message, the SM must send a `SaveYourself` message with type = Local, shutdown = False, interact-style = None, and fast = False immediately after the `RegisterClientReply` The client should respond to this like any other `SaveYourself` message.

`SaveYourself [Client ← SM]`

`type`: `SAVE_TYPE`
`shutdown`: `BOOL`
`interact-style`: `INTERACT_STYLE`
`fast`: `BOOL`

```
Valid Responses:
  SetProperties
  DeleteProperties
  GetProperties
  SaveYourselfDone
  SaveYourselfPhase2Request
  InteractRequest
```

The SM sends this message to a client to ask it to save its state. The client writes a state file, if necessary, and, if necessary, uses `SetProperties` to inform the SM of how to restart it and how to discard the saved state. During this process it can, if allowed by *`interact-style`*, request permission to interact with the user by sending an `InteractRequest` message. After the state has been saved, or if it cannot be successfully saved, and the properties are appropriately set, the client sends a `SaveYourselfDone` message. If the client wants to save additional information after all the other clients have finished changing their own state, the client should send `SaveYourselfPhase2Request` instead of `SaveYourselfDone` The client must then freeze interaction with the user and wait until it receives a `SaveComplete` `Die` or a `ShutdownCancelled` message.

If *`interact-style`* is `None` the client must not interact with the user while saving state. If the *`interact-style`* is `Errors` the client may interact with the user only if an error condition arises. If *`interact-style`* is `Any` then the client may interact with the user for any purpose. This is done by sending an `InteractRequest` message. The SM will send an `Interact` message to each client that sent an `InteractRequest` The client must postpone all interaction until it gets the `Interact` message. When the client is done interacting it should send the SM an `InteractDone` message. The `InteractRequest` message can be sent any time after a `SaveYourself` and before a `SaveYourselfDone`

Unusual circumstances may dictate multiple interactions. The client may initiate as many `InteractRequest` - `Interact` - `InteractDone` sequences as it needs before it sends `SaveYourselfDone`

When a client receives `SaveYourself` and has not yet responded `SaveYourselfDone` to a previous `SaveYourself` it must send a `SaveYourselfDone` and may then begin responding as appropriate to the newly received `SaveYourself`

The *`type`* field specifies the type of information that should be saved: `Global` `Local` or `Both` The `Local` type indicates that the application must update the properties to reflect its current state, send a `SaveYourselfDone` and continue. Specifically it should save enough information to restore the state as seen by the user of this client. It should not affect the state as seen by other users. The `Global` type indicates that the user wants the client to commit all of its data to permanent, globally-accessible storage. `Both` indicates that the client should do both of these. If `Both` is specified, the client should first commit the data to permanent storage before updating its SM properties.

Examples

If a word processor was sent a `SaveYourself` with a type of `Local` it could create a temporary file that included the current contents of the file, the location of the cursor, and other aspects of the current editing session. It would then update its `RestartCommand` property with enough information to find the temporary file, and its `DiscardCommand` with enough information to remove it.

If a word processor was sent a `SaveYourself` with a type of `Global` it would simply save the currently edited file.

If a word processor was sent a `SaveYourself` with a type of `Both` it would first save the currently edited file. It would then create a temporary file with information such as the current position of the cursor and what file is being edited. It would then update its `RestartCommand` property with enough information to find the temporary file, and its `DiscardCommand` with enough information to remove it.

Once the SM has send `SaveYourself` to a client, it can't send another `SaveYourself` to that client until the client either responds with a `SaveYourselfDone` or the SM sends a `ShutdownCancelled`

Advice to Implementors

If the client stores local any state in a file or similar "external" storage, it must create a distinct copy in response to each `SaveYourself` message. It *must not* simply refer to a previous copy, because the SM may discard that previous saved state using a `DiscardCommand` without knowing that it is needed for the new checkpoint.

The *`shutdown`* field specifies whether the system is being shut down.

Rationale

The interaction may be different depending on whether or not shutdown is set.

The client must save and then must prevent interaction until it receives a `SaveComplete` `Die` or a `ShutdownCancelled` because anything the user does after the save will be lost.

The *`fast`* field specifies whether or not the client should save its state as quickly as possible. For example, if the SM knows that power is about to fail, it should set the *`fast`* field to `True`.

```
SaveYourselfPhase2 [Client → SM]

  Valid Responses:
    SetProperties
    DeleteProperties
    GetProperties
    SaveYourselfDone
    InteractRequest
```

The SM sends this message to a client that has previously sent a `SaveYourselfPhase2Request` message. This message informs the client that all other clients are in a fixed state and this client can save state that is associated with other clients.

Rationale

> Clients that manager other clients (window managers, workspace managers, etc) need to know when all clients they are managing are idle, so that the manager can save state related to each of the clients without being concerned with that state changing.

The client writes a state file, if necessary, and, if necessary, uses `SetProperties` to inform the SM of how to restart it and how to discard the saved state. During this process it can request permission to interact with the user by sending an `InteractRequest` message. This should only be done if an error occurs that requires user interaction to resolve. After the state has been saved, or if it cannot be successfully saved, and the properties are appropriately set, the client sends a `SaveYourselfDone` message.

```
SaveYourselfRequest [Client → SM]

  type: SAVE_TYPE
  shutdown: BOOL
  interact-style: INTERACT_STYLE
  fast: BOOL
  global: BOOL

  Valid Responses: SaveYourself
```

An application sends this to the SM to request a checkpoint. When the SM receives this request it may generate a `SaveYourself` message in response and it may leave the fields intact.

Example

> A vendor of a UPS (Uninterruptible Power Supply) might include an SM client that would monitor the status of the UPS and generate a fast shutdown if the power is about to be lost.

If *global* is set to `True` then the resulting `SaveYourself` should be sent to all applications. If *global* is set to `False` then the resulting `SaveYourself` should be sent to the application that sent the `SaveYourselfRequest`

```
InteractRequest [Client → SM]

  dialog-type: DIALOG_TYPE

  Valid Responses: Interact ShutdownCancelled

  Possible Errors: BadState (CanContinue)
```

During a checkpoint or session-save operation, only one client at a time might be granted the privilege of interacting with the user. The `InteractRequest` message causes the SM to emit an `Interact` message at some later time if the shutdown is not cancelled by another client first.

The *dialog-type* field specifies either `Errors` indicating that the client wants to start an error dialog or `Normal` meaning the client wishes to start a non-error dialog.

```
Interact [Client ← SM]

  Valid Responses: InteractDone
```

This message grants the client the privilege of interacting with the user. When the client is done interacting with the user it must send an `InteractDone` message to the SM unless a shutdown cancel is received.

Advice to Implementors

If a client receives a ShutdownCancelled after receiving an `Interact` message, but before sending a `InteractDone` the client should abort the interaction and send a `SaveYourselfDone`

```
InteractDone [Client → SM]

  cancel-shutdown: BOOL

  Valid Responses: ShutdownCancelled
```

This message is used by a client to notify the SM that it is done interacting.

Setting the *cancel-shutdown* field to `True` indicates that the user has requested that the entire shutdown be cancelled. *Cancel-shutdown* may only be `True` if the corresponding `SaveYourself` message specified `True` for the shutdown field and `Any` or `Errors` for the *interact-style* field. Otherwise, *cancel-shutdown* must be `False`.

```
SaveYourselfDone [Client → SM]

  success: BOOL

  Valid Responses:
    SaveComplete
    Die
    ShutdownCancelled
```

This message is sent by a client to indicate that all of the properties representing its state have been updated. After sending `SaveYourselfDone` the client must wait for a `SaveComplete` `ShutdownCancelled` or `Die` message before changing its state. If the `SaveYourself` operation was successful, then the client should set the *success* field to `True` otherwise the client should set it to `False`.

Example

If a client tries to save its state and runs out of disk space, it should return `False` in the *success* field of the `SaveYourselfDone` message.

```
SaveYourselfPhase2Request [Client → SM]

  Valid Responses:
    ShutdownCancelled
    SaveYourselfPhase2
```

This message is sent by a client to indicate that it needs to be informed when all the other clients are quiescent, so it can continue its state.

```
Die [Client ← SM]

  Valid Responses: ConnectionClosed
```

When the SM wants a client to die it sends a `Die` message. Before the client dies it responds by sending a `ConnectionClosed` message and may then close its connection to the SM at any time.

```
SaveComplete [Client → SM]

  Valid Responses:
```

When the SM is done with a checkpoint, it will send each of the clients a `SaveComplete` message. The client is then free to change its state.

```
ShutdownCancelled [Client ← SM]
```

The shutdown currently in process has been aborted. The client can now continue as if the shutdown had never happened. If the client has not sent `SaveYourselfDone` yet, the client can either abort the save and send `SaveYourselfDone` with the success field set to `False` or it can continue with the save and send a `SaveYourselfDone` with the success field set to reflect the outcome of the save.

```
ConnectionClosed [Client → SM]

  reason: LISTofARRAY8
```

Specifies that the client has decided to terminate. It should be immediately followed by closing the connection.

The *reason* field specifies why the client is resigning from the session. It is encoded as an array of Compound Text strings. If the resignation is expected by the user, there will typically be zero ARRAY8s here. But if the client encountered an unexpected fatal error, the error message (which might otherwise be printed on stderr on a POSIX system) should be forwarded to the SM here, one ARRAY8 per line of the message. It is the responsibility of the SM to display this reason to the user.

After sending this message, the client must not send any additional XSMP messages to the SM.

Advice to Implementors

If additional messages are received, they should be discarded.

Rationale

The reason for sending the `ConnectionClosed` message before actually closing the connections is that some transport protocols will not provide immediate notification of connection closure.

`SetProperties [Client → SM]`

`properties``: LISTofPROPERTY`

Sets the specified *`properties`* to the specified values. Existing properties not specified in the `SetProperties` message are unaffected. Some properties have predefined semantics. See section 11, “Predefined Properties.”

The protocol specification recommends that property names used for properties not defined by the standard should begin with an underscore. To prevent conflicts among organizations, additional prefixes should be chosen (for example, _KPC_FAST_SAVE_OPTION). The organizational prefixes should be registered with the X Registry. The XSMP reserves all property names not beginning with an underscore for future use.

`DeleteProperties [Client → SM]`

`property-names``: LISTofARRAY8`

Removes the named properties.

`GetProperties [Client → SM]`

`Valid Responses: GetPropertiesReply`

Requests that the SM respond with the values of all the properties for this client.

`GetPropertiesReply [Client ← SM]`

`values``: LISTofPROPERTY`

This message is sent in reply to a `GetProperties` message and includes the *`values`* of all the properties.

Chapter 8. Errors

When the receiver of a message detects an error condition, the receiver sends an ICE error message to the sender. There are only two types of errors that are used by the XSMP: BadValue and BadState These are both defined in the ICE protocol.

Any message received out-of-sequence will generate a BadState error message.

Chapter 9. State Diagrams

These state diagrams are designed to cover all actions of both the client and the SM.

Client State Diagram

start:
  ICE protocol setup complete → *register*

register:
  send *RegisterClient* → *collect-id*

collect-id:
  receive *RegisterClientReply* → *idle*

shutdown-cancelled:
  send *SaveYourselfDone* → *idle*

idle: [Undoes any freeze of interaction with user.]
  receive *Die* → *die*
  receive *SaveYourself* → *freeze-interaction*
  send *GetProperties* → *idle*
  receive *GetPropertiesReply* → *idle*
  send *SetProperties* → *idle*
  send *DeleteProperties* → *idle*
  send *ConnectionClosed* → *connection-closed*
  send *SaveYourselfRequest* → *idle*

die:
  send *ConnectionClosed* → *connection-closed*

freeze-interaction:
  freeze interaction with user → *save-yourself*

save-yourself:
  receive *ShutdownCancelled* → *shutdown-cancelled*
  send *SetProperties* → *save-yourself*
  send *DeleteProperties* → *save-yourself*
  send *GetProperties* → *save-yourself*
  receive *GetPropertiesReply* → *save-yourself*
  send *InteractRequest* → *interact-request*
  send *SaveYourselfPhase2Request* → waiting-for-phase2

save-yourself:
  if shutdown mode:

```
        send SaveYourselfDone → save-yourself-done
    otherwise:
        send SaveYourselfDone → idle


waiting-for-phase2:
    receive ShutdownCancelled → shutdown-cancelled
    receive SaveYourselfPhase2 → phase2


phase2:
    receive ShutdownCancelled → shutdown-cancelled
    send SetProperties → save-yourself
    send DeleteProperties → save-yourself
    send GetProperties → save-yourself
    receive GetPropertiesReply → save-yourself
    send InteractRequest → interact-request (errors only)
    if shutdown mode:
        send SaveYourselfDone → save-yourself-done
    otherwise:
        send SaveYourselfDone → idle


interact-request:
    receive Interact → interact
    receive ShutdownCancelled → shutdown-cancelled


interact:
    send InteractDone → save-yourself
    receive ShutdownCancelled → shutdown-cancelled


save-yourself-done: (changing state is forbidden)
    receive SaveComplete → idle
    receive Die → die
    receive ShutdownCancelled → idle


connection-closed:
    client stops participating in session
```

Session Manager State Diagram

```
start:
    receive ProtocolSetup → protocol-setup


protocol-setup:
    send ProtocolSetupReply → register


register:
```

```
    receive RegisterClient → acknowledge-register

acknowledge-register:
    send RegisterClientReply → idle

idle:
    receive SetProperties → idle
    receive DeleteProperties → idle
    receive ConnectionClosed → start
    receive GetProperties → get-properties
    receive SaveYourselfRequest → save-yourself
    send SaveYourself → saving-yourself

save-yourself:
    send SaveYourself → saving-yourself

get-properties:
    send GetPropertiesReply → idle

saving-get-properties:
    send GetPropertiesReply → saving-yourself

saving-yourself:
    receive InteractRequest → saving-yourself
    send Interact → saving-yourself
    send ShutdownCancelled → idle
    receive InteractDone → saving-yourself
    receive SetProperties → saving-yourself
     receive DeleteProperties → saving-yourself
    receive GetProperties → saving-get-properties
    receive SaveYourselfPhase2Request → start-phase2
    receive SaveYourselfDone → save-yourself-done

start-phase2:
    If all clients have sent either SaveYourselfPhase2Request or SaveYourselfDone:
        send SaveYourselfPhase2 → phase2
    else
        → saving-yourself

phase2:
    receive InteractRequest → saving-yourself
    send Interact → saving-yourself
    send ShutdownCancelled → idle
    receive InteractDone → saving-yourself
    receive SetProperties → saving-yourself
     receive DeleteProperties → saving-yourself
    receive GetProperties → saving-get-properties
```

```
    receive SaveYourselfDone → save-yourself-done


save-yourself-done:
    If all clients are saved:
        If shutting down:
            send Die → die
        otherwise
            send SaveComplete → idle

    If some clients are not saved:
    → saving-yourself


die:
    SM stops accepting connections
```

Chapter 10. Protocol Encoding

Types

BOOL			
	0	False	
	1	True	
INTERACT_STYLE			
	0	None	
	1	Errors	
	2	Any	
DIALOG_TYPE			
	0	Error	
	1	Normal	
SAVE_TYPE			
	0	Global	
	1	Local	
	2	Both	
ARRAY8			
	4	CARD32	length
	n	ListofCARD8, the array	p = pad (4 + n, 8)
	2	Both	
LISTofARRAY8			
	4	CARD32	count
	4		unused
	a	ARRAY8	first array
	b	ARRAY8	second array
	.		
	.		
	.		
	q	ARRAY8	last array
PROPERTY			
	a	ARRAY8	name
	b	ARRAY8	type (XPCS encoded in Latin-1, case sensitive)

	c	LISTofARRAY8	values

LISTofPROPERTY

	4	CARD32	count
	4		unused
	a	PROPERTY	first property
	b	PROPERTY	second property
	.		
	.		
	.		
	q	PROPERTY	last property

Messages

XSMP is a sub-protocol of ICE. The major opcode is assigned at run-time by ICE and is represented here by '?'.

To start the XSMP protocol, the client sends the server an ICE `ProtocolSetup` message. The protocol-name field should be specified as "`XSMP`", the major version of the protocol is 1, the minor version is 0. These values may change if the protocol is revised. The minor version number will be incremented if the change is compatible, otherwise the major version number will be incremented.

In `ProtocolReply` message sent by the session manager, the XSMP protocol defines the vendor parameter as product identification of the session manager, and defines the release parameter as the software release identification of the session manager. The session manager should supply this information in the ICE `ProtocolReply` message.

`RegisterClient`

	1	?	XSMP
	1	1	opcode
	2		unused
	4	a/8	length of remaining data in 8-byte units
	a	ARRAY8	previous-ID

`RegisterClientReply`

	1	?	XSMP
	1	2	opcode
	2		unused
	4	a/8	length of remaining data in 8-byte units

	a	ARRAY8	client-ID
SaveYourself			
	1	?	XSMP
	1	3	opcode
	2		unused
	4	1	length of remaining data in 8-byte units
	1	SAVE_TYPE	type
	1	BOOL	shutdown
	1	INTERACT_STYLE	interact-style
	1	BOOL	fast
	4		unused
SaveYourselfRequest			
	1	?	XSMP
	1	4	opcode
	2		unused
	4	1	length of remainning data in 8-byte units
	1	SAVE_TYPE	type
	1	BOOL	shutdown
	1	INTERACT_STYLE	interact-style
	1	BOOL	fast
	3		unused
InteractRequest			
	1	?	XSMP
	1	5	opcode
	1	DIALOG_TYPE	dialog type
	1		unused
	4	0	length of remaining data in 8-byte units
Interact			
	1	?	XSMP
	1	6	opcode
	2		unused
	4	0	length of remaining da-

			ta in 8-byte units
`InteractDone`			
	1	?	XSMP
	1	7	opcode
	1	BOOL	cancel-shutdown
	1		unused
`InteractDone`			
	1	?	XSMP
	1	7	opcode
	1	BOOL	cancel-shutdown
	1		unused
	4	0	length of remaining data in 8-byte units
`SaveYourselfDone`			
	1	?	XSMP
	1	8	opcode
	1	BOOL	success
	1		unused
	4	0	length of remaining data in 8-byte units
`Die`			
	1	?	XSMP
	1	9	opcode
	1		unused
	4	0	length of remaining data in 8-byte units
`ShutdownCancelled`			
	1	?	XSMP
	1	10	opcode
	2		unused
	4	0	length of remaining data in 8-byte units

`ConnectionClosed`

	1	?	XSMP
	1	11	opcode
	2		unused
	4	a/8	length of remaining data in 8-byte units
	a	LISTofARRAY8	reason

`SetProperties`

	1	?	XSMP
	1	12	opcode
	2		unused
	4	a/8	length of remaining data in 8-byte units
	a	LISTofPROPERTY	properties

`DeleteProperties`

	1	?	XSMP
	1	13	opcode
	2		unused
	4	a/8	length of remaining data in 8-byte units
	a	LISTofPROPERTY	properties

`GetProperties`

	1	?	XSMP
	1	14	opcode
	2		unused
	4	0	length of remaining data in 8-byte units

`GetPropertiesReply`

	1	?	XSMP
	1	15	opcode
	2		unused

	4	a/8	length of remaining data in 8-byte units
	a	LISTofPROPERTY	properties
SaveYourselfPhase2Request			
	1	?	XSMP
	1	16	opcode
	2		unused
	4	0	length of remaining data in 8-byte units
SaveYourselfPhase2			
	1	?	XSMP
	1	17	opcode
	2		unused
	4	0	length of remaining data in 8-byte units
SaveComplete			
	1	?	XSMP
	1	18	opcode
	2		unused
	4	0	length of remaining data in 8-byte units

Chapter 11. Predefined Properties

All property values are stored in a LISTofARRAY8. If the type of the property is CARD8, the value is stored as a LISTofARRAY8 with one ARRAY8 that is one byte long. That single byte contains the CARD8. If the type of the property is ARRAY8, the value is stored in the first element of a single element LISTofARRAY8.

The required properties must be set each time a client connects with the SM. The properties must be set after the client sends `RegisterClient` and before the client sends `SaveYourselfDone` Otherwise, the behavior of the session manager is not defined.

Clients may set, get, and delete nonstandard properties. The lifetime of stored properties does not extend into subsequent sessions.

Name	**Type**	**Posix Type**	**Required?**
CloneCommand	OS-specific	LISTofARRAY8	Yes
CurrentDirectory	OS-specific	ARRAY8	No
DiscardCommand	OS-specific	LISTofARRAY8	No*
Environment	OS-specific	LISTofARRAY8	No
ProcessID	OS-specific	ARRAY8	No
Program	OS-specific	ARRAY8	Yes
RestartCommand	OS-specific	LISTofARRAY8	Yes
ResignCommand	OS-specific	LISTofARRAY8	No
RestartStyleHint	CARD8	CARD8	No
ShutdownCommand	OS-specific	LISTofARRAY8	No
UserID	ARRAY8	ARRAY8	Yes

* Required if any state is stored in an external repository (e.g., state file).

CloneCommand	This is like the `RestartCommand` except it restarts a copy of the application. The only difference is that the application doesn't supply its client id at register time. On POSIX systems the type should be a LISTofARRAY8.
CurrentDirectory	On POSIX-based systems specifies the value of the current directory that needs to be set up prior to starting the program and should be of type ARRAY8.
DiscardCommand	The discard command contains a command that when delivered to the host that the client is running on (determined from the connection), will cause it to discard any information about the current state. If this command is not specified, the SM will assume that all of the client's state is encoded in the `RestartCommand` On POSIX systems the type should be LISTofARRAY8.
Environment	On POSIX based systems, this will be of type LISTofARRAY8 where the ARRAY8s alternate between environment variable name and environment variable value.

ProcessID	This specifies an OS-specific identifier for the process. On POSIX systems this should of type ARRAY8 and contain the return value of getpid() turned into a Latin-1 (decimal) string.
Program	The name of the program that is running. On POSIX systems this should be the first parameter passed to execve and should be of type ARRAY8.
RestartCommand	The restart command contains a command that when delivered to the host that the client is running on (determined from the connection), will cause the client to restart in its current state. On POSIX-based systems this is of type LISTofARRAY8 and each of the elements in the array represents an element in the argv array. This restart command should ensure that the client restarts with the specified client-ID.
ResignCommand	A client that sets the `RestartStyleHint` to `RestartAnyway` uses this property to specify a command that undoes the effect of the client and removes any saved state.

Example

A user runs xmodmap. xmodmap registers with the SM, sets `RestartStyleHint` to `RestartAnyway` and then terminates. In order to allow the SM (at the user's request) to undo this, xmodmap would register a `ResignCommand` that undoes the effects of the xmodmap.

RestartStyleHint

If the RestartStyleHint property is present, it will contain the style of restarting the client prefers. If this flag isn't specified, `RestartIfRunning` is assumed. The possible values are as follows:

Name	Value
RestartIfRunning	0
RestartAnyway	1
RestartImmediately	2
RestartNever	3

The `RestartIfRunning` style is used in the usual case. The client should be restarted in the next session if it is connected to the session manager at the end of the current session.

The `RestartAnyway` style is used to tell the SM that the application should be restarted in the next session even if it exits before the current session is terminated. It should be noted that this is only a hint and the SM will follow the policies specified by its users in determining what applications to restart.

Rationale

This can be specified by a client which supports (as MS-Windows clients do) a means for the user to indicate while exiting that restarting is desired. It can also be used for clients that spawn other clients and then go away, but which want to be restarted.

A client that uses `RestartAnyway` should also set the `ResignCommand` and `ShutdownCommand` properties to commands that undo the state of the client after it exits.

The `RestartImmediately` style is like `RestartAnyway` but in addition, the client is meant to run continuously. If the client exits, the SM should try to restart it in the current session.

Advice to Implementors

It would be wise to sanity-check the frequency which which `RestartImmediately` clients are restarted, to avoid a sick client being restarted continuously.

The `RestartNever` style specifies that the client does not wish to be restarted in the next session.

Advice to Implementors

This should be used rarely, if at all. It will cause the client to be silently left out of sessions when they are restarted and will probably be confusing to users.

ShutdownCommand

This command is executed at shutdown time to clean up after a client that is no longer running but retained its state by setting `RestartStyleHint` to `RestartAnyway` The command must not remove any saved state as the client is still part of the session.

Example

A client is run at start up time that turns on a camera. This client then exits. At session shutdown, the user wants the camera turned off. This client would set the `RestartStyleHint` to `RestartAnyway` and would register a `ShutdownCommand` that would turn off the camera.

UserID

Specifies the user's ID. On POSIX-based systems this will contain the the user's name (the *pw_name* field of struct passwd).