Application Interoperability 6

Introduction

One of the strengths of a multitasking operating system coupled with an advanced windowing system, such as Solaris 2.x, is that it presents end users with a desktop metaphor on their screens, which may have many applications running simultaneously. Often these applications are from different vendors and have no knowledge of each other. However, users rightly expect that the applications work together to solve their computational problems. Solaris 2.x provides a suite of technologies that allows seamless cooperation and interoperability between applications. These technologies are collectively termed *desktop integration*. An application that adopts the desktop integration technologies will be well positioned to be a good citizen on the desktop and will provide greater functionality at lower cost than a non-integrated application.

The DeskSet™ suite of applications is part of OpenWindows and uses desktop integration technologies. Thus, it behooves the application developer to interoperate with at least DeskSet so that users are presented with a consistent environment. Application users should be able to drag and drop their application-specific data file from File Manager onto the corresponding application and expect that the application will load the file. They should be able to double-click on an icon representing an application-specific data file in an attachment to Mail Tool and launch the application with the data file loaded. They should be able to drag and drop a data file (or part of a data file) from an application onto Print Tool and have it printed, with filtering by the application if necessary.

Additionally, the desktop windowing environment is controlled by the window manager, such as the OPEN LOOK® Window Manager, and an application must understand the protocols used to cooperate with the window manager for such things as:

- application minimum and maximum window size and resize increments
- application window decorations, especially for transient windows
- strings displayed in the window header
- preferred icon name, size, location and glyph to be used
- termination notices received on its windows
- notices on window state changes received

What is Application Interoperability?

Application interoperability, in this chapter, refers to what you must provide in your applications to take full advantage of desktop integration technologies. It also encompasses how you should prepare your applications for the next generation of distributed, object-oriented facilities, which are inherently application and desktop spanning.

Application interoperability ranges from the simple to the very complex and is best envisioned through realistic user scenarios. The following descriptions give a feel for what applications should be able to do with current technology and what the future holds.

Current Application Interoperability Scenarios

Some scenarios, going from the simple to the complex, include the following:

- You can select on-screen text data, for example, in a spreadsheet, cut or copy it to a potentially invisible clipboard, and then paste that data into another application, such as a text editor.
- Your actions to achieve the preceding, such as clicking to select, copy or paste either through a menu or a function key, and then clicking to paste, can be simplified by using a drag and drop gesture. You select the data, “drag” it by holding down the left mouse button and moving the mouse to the other application, receiving visual feedback in the process, and “drop” it by releasing on the left mouse button for inclusion in its new location. The visual feedback is the responsibility of the application and is given by changing the pointer to a “drag-in-progress” cursor during the drag operation.
- Applications can communicate about data formats so that more than simple text can be acted upon (for example, raster images), or the text may retain attributes such as font type, size, color, and boldface or underlining information.
- You can drag a data object on the desktop to Print Tool to be printed in the appropriate format. The application managing the data object communicates with Print Tool with respect to the format and contents of the object.
- You can attach an icon representing a desktop publishing file to a mail message. The message recipient can then access the desktop publishing file simply by double-clicking on the icon, which launches the desktop publishing application and loads the file.
- Groupware applications can be developed which allow several people to work simultaneously on a document or program while the system automatically performs housekeeping chores, such as updating files automatically and informing group members of file changes, thereby ensuring data consistency.

These scenarios and the window manager interactions can be handled using current Solaris desktop integration technologies:

- ICCCM for window manager interaction – ICCCM, or the *Inter-Client Communication Conventions Manual*, is a document owned by the X Consortium; it specifies the protocols used for inter-client and client-to-window manager communication.
- Selections and drag and drop for interapplication data exchange – these provide a mechanism for user-directed flow of information from one application to another.
- Classing Engine for application registration with DeskSet tools – this provides methods for applications to determine the identity and operating characteristics of objects on the desktop.
- The ToolTalk service for object- and process-oriented messaging – this mechanism passes messages and commands between applications and is largely context-free.
- Project DOE for distributed, object-oriented application building – this is a key set of technologies being developed at SunSoft for building the next generation of interoperating applications. The ToolTalk service provides a migration path to Project DOE. Use of Project DOE technologies is described in the next section and also in Chapter 16, "Project DOE: Future Solaris Strategies".

This order represents the technology used as an application goes from a minimum to a maximum level of interoperation and inter-cooperation.

The following figure shows the relationships between these components.

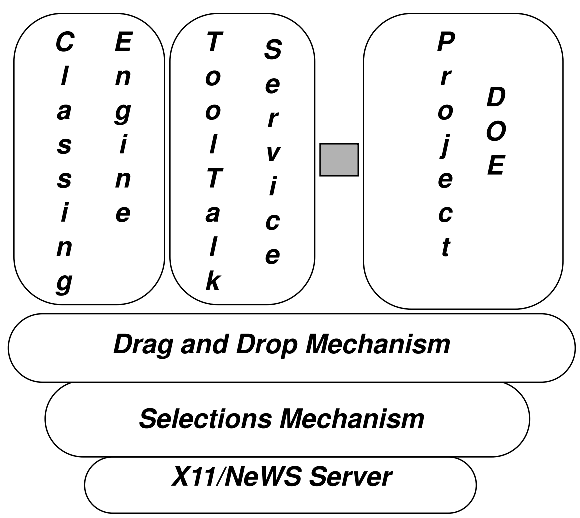

Figure 6-1 Components of Application Interoperability

Future Scenarios for Application Interoperability

Looking into the future, SunSoft has embarked upon developing a new set of technologies to enable cooperating, distributed, object-oriented applications to share and exchange data transparently. This set is collectively termed Project DOE (Distributed Objects Everywhere). DOE-enabled applications will allow scenarios such as the following:

- You can establish links between data objects in two or more applications. When data in the originating application changes, the link is refreshed and updated in the linked-to applications. The data is stored and managed by the linked-from application.
- You can embed objects from one application inside another application. You are presented with a single “compound” document, and the applications that work with pieces of the document are automatically invoked as various objects are referenced. For example, a spreadsheet could embed a graph object inside a presentation table. The data to generate the graph would come from ranges in the spreadsheet and the graphing application would be invoked when the graph is selected by the user. The embedded graph object is stored along with the other pieces of the compound document, but the spreadsheet has no knowledge of data format and may have the graphing application assist it in storing the data.
- You can work with collections of objects, some supplied by SunSoft, others supplied by outside vendors. The object handlers only understand their own objects and depend on well-defined interfaces to communicate with other object handlers. Communication is completely transparent to the end user, who does not have to be concerned about details such as file systems, database formats, or concurrency control.

For example, you might want to create an application that will project the impact on air pollution if a recently discovered gasoline additive is introduced in the market. Such an application could go through the following chain of events:

1. Communicate with various database handlers to gather statistics on current air pollution levels, age, efficiency and type of automobiles affected by the new technology. Potentially, an on-line journal server could get the technical details that describe changes in emissions due to the new additive.
2. Communicate with a spreadsheet application to package the gathered statistics and technical details into tabular form.
3. Communicate with a statistical modeling package to run various mathematical models on the gathered statistics and various marketing strategies in the introduction of the additive and feed these results back into the spreadsheet.
4. Communicate with a graphing or presentation package to summarize the modeling results and embed the graphs into the spreadsheet.
5. Communicate with a word processor to incorporate the results into a report.

6. Communicate with a multimedia application to insert live-video or audio and a drawing application to enhance the report.

An application such as the one being discussed, built using current technology, will be large, complex, monolithic and probably of limited functionality. Project DOE lets you work on your area of expertise knowing your application will plug-and-play in the Solaris environment.

Current SunSoft desktop integration technologies, in particular the ToolTalk service, provide a migration path to Project DOE. Thus, adoption of ToolTalk now is important for developers who wish to be in position to participate in Project DOE and also protect their development investment. For more information on Project DOE, see the chapter titled "Project DOE: Future Solaris Technologies."

ICCCM For Window Manager Interaction

Applications designed to run under OpenWindows share window server resources with other applications. The mediator for competing demands is the window manager, which is another client of the server, but one with special properties. The most basic function that application clients are expected to do is to set certain properties which provide the window manager with the information on which to base its decisions. However, applications should not depend on any particular window manager for their proper operation. For this reason, the X Consortium has endorsed a standard set of properties that all adherent window managers and applications conform to and which allows them to communicate effectively. The full set is described, albeit somewhat cryptically, in the X Consortium's *Inter-Client Communications Conventions Manual* (ICCCM). This manual is reprinted as Appendix L in the *X Protocol Reference Manual*.

Applications using OLIT can have all essential properties set by the call to `OlToolkitInitialize()`, which must be used to initialize the toolkit prior to creation of a top-level shell widget. They can also use a number of convenience shell resources to set optional window manager hints. These are listed in Section 10.2 of the *X Toolkit Intrinsics Programming Manual*. The basic set of optional resources includes features such as:

- width and height of fixed components and desired resize increments
- icon pixmap or window, icon name, and icon mask
- whether resizing is allowed
- initial state (iconic or normal)
- aspect ratios
- keyboard input model
- string displayed in the top-level window header

OLIT also allows registration of callback routines that are invoked when one of the window manager protocol messages is received by the application. For example, the QUIT button on the menu placed by OLWM on the top-level window of a client application will deliver a `WM_PROTOCOL` message, with the value of `OL_WM_DELETE_WINDOW`, when selected.

To handle this, you can use the following:

```
/* Add quit_callback as part of application initialization */
OlAddCallback(toplevel, XtNwmProtocol, wm_protocol_callback, NULL);
......
void wm_protocol_callback( Widget w, XtPointer client_data, XtPointer
call_data) {
Display *dpy = XtDisplay(w);
OlWMProtocolVerify *olwmpv = (OlWMProtocolVerify *)call_data;
if (olwmpv->msgtype == OL_WM_DELETE_WINDOW) {
/* save any files, cleanup, etc.. */
...
}
exit(0);
}
```

Other values that may be returned to the callback function set up for the `XtNwmProtocol` resource are `OL_WM_TAKE_FOCUS` and `OL_WM_SAVE_YOURSELF`. These could also be handled in `wm_protocol_callback()`.

Further details on application integration with the window manager may be found in the following places:

- Section 10.2 of the *Xlib Programming Manual.* This section explains the Window and Session manager protocols that applications need to be aware of. It does not deal with toolkit-based applications, but describes the ICCCM and some window manager protocols.
- Section 10.1 of the *X Toolkit Intrinsics Programming Manual*. This describes how the Intrinsics layer provides an easy API for ICCCM protocols.
- Chapter 11 of *The X Window System Programming & Applications with Xt OPEN LOOK Edition*. This describes window properties and how to use them effectively.
- Section 3.7 on "Notes on X11 Programming" of the *OpenWindows Version 3 Programmer's Guide*. See especially the paragraph on the `WM_DELETE_WINDOW` client message.
- The description of the `XtNwmProtocol` resource in resources man page in the *OLIT 3.0.1 Reference Manual*.
- The `protocol` man page in the *OLIT 3.0.1 Reference Manual* that describes the `OlWMProtocolAction()` function, which may be used to simulate a response to any window manager protocol messages.

Selections and Drag and Drop

Selections are a mechanism for moving data between or within on-screen applications using the X- server as the rendezvous point. Drag and drop is an implementation of selections that allow users to select a data object with the mouse, drag it across the screen, and drop it onto another application for usage.

Selections Mechanism

The term *selection* refers both to the way a user highlights an object, such as a block of text, a file icon, or a window, for moving or copying, and to the object itself once the highlighting is done. To further overload the term, selection also may refer to the "well-known" X-server token representing the selected object. These tokens are called *atoms* in X terminology.

The manner in which a user gestures to select an object may differ between applications, but generally involve the SELECT and ADJUST mouse buttons and the CUT, COPY, and PASTE command keys for acting upon text blocks. User interface conventions for data object selection are outlined in the *OPEN LOOK Functional Specification*.

Selections provide a communications link between an *owner* client and a *requestor* client. All data transfer and communication between owner and requestor goes through the X-server. Thus the owner client informs the server that it owns a particular selection (in response to a user gesture, such as highlighting text with the mouse and pressing the COPY key) and the requestor client informs the server that it wants a particular selection (in response to the user pressing the PASTE key). The X-server mediates the message and data exchange, because the clients do not have to be running on the same machine (or even on the same machine type), making more direct interprocess communication methods, such as shared memory or semaphores, impossible.

Applications will play both owner and requestor client roles at different points during their lifetimes. Selections are generally limited to simple and similar data objects such as a contiguous block of text, a pixmap, or a file icon.

Selection Owner

The following are the responsibilities of a selection owner:

- Define and detect the end-user gesture to select an object; for example, using a mouse-driven wipe-through to select text.
- Provide visual feedback to the user when an object has been selected, such as displaying selected text in reverse video.

- Acquire ownership of a selection so that the X-server can then direct requests from requestor clients to the owner client. Here selection refers to an atom that the X-server can use as a mediation entity for owner and requestor clients.
- Provide a conversion procedure that is called by the X-server when a selection request is made. The conversion procedure is registered in the call to acquire selection ownership. The conversion procedure must:
  - convert the contents of the selected object to the requested data type
  - place this data in the named property on the named window. The named property and window are supplied by the requestor as part of the request
  - send the requestor an event to let it know that the named property is now available

Selection Requestor

The responsibilities of a selection requestor are to:

- Define and detect the user gesture to request a selection. For example, pressing the PASTE key.
- Request the data by posting a request to get data. The X-server directs the request to the current selection owner. The request must contain:
  - the name of the selection desired
  - the name of the property that the selection data is placed upon
  - the name (*window id*) of the window making the request
  - an atom representing the data type desired

Role of the ICCCM

The ICCCM specifies three predefined selection atoms that may be used for acquiring ownership and for requesting data:

- `PRIMARY`—used for exchanging text (but not text attributes, such as font, size, color, etc.)
- `SECONDARY`—alternative selection for exchanging text (especially when `PRIMARY` should not be disturbed)
- `CLIPBOARD`—non-textual data exchange

The ICCCM also specifies many predefined data type atoms that requestors may specify and which owner clients should support. In addition, a mechanism for incremental transfer of large data objects is also specified (the INCR model).

Toolkit Considerations

The task of implementing the selections mechanism is considerably simplified when using an X toolkit such as OLIT or XView. This discussion will focus on using OLIT, followed by a short list of the differences when using XView.

Most of the selections mechanism is implemented in the intrinsics layer of OLIT. There are convenience functions to acquire ownership and request a selection.

- `XtOwnSelection()`
  Asserts ownership of a named selection, such as `PRIMARY`, and registers the conversion procedure. Also allows registration of a `lose ownership` callback procedure and a `transfer done` callback procedure. The first is invoked when the widget loses the selection because another widget asserts ownership; the latter is called when data transfer completes.
- `XtGetSelectionValue()`
  Requests the value of a named selection, such as `PRIMARY`, and registers a `requestor` callback procedure that is invoked when the selection owner reports a successful conversion. The `requestor` callback retrieves the data placed on the property passed in the `XtGetSelectionValue()` call and pastes it into the application.
- There are also calls to:
  - get selection data incrementally: `XtGetSelectionValueIncremental()`
  - get or set a timeout within which widgets must respond to each other: `XtSetSelectionTimeout()`, `XtAppSetSelectionTimeout()`, `XtGetSelectionTimeout()`, and `XtAppGetSelectionTimeout()`
  - get data in more than one representation: `XtGetSelectionValues()`

Text manipulation between OLIT text-oriented widgets is handled transparently. Applications do not have to have any code to handle copy and paste between two `TextEdit` widgets or between a `TextField` and a `TextEdit` widget.

Differences Using XView

Generally, OLIT and XView are similar in that they use atoms as selection owner and requestor common mediation entities. Being somewhat more object-oriented, XView requires the application to create a `SELECTION_OWNER` object as part of its initialization or when someone using the application initiates a selection action. XView uses the term *selection rank* to refer to the predefined selection atoms. Rank 1 refers to `PRIMARY`, rank 2 refers to `SECONDARY`, and rank 3 refers to `CLIPBOARD`.

CLIPBOARD holds data when the CUT or COPY keys are pressed; PRIMARY is used to hold the data when the selection is made using the SELECT or ADJUST mouse keys or the mouse-driven wipe-through. When the COPY key is hit, the selection rank is owned by the SELECTION_OWNER object. Pressing the COPY key delivers an ACTION_COPY event to the XView application. Upon receipt of this event, it is the responsibility of the application to:

- own rank 1 (PRIMARY)
- own CLIPBOARD and copy data to it

The requesting application can ask for a selection in either blocking or non-blocking mode. OLIT, however, is inherently non-blocking. Non-blocking mode is achieved by registering a reply proc when the request is initiated. The reply proc will be invoked when the selection owner returns from its conversion proc. See the *XView Programming Manual* for more details.

XView has a built-in, default conversion proc, `sel_convert_proc()`, that handles entities called selection items and should be called by the application-defined conversion proc before its return.

The DeskSet OpenWindows Shell Tool and Command Tool use the CLIPBOARD selection when text is copied using the COPY key.

Drag and Drop

The importance of the drag and drop mechanism is that it allows applications created by developers to interoperate with the DeskSet suite of applications and with other applications implementing drag and drop.

Overview

Drag and drop is an implementation of selections that allows users to select a data object, such as a text block, graphic, audio object, or file icon with the mouse, drag it across the screen, and drop it into another application. For example, a text file icon can be selected, dragged to the Print Tool, and dropped, where it is printed. Another example involves dropping a spreadsheet data file onto a spreadsheet application icon where it will be loaded and displayed on the screen. Note that drag and drop differs from the CUT, COPY, and PASTE command keys in that it is not limited to moving only text blocks, but can move complete data objects.

Applications that implement drag and drop can exchange data with other applications. Drag and drop, like selections, has a different API for the XView and OLIT toolkits. Once implemented in a client, however, drag and drop works invisibly and seamlessly between both toolkits.

Drag and Drop User Interface

To implement drag and drop, you must understand the drag and drop user interface. This section briefly describes this interface for purposes of terminology.

Overview

Drag and drop allows users to transfer data objects with the mouse, between or within applications. A drag and drop action consists of a *source* (object to be transferred) and a *destination* (the place where the source will be dropped). Before an object can be dragged or dropped, it must be *selected*. Two types of objects that can be selected: a *data span* or *glyph*. A data span is a segment of on-screen data. It can be a segment of text, digitized audio, video, and so forth. A *glyph* is an on-screen representation of some object, such as a file, application, or directory.

A data span can be selected in three ways: the wipe method, the select-adjust method, and the multi-click method. With the wipe method you place the pointer at the beginning of the data span, press the SELECT button, drag the mouse to the end of the selection, and release the mouse. In the select-adjust method, you place the pointer at the beginning of the selection and click the SELECT button to select the starting point. Then you move the pointer to the end of the desired span and click the ADJUST button on the mouse to make the selection. With the multi-click method you rapidly press the SELECT button to select increasingly larger segments of the segment. For example, two rapid clicks selects a word, three a line, and four a paragraph. A selected data span is displayed in reverse video.

Glyphs are selected by clicking the SELECT button on the glyph. To select additional glyphs, click ADJUST on additional glyphs.

Initiating the Drag

Drag and drop can be initiated as either a *cut-and-paste* or a *copy-and-paste* operation. In a cut-and-paste operation, the original object is deleted after it is dropped. In a copy-and-paste operation, the original object remains after the object is dropped on a destination—the original object is not deleted. The default for the Solaris desktop is cut and paste. To initiate a drag, the user puts the pointer on the selection, presses and holds the SELECT button, and drags the pointer to the location of the new drop location. To copy the selection, the user presses the Control key while starting the drag.

Visual Feedback

When an object is dragged, the pointer changes appearance and a representation of the object follows the pointer as the mouse is moved. In addition, when the pointer is dragged over potential drop sites, the pointer changes to indicate the receptivity of that drop site.

The Drop

The final action in the drag and drop gesture is to drag the selection over the destination object and release the SELECT mouse button. The destination is determined by the position of the pointer's hot spot at the time the user releases the SELECT button.

Applications supporting drops other than a simple cut or copy sometimes require a specific drop site, referred to as a *drag and drop target*. A drag and drop target is a graphical element located in the control area of an open window. In addition to serving as the destination in drag and drop operations, drag and drop targets sometimes contain a glyph which can be used as the source in a drag and drop operation.

A complete specification of the drag and drop UI is found in Appendix A, "Drag and Drop User Interface Specification" of the *OpenWindows Version 3.1 Desktop Integration Guide*.

Implementing Drag and Drop in Your Application

The following sections summarize the toolkit-independent processes required for sourcing drags and receiving drops.

Sourcing a Drag

To adapt an application to source a drag, the following steps are required:

1. Define a drag and drop object and associate a drag pointer with it. The window manager will use the drag pointer to provide visual feedback to the user when the object is selected.
2. Associate a selection with your drag and drop object that will contain the data you want to make available to the target.
3. Provide an event callback procedure for your drag and drop object that will detect when it has been dragged. Set the actual data in the data object for the source to retrieve and wait for a source response or error condition.

Depending on the application, you may also want to perform the following:

1. Define a selection conversion procedure for your own data types.
2. Provide the data through an Alternate Transport Mechanism (ATM), such as sockets or the ToolTalk service.

The handshake protocol for a drag occurrence is shown in the following code sample:

```
/* Set default values that the owner app. provides to a requestor that doesn't
make any specific requests */
set item = default item       /* Generally set to the first item */
set type = default_type       /* Set the data type you would like to send the
                              /* default item in */
set transport = X_SERVER      /* Default transport method is the X-Server*/
set length = length of 1st item  /* Length of default item */
/* make available the types you support for the default item */
set avail_types = types for item or types you support (array of types)
/* set up the Convert proc for the drag-and-drop process */
In a registered callback, get ready to provide information:
if filename                   /* if they ask for a filename */
    set filename              /* fill in the filename atom info */
    return true               /* return the information */
if data label                 /* if they ask for the data label */
    set data label            /* fill in the data label atom info */
    return true               /* return the information */
/* Advertise the targets supported by the owner app. (i.e., this is the list
of atoms returned when the TARGETS is converted */
if targets
    set targets
    create array of:
        _SUN_ENUMERATION_COUNT/ITEM
        _SUN_AVAILABLE_TYPES
        LENGTH = length
    return true
if _SUN_ENUMERATION_ITEM      /* only if handling multiple selections and
                              enumeration count > 1 is the case */
set item = _SUN_ENUMERATION_ITEM
    set length = length of item
set name = name of item       /* or data label if needed */
    specify _SUN_AVAILABLE_TYPES for that item

/* Additional information can be set or simply returned as false */
```

Receiving a Drop

To adapt your application to receive drops, perform the following steps:

1. Define a drop site and associate an event procedure with it.
2. (Optional) Provide an image for drop site previewing which will provide visual feedback when the pointer is over the drop site.
3. In your event callback procedure, determine the event type and obtain data from the source selection.
4. Depending on the application, you may also need to do the following:
   – Provide drop site feedback when pointer enters and leaves the drop site.
   – Use an alternate transport method such as sockets or the ToolTalk service to transfer data if your application design requires it.

Handshake protocol for a drop occurrence is shown in the following code sample:

```
get enumeration_count     /* gets number of selection objects */
if no count
  count = 1
  if count > 1 and application can only handle one
      end dragdrop
      return False
  else
      get ready to handle multiple selections
/* obtain drop information from owner application. This applies both when
handling single or multiple selections*/
For each item in count:
  /* Note: a special atom corresponds to all the information below. So using
  selections, request the information by supplying the pre-defined atom */
Specify item being requested   /* specify item number */
Request host name              /* Get owners host name */
Request filename               /* Request pathname of item */
Request data label             /* Data Label is an identifier for the */
      /* item (usually the name for items items that don’t */
      /* have real files). For example, an attachment in */
      /* mailtool may have a data label without a filename */
      /* attached to it */
Request available types /* List of data types in which the
      /* owner may send the data */
      /* Now that we have information on data, let source know that we are ready
      toload the data by converting the load atom*/
```

```
 Request Load
specify item, and data type/* Specify item and data type in which the
                               requestor would like the data */
 if delete then
     delete              /* If the action is a move, send owner
     done
```

Good references with complete code samples are:

- Section 11.5 of the *X Window System Programming and Applications with Xt: OPEN LOOK Edition*. This book uses the OLIT API.
- The *OLIT 3.0.1 Widget Set Reference Manual*. This describes the complete OLIT API to drag and drop.
- Chapters 3 and 4 of the *OpenWindows Version 3.1 Desktop Integration Guide*. These chapters describe interoperating with DeskSet applications with complete code samples. Appendix E provides an OLIT-based example program.
- The `olwm` man page. This describes `dsdm`, the drop site data manager which is built into the OPEN LOOK Window Manager.

Data Type Registration for Drag and Drop

It is very important that you support common data types to ensure interoperability and that you publish specialized data types so that other developers' applications can interoperate with your applications.

If a receiving application is to receive a drop from a source application, the source application must send the data in a format or type readable by the receiving application. (In this discussion the terms data format and data type are used interchangeably.) For example, if Text Editor wishes to drop data into Mail Tool, Text Editor must be able to convert the data to a format which Mail Tool can read. Conversely, if Mail Tool wishes to drop data into Text Editor, Mail Tool must be able to convert the data to a format Text Editor can read.

Although the source application is responsible for converting data to a format readable by the receiving application, it is also useful if the receiving application can receive data in a commonly used data format such as ASCII, Sun raster image, or PostScript.

As detailed earlier, drag and drop handshaking works as follows:

1. Data is selected from the source application.
2. Data is sent (dropped) on the receiving application.

3. Receiving application requests a list of the data formats in which the source application can send the drop.
4. Source application replies with a list of data formats.
5. Receiving application tells the source application which format it would like the data sent.
6. Data is transferred.

A source application must have data conversion routines for each application to which it wishes to drop data. Creating conversion routines consists of finding out the data format of the desired drop applications and writing conversion routines specifically for those formats. Refer to the receiving application's manuals or call the company that produces the receiving application for details of the data format.

Again, if you wish your application to be able to receive drops from other applications, ensure that your application can receive data in some of the more common data formats.

SunSoft has undertaken a data type registration program to help standardize the data format names by which applications request data formats from each other. SunSoft encourages all companies that wish to share their data with other applications to register data format names for their application's data. This name will be used by other applications to reference desired data formats. Refer to Appendix C, "Vendor Data Type Registration,"of the *OpenWindows Version 3.1 Desktop Integration Guide* for more information on data type registration.

DeskSet Considerations for Drag and Drop

This section discusses what developers must do to enable their applications to work with the DeskSet suite of tools (File Manager, Mail Tool, Print Tool, Text Editor, PageView, and Audio Tool). Since these tools are shipped with OpenWindows, they will be available on all desktops and should be the minimal set of applications that interoperate with any developer's application.

DeskSet applications are built with XView. Thus, an understanding of the XView selections and the drag and drop mechanism is a prerequisite for interoperating with DeskSet. This is not meant to imply that your application must be written using XView; rather that the OLIT API is somewhat different from XView and both must be understood.

The DeskSet drag and drop communications protocol consists of a set of predefined atoms used by owner and requestor applications to exchange information about a drag and drop selection. These predefined atoms are also called *targets*. DeskSet targets describe the nature of a drag and drop selection as well as the interaction that can occur between the owner application and the requestor application.

Targets in a DeskSet drag and drop conversation represent a series of questions or commands sent by the receiving application to the owner application. When a requesting application asks an owner application to convert a target, it is equivalent to asking the owner a predefined question or giving it a predefined command. The selection owner may or may not support answering that particular question or command. If it does not, the requestor must terminate the drag and drop procedure or try to ask other questions that will allow the procedure to continue. If the selection owner does support that target, it will respond with the answer to the requestor.

The DeskSet conventions establish a set of targets (questions) that cooperating selection owners understand. Of course, a well-behaved requestor program will try to deal with selection owners that cannot answer the standard questions, but there will be some loss of functionality when dealing with a selection holder of this type.

Drag and drop between DeskSet applications and other applications has different levels of complexity depending on the features incorporated into the process. In all cases, however, drag and drop handshaking begins with the requestor application asking the owner application to convert the `TARGETS` atom. The owner responds with a list of atoms it understands. If the requestor recognizes enough atoms to continue the procedure, it continues. If the requestor does not recognize enough atoms to complete the process, then the transfer is aborted.

Two cases of handshaking with a DeskSet application follow.

- Handshaking—Simplest Case
  The simplest case of drag and drop is when a receiver application asks the owner to send it selection data without regard to data format, alternative transport method, or any other considerations, as follows:

  a. The requestor asks the owner to convert the `_SUN_ENUMERATION_COUNT` atom. The owner responds by sending the number of source objects in the current selection.
  If the response to `_SUN_ENUMERATION_COUNT` is more than `1`, then the programs must agree upon the selection object to be dragged and dropped. The owner converts the `_SUN_ENUMERATION_ITEM` target which specifies a selection item to process.

  b. For each selection item, the requestor asks the owner to convert the `TEXT` target which instructs the owner to send the selection data in any format. The requestor assumes that it can convert the data into a usable format. If it cannot convert the data, it will not accept the data and the transfer is terminated. Note that the requesting application bears the responsibility of converting the data to a usable format.

Handshaking with _SUN_AVAILABLE_TYPES

The following describes the handshaking protocol of a requesting application that wishes to specify the format of the requested data, an optional feature.

a. After obtaining the list of supported targets the requestor asks the owner to convert the _SUN_ENUMERATION_COUNT atom. The owner responds by sending the number of source objects in the current selection.

   If the response to _SUN_ENUMERATION_COUNT is more than 1, then the programs must agree upon the selection object to be dragged and dropped. The owner converts the _SUN_ENUMERATION_ITEM target which specifies a selection item to process. For each item in the selection, the program converts the targets described in steps a and b under "Handshaking—simplest case" in the section above.

b. The requestor application then asks the owner to convert _SUN_AVAILABLE_TYPES. The owner sends a list of data type atoms which specify the format in which the selection data may be supplied.

   The formats may be GIF, PostScript, audio, or some other format. If the applications do not wish to use an alternate transport method, then the requestor application asks the owner to convert the desired data type target. The owner responds by sending the stream of data representing the object selected in the type requested.

c. The requestor application writes the data into its address space and closes the selection transfer with the _SUN_SELECTION_END atom.

Specifying _SUN_ENUMERATION_ITEM

In the case of recipients that wish to process selections with multiple objects, care must be used in enclosing the setting of the count value and the associated target conversions in MULTIPLE transfers.

When multiple items are selected, the _SUN_ENUMERATION_ITEM target specifies which object is the subject of discussion in the selection negotiation. All targets which require the specification of _SUN_ENUMERATION_ITEM and the setting of the object number as its side effect should request these conversions as part of a MULTIPLE conversion.

The use of MULTIPLE adds a transaction-like nature to the set of target conversions, ensuring that other target conversions delivered to the selection holder will not change the state of the selection.

Specifically, a sample conversion of a target should look like this:

```
begin multiple
_SUN_ENUMERATION_ITEM = object number
FILE_NAME
_SUN_FILE_HOST_NAME
_SUN_ENUMERATION_ITEM = -1
end multiple
```

The use of MULTIPLE ensures that another client cannot convert `_SUN_ENUMERATION_ITEM` on the same selection halfway through the conversions for your targets. If this were to happen, the results could prove unpredictable, with conversion results being relative to different objects within the selection and your client unaware of this.

Within the OpenWindows Version 3.x implementation of drag and drop, each drag and drop transfer acquires a unique selection rank. So, converting `_SUN_ENUMERATION_ITEM` on the same selection halfway through target conversion may not be an immediate problem. But if your client is ever to use the same code to support cut, copy, and paste, it is likely your client could get conversion requests on a selection from more than one client at a time.

For a more complete discussion of MULTIPLE and how it relates to selection target conversions, see the ICCCM.

DeskSet Drag and Drop Target Atoms

The following section contains a list of atoms used by DeskSet applications. The first set is specified by the ICCCM, the second set is specific to DeskSet.

ICCCM Target Atoms

These atoms are specified by the ICCCM and should be supported by all applications:

- `DELETE`
  Converting this atom tells the selection owner to delete the selection. This is typically done when the drag and drop operation is a move command to something that can actually store the data, such as a File Manager. An application such as a Print Tool should always reject the `DELETE` atom. Owner application should return a zero-length property of type NULL if the deletion is successful.

- `TARGETS`
  Returns a list of targets supported by the owner.

- `TIMESTAMP`
  An integer timestamp used to acquire the selection.

- FILE_NAME
  When converted, requests the filename for the current object in the selection. The response is a name of a file that contains the selection data. This filename is not useful unless the answer to the _SUN_FILE_HOST_NAME is a host that the recipient can access. The FILE_NAME response will be relative to the machine named in _SUN_FILE_HOST_NAME. FILE_NAME is not needed if _SUN_ATM_FILENAME is not supported.
- NAME
  The name of the application as represented in the title bar. Refer to the ICCCM for details.
- INSERT_SELECTION
  Replaces the object with the contents of the named selection.

DeskSet Target Atoms

- _SUN_ALTERNATE_TRANSPORT_METHODS
  Like the TARGETS target—it returns a list of atoms which represent the alternate transports that the owner supports. Note that use of this and other ATM targets is very rare and almost always unnecessary.
- _SUN_ATM_FILE_NAME
  Indicates that the requestor is going to get the data via a filename. A filename consists of both a pathname (as returned by FILE_NAME) and a host name (as returned by _SUN_FILE_HOST_NAME).
- _SUN_ATM_TOOL_TALK
  Specifies the ToolTalk services to pass data.
- _SUN_AVAILABLE_TYPES
  Like the TARGETS atom, but instead the owner responds with the data types in which it can provide data. This may be only one or it may be several. Note that the responsibility for type translation lies with the requestor.

Table 6-1 lists the data types supported by DeskSet.

Table 6-1 DeskSet Data Type Atoms

Data Type	Data Type Atom	Type Description
Owner's Choice	TEXT	Can be any data type of the owner's choice.
Text	STRING	8-bit ISO 8859-1
Graphics	_SUN_TYPE_gif-file	GIF
	_SUN_TYPE_tiff-file	TIFF
	_SUN_TYPE_postscript-file	PostScript
	_SUN_TYPE_sun-raster	Sun Raster
	_SUN_TYPE_xpm-file	XPM
Audio	_SUN_TYPE_audio-file	SunSoft Audio Tool

- `_SUN_DRAGDROP_DONE`
  Signifies the end of a drag and drop process. This target is hidden in the toolkits so the requestor application never deals with them. The owner application does, however, see them.
- `_SUN_SELECTION_END`
  Ends the selection process by indicating that the requestor is done with the current selection. Use `SUN_SELECTION_END` to notify the owner that you are done with the selection used in response to `INSERT_SELECTION` or `_SUN_LOAD`. See the section on `_SUN_LOAD` for more information.
- `_SUN_DATA_LABEL`
  A string identifier that represents the current object which is not a filename. This could be something like the last component of the pathname.
- `_SUN_ENUMERATION_COUNT`
  Specifies how many objects are in the current selection. This allows several icons to be selected and separately negotiated.
- `_SUN_ENUMERATION_ITEM`
  Specifies which object is the subject of discussion in the selection negotiation. Objects are numbered from zero to `_SUN_ENUMERATION_COUNT -1`. References to objects outside this range should be refused. References to the item `-1` should be treated as resetting the current item to unspecified.

  This target only has meaning within the context of a selection that contains multiple disjointed objects. It should be:

  1) converted with the number of the object in the selection you wish to refer to;
  2) converted to the desired targets;
  3) converted again to set the current item back to `0`.

- `_SUN_FILE_HOST_NAME`
  The name of the host to which `FILE_NAME` is relative.
- `_SUN_LOAD`
  When converted, the owner returns a selection atom to use as a reference to the current object.
- `_SUN_SELECTION_ERROR`
  If the requestor has decided that it cannot convert a selection, it should convert this target. This target has no side effects and returns no data, but can be used to inform the user that the drag and drop operation did not succeed. The convention for informing the user of a drag and drop failure is to present an error message in the footer of the owner application, and send a popup message to the requestor application.

Classing Engine

The *Classing Engine* (CE) is a Solaris database used to identify the characteristics, or *attributes*, of files. The CE specifies attributes such as print method, desktop icon, and opening commands for specific *file types*, such as ASCII, PostScript, or Sun raster files; FrameMaker® or Lotus 1-2-3® data files; File Manager, Mail Tool, or Wingz® executable files.

The CE consists of two parts: a database that stores file type names and attributes, and a collection of routines (an API) that query the database. Some of the more common file attributes are:

- A filename pattern or content string to identify the file type
- Directory location of a file type icon
- Foreground and background colors of a file type icon
- Print command of a file type, if applicable
- Launch and load command of a data file of a particular application
- Edit or display command of a file

Other attributes, such as data exchange filters, text compression procedures, and ToolTalk attributes can be associated with a file type as well—the CE is completely extensible. In addition, custom databases for other data objects can be added to the CE.

Because the CE acts as a central repository for all file types and their attributes and provides applications with a set of routines for determining a file's type and retrieving its attributes, it may be used for getting the characteristics of a drag and dropped glyph representing a file.

Developers should also register their application-specific file attributes with the CE so that DeskSet tools (File Manager in particular) know how to correctly launch, print, and display (as an icon) the application and its files. The CE file type registration should be part of the application's installation.

File Manager provides an example of how the CE can be used on the desktop. File Manager is a DeskSet application that graphically displays a UNIX file system. You can move, copy and delete files by dragging and dropping file icons onto directory icons, or onto a wastebasket icon. In addition, File Manager allows users to double-click on a data file icon to open the file in its parent application (file opening commands are stored in the CE database). For example, double-clicking on a spreadsheet data file could start the spreadsheet application program and open the data file. Double-clicking on an ASCII file will open the file with the Text Editor. File Manager also lets users print a data file by simply dropping the file's icon onto the Print Tool.

Another feature of the File Manager is representation of different file types by different icons. Thus, one application's files will have one icon, and the files of another application will have a different icon. Unique icons allow users to identify a file without opening it. File Manager retrieves the icon location from the CE.

File Type Registration

For an application to obtain the operating attributes of a file, the file's identifier and attributes must be stored in the CE database. This requires that the file's originators, typically the vendor whose application created the file, incorporate the file type name, its identifier, and its attributes into the CE database. File types can be incorporated into the CE database in the following ways:

- Software vendors may register file types and their attributes with SunSoft through the Developer Integration Format Registration program (DIFR). The new file types will be incorporated into the CE database and distributed in subsequent CE releases.
- Software vendors may use the CE utilities in their software installation process to update their user's CE databases with new file type information. Thus, a vendor's application can, as part of the installation process, enter its file types and attributes into the CE database.
- End users can use command-line CE utilities or Binder, a DeskSet application, to enter new file type information into the CE database.

Classing Engine Database

The CE database contains file type names, identification patterns, and attributes. The CE database is one logical database that is the composite of three physical databases called the *user*, *system*, and *network databases*. Multiple databases allow users to personalize their environment while still having access to global data.

The user database is unique to each end user and resides in the end-user's directory structure; the system portion is common to all users on that specific machine, and the network portion is available to everyone on the network. The CE treats these three portions as overlays. When an application queries the CE database for information, the CE will first read the entry in the user database. If an entry is not found in the user database, the CE tries the system database, and finally the network database. This procedure ensures that any CE database information customized by an end user or the system will be used if it exists; otherwise, network information is used. The following discussion treats the three databases as a single aggregate database.

Default Location of Classing Engine Databases

Each of the three Classing Engine (CE) databases has a default location, as shown in the table below. These files are in a non-readable format. To convert these files into an ASCII-readable format, use the `ce_db_build` utility by typing:

```
ce_db_build <user | system | network> -to_ascii <filename>
```

Table 6-2 Default Classing Engine Database Locations

Database	Default Location
user	`~/.cetables/cetables`
system	`/etc/cetables/cetables`
network	`$OPENWINHOME/lib/cetables/cetables`

Namespace Tables

Each CE database file consists of two *namespace tables* that are databases of file entries:

- A *files namespace table* that contains file type names and identifiers
- A *types namespace table* stores file type attributes

Both of these namespace tables are resident in the same file. A third namespace table for the ToolTalk service is also in this file, but is not included in this discussion. To view the namespace tables, use the `ce_db_build` command described in the previous section.

File Type Identification

Before an application can use a file's attributes, the application must identify, or *derive*, the file type. In other words, it must determine whether a file is an ASCII file, Mail Tool executable file, PostScript file, and so forth. Two methods are used to determine file types: type-by-pattern or type-by-content.

Typing by *pattern* involves matching the filename with a filename pattern. For example, all files whose names end in `.c` are C source files, all files that end in `.exe` are DOS executable files, and all files that end in `.ps` are PostScript files.

Typing by *content* involves matching the contents of a file to a predefined string or number. For example, files that have the string `WNGZWZSS`, `WNGZWZSC`, `or WNGZWZHP` as their first characters are Wingz worksheet files. Files that contain `MakerFile` as their first characters are FrameMaker files. This procedure is similar to that used by the standard UNIX `file` command which uses the `/etc/magic` file.

Types Namespace Table

The types namespace table contains the attribute values of the file types. Once a file type is derived, the CE can retrieve the file's attributes from the types namespace table.

Two examples of arguments in an entry in the type namespace table are `TYPE_ICON` and `TYPE_ICON_MASK`:

- `TYPE_ICON` is the file containing the icon representation of the file type.
- `TYPE_ICON_MASK` is the file containing the icon representation of file when it is selected.

Adding a New File Type

The basic steps for adding a new file type to the CE database are:

1. Create an ASCII description file for the new file entry. Either extract the ASCII description file for the entire CE database by means of `ce_db_build` or create a new ASCII description file for the new file type entry.
2. Add a file type name and file type pattern to the files namespace table in the ASCII description file. You only need to add the file type pattern if the file type is derived by the type-by-pattern method. If the file is derived by the type-by-content method, add a magic match, magic match data type, and an offset.
3. After a new file type has been added to the files namespace table, add its attributes to the types namespace table.
4. Once the attributes are added, you can overwrite the old CE database file with the one you just created using the `ce_db_build` command. Use this command only if you are replacing the entire CE database file. If you have created an ASCII description file for a subset of the entire CE database (this procedure is described in the next section), merge the file into the current CE database with the `ce_db_merge` command.

See Chapter 5 “Classing Engine” in the *OpenWindows Version 3.1 Desktop Integration Guide* for a complete example of a CE database and how to add a new file type to the CE database.

The Classing Engine API

CE database entries may be accessed programmatically through the CE API (`libce.so`). The basic steps are:

1. Open the CE database using `ce_begin(NULL)`.
2. Get namespace identifiers by using `ce_get_namespace_id(char *namespace)`.
3. Retrieve desired attribute ids by using `ce_get_attribute_id(namespace_id, char *attibute)`.
4. Get desired entry in the desired namespace by using retrieved attribute id via `ce_get_entry()` and `ce_get_attribute()`.

See Chapter 5, “Classing Engine,” in the *OpenWindows Version 3.1 Desktop Integration Guide* for a complete example of a program that demonstrates these steps.

The ToolTalk Service

The ToolTalk service is a network-spanning, interapplication communication service that allows your application to communicate with other autonomous applications in a process-oriented or object-oriented manner. The ToolTalk service is used by independent applications to communicate with each other without having direct knowledge of each other. Applications communicate by creating and sending ToolTalk messages. The ToolTalk service receives these messages, determines the recipients, and then delivers the messages to the appropriate applications.

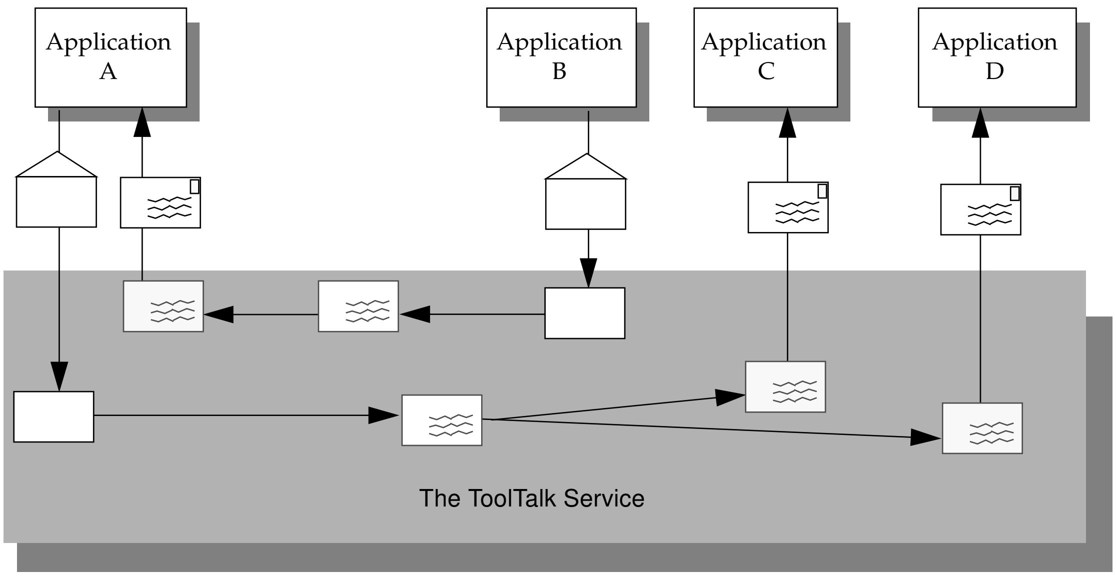

Figure 6-2 Applications Using The ToolTalk Service To Communicate

The ToolTalk service provides *multicast* messaging. That is, an application sends a message that is delivered by the ToolTalk service to multiple receivers. Multicast messaging, with the concept of one-to-many communications, falls between broadcast messaging (one-to-all) and point-to-point messaging (one-to-one). The ToolTalk service also provides point-to-point messaging between applications.

The ToolTalk service supports two types of messaging, *process-oriented* and *object-oriented* messaging. Process-oriented messages are addressed to other processes; object-oriented messages are addressed to objects managed by processes.

Before modifying your application to use the ToolTalk service, you must define (or locate) a *message protocol*, a set of ToolTalk messages that describe operations applications agree to perform. The message protocol specification includes the set of messages and how applications should behave when they receive the messages.

The main differences between the ToolTalk and the Selections, Drag and Drop, and Classing Engine desktop integration technologies are:

- ToolTalk is network spanning. The two applications can exist anywhere on the network.
- ToolTalk can provide multicast messaging. The other technologies are inherently point-to-point.
- ToolTalk does not require any user intervention and has no UI components.

- ToolTalk messaging may be anonymous. Once the message set is established, any application may substitute for another.
- ToolTalk supports object-oriented programming and messaging.
- ToolTalk provides an easy migration path towards a distributed object computing environment. This is discussed in more detail in the next section.

Overview of ToolTalk Messaging

To communicate, applications create, send, and receive ToolTalk messages. Sending applications create, fill in, and send a message; the ToolTalk service determines the recipient and delivers the message to the receiving application. Receiving applications retrieve messages and after examining the information in the message either perform an operation and reply with the results or discard the message.

An important ToolTalk feature is that sending applications need to know little about the receiving application, because applications that want to receive messages register their interest in specific types of messages with the ToolTalk service.

The reason that sending applications need to know little about the receiving application is because applications that want to receive messages explicitly state what these messages should look like. This information is registered with ToolTalk in the form of *message patterns*. These message patterns usually match the message protocols that applications have agreed to use. Applications can add more patterns for individual use.

Message patterns are created much like a message is created; the same type of information is used in both. For each type of message an application wants to receive, it obtains an empty message pattern, fills in the attributes, and registers the pattern with the ToolTalk service. When the ToolTalk service receives a message from a sender, it compares the information in the message to the patterns that are registered. Once matches have been found, the ToolTalk service delivers copies of the message to all recipients.

For each pattern describing a message it wants to receive, an application declares whether it can handle the message or just wants to observe it. Although many applications can observe a message, only one application can handle it to ensure that a requested operation is performed at most once. If the ToolTalk service cannot find a handler for a message, it returns the message to its sender indicating that delivery failed.

Applications can provide message patterns to the ToolTalk service at installation time or while the application is running.

Sending ToolTalk Messages

To send a ToolTalk message, an application obtains an empty message, fills in the message attributes, and sends the message. ToolTalk messages are simple structures containing fields for address, subject, and delivery information. To send a ToolTalk message, an application need only provide this information:

- Should the recipient respond to the message; is this a notice or a request?
- What operation should be performed/has been performed? Are there arguments needed to perform the operation?
- What interest does the recipient share with the sender; is the recipient running in a specific user session or interested in a specific file?

To help narrow the focus of the message delivery, an application can provide more information in the message.

Receiving ToolTalk Messages

When the ToolTalk service determines that a message needs to be delivered to a specific process, it creates a copy of the message and notifies the process that a message is waiting. If a receiver is not running, the ToolTalk service looks for instructions (provided by the application at installation time through registration) on how to start the application. The process retrieves the message and examines its contents. The message may contain a notice that some operation has been performed. In this case, the process discards the message after reading the information. If the message contains a request to perform an operation, the process performs the operation and returns the result of the operation in a reply to the original message. Once the reply has been sent, the process can discard the original message.

Process Oriented Messaging

Process-oriented messaging provides a communication path between applications to deliver information or request that an operation be performed by the process receiving the message. To use process-oriented messaging, you need to be familiar with these ToolTalk concepts: Processes, Sessions and Files.

- Processes
  One execution of an application, tool, or program that uses the ToolTalk service is called a *process* in this chapter. A process is a SunOS process that has initialized and registered with the ToolTalk service.

- Procid
  When a process opens communication with the ToolTalk service, it receives a *process identifier* (procid). Applications who want to send a message to a specific process must know the receiving application's procid. To find out this information, the sender can look at a previous message sent by the target application. The ToolTalk service automatically fills in a message's sender attribute with the sender's procid.
- Ptype
  You can instruct the ToolTalk service to consider your application as a potential message receiver when no process is running the application. This is done by providing message patterns and instructions for starting the application in a *process type* (ptype). The ptype file is compiled with the ToolTalk type compiler, tt_type_comp, at application installation time to make the information available to the ToolTalk service.

  When an application has a ptype, part of the registration and initialization activities include registering the ptype with the ToolTalk service. Registering a ptype will automatically register the message patterns listed in it.

- Sessions
  A group of processes running in the same X session or process tree session is called a *session* in this chapter. A session also contains an instance of the ToolTalk communication program ttsession.

  When a process opens communication with the ToolTalk service, the process that actually provides the procid is ttsession. A side effect of opening ToolTalk communication is that the session in which the action takes place becomes the default session for the process.

  The concept of a session is important in the delivery of messages. Senders can "scope" a message to a session and the ToolTalk service will deliver it to all processes that have message patterns that refer to the current session. To update message patterns with the current *session identifier* (sessid), applications "join" the session.

- Files
  A container for data that is of interest to applications is called a *file* in this chapter. The concept of a file is important in the delivery of messages. Senders can scope a message to a file and the ToolTalk service will deliver it to all processes that have message patterns that refer to the file without regard to the process's default session. To update message patterns with the current file pathname, applications "join" the file.

  It is also possible to scope a message to a file within a session. The ToolTalk service will deliver the message to all processes that refer to both the file and session in their message patterns.

Object Oriented Messaging

Object-oriented messages are addressed to objects managed by applications. Applications that create an object-oriented message address the message to either a specific object or to a particular type of object. Object-oriented messages are particularly useful for applications that currently use objects or that are to be designed around objects. If an existing application is not object-oriented, the ToolTalk service allows applications to identify portions of application data as objects so that applications can begin to communicate about these objects.

To use object-oriented messaging, you need to be familiar with process-oriented messaging concepts and the ToolTalk concept of object.

- Object Data

Object data is stored in two parts as shown in Figure 6-3.

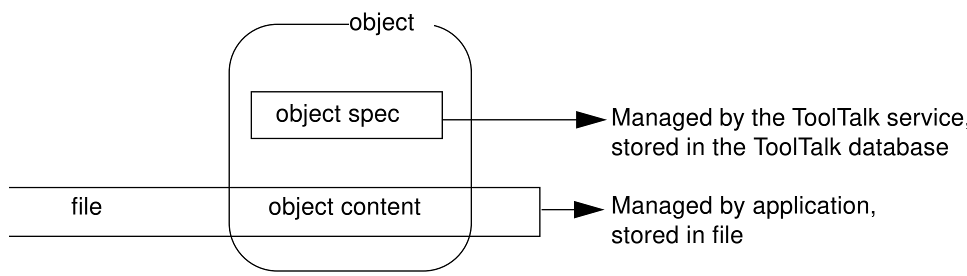

Figure 6-3 ToolTalk Object Data

One part is called the *object content*. The object content is managed by the application that creates or manages the object and is typically a piece, or pieces, of an ordinary file: a paragraph, a source code function, or a range of spreadsheet cells, for example.

The second part is called the *object specification* (*spec*). A spec contains standard properties such as the type of object, the name of the file in which the object contents are located, and the object owner. Applications can also add their own properties to a spec, for example, the location of the object content within a file. Because applications can store additional information in specs, you can identify data in existing files as objects without changing the formats of the files. You can also create objects from pieces of read-only files. Applications create and write specs to the ToolTalk database managed by `rpc.ttdbserverd`.

Note – You cannot create objects in files that reside in a read-only file system. The ToolTalk service must be able to create a database in the same file system that contains the object.

A *ToolTalk object* is a portion of application data for which a ToolTalk spec has been created.

The ToolTalk Architecture

The following ToolTalk service components work together to provide interapplication communication and object information management:

- `ttsession` is the ToolTalk communication process.
  One `ttsession` runs in an *X* server session or process tree session and communicates with other `ttsessions` when a message needs to be delivered to an application in another session.

- `rpc.ttdbserverd` is the ToolTalk database server process.
  One `rpc.ttdbserverd` is installed on each machine that contains a disk partition that stores files of interest to ToolTalk clients or files that contain ToolTalk objects.

  File and ToolTalk object information is stored in a records database managed by `rpc.ttdbserverd`.

- `libtt` is the ToolTalk application programming interface (API) library.
  Applications include the API library in their program and call the ToolTalk functions in the library.

The ToolTalk service uses the SunSoft ONC Remote Procedure Call (RPC) to communicate between these ToolTalk components.

Applications provide the ToolTalk service with process and object type information. This information is stored in the ToolTalk Types database.

Figure 6-4 illustrates the ToolTalk service architecture.

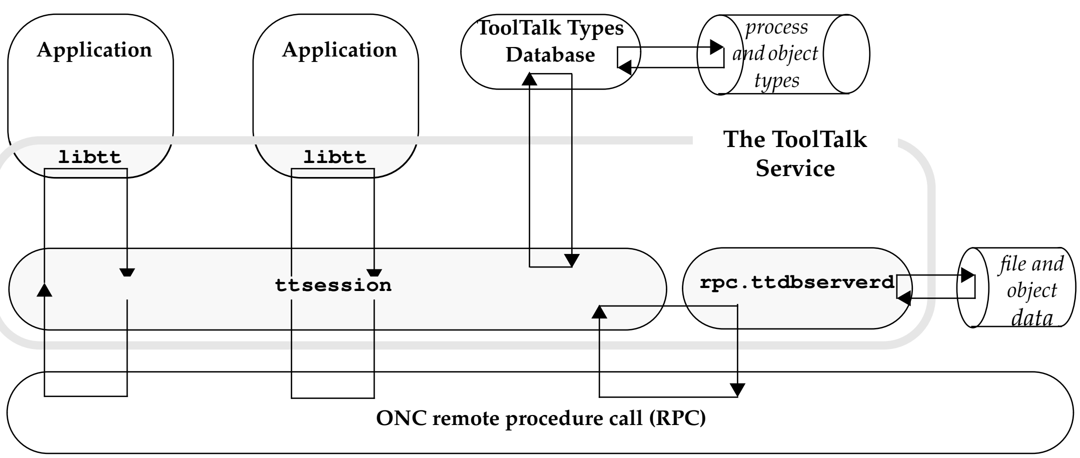

Figure 6-4 ToolTalk Service Architecture

ToolTalk Messages

ToolTalk messages contain attributes that store message information and provide delivery information to the ToolTalk service. ToolTalk uses this delivery information to route the messages to the appropriate receivers.

ToolTalk messages are simple structures containing attributes for address, subject (*operation* and *arguments*), and delivery information (*class and scope*). Each message consists of attributes such as those shown in Table 6-3.

Table 6-3 Basic Message Attributes

Attribute	Value
Address	procedure, process, object, object type
Class	request or notice
Operation	action to be performed, or event that has occurred
Arguments	input values for operation, output results
Scope	session, file, both, file-in-session

Note – These are fundamental attributes.There are more message attributes available for use by an application. See *The ToolTalk Service: An Inter-Operability Solution* for a complete listing of message attributes.

Messages addressed to other applications can be addressed to a particular process (*process* address) or any process that has registered a pattern that matches your message (*procedure* address). When addressing a message to a process, you need to know the process identifier (*procid*) of the other application. Applications receive a *procid* when they open communication with ToolTalk. The *procid* is unique within the user's session in which the application was started.

However, it is unusual for one process to know another's *procid*; more often a sender does not care which process performs an operation (request message) or learns of an event (notice message).

Applications can send two classes of ToolTalk messages, *notices* and *requests.* A notice is informational, a way for an application to announce an event. Applications that receive a notice absorb the message without returning results to the sender. A request is a call for an action, with the results of the action recorded in the message, and the message returned to the sender as a reply.

Applications using the ToolTalk service to communicate usually have something in common– the applications are running in the same session or they are interested in the same file or data. Applications register this interest by joining sessions or files (or both) with the ToolTalk service. This file and session information is used by the ToolTalk service in conjunction with the message patterns to determine what applications should receive a message.

- File Scope
  When a message is "scoped" to a file, only those applications that have joined the file (and match the remaining attributes) will receive the message. Applications that share interest in a file do not have to be running in the same session.
- Session Scope
  When a message is scoped to a session, only those applications that have joined the session will be considered as potential recipients.
- File In Session Scope
  Applications can be very specific about the distribution of a message by specifying file-in-session for the message scope. Only those applications that have joined both the file and the session indicated will be considered as potential recipients.

How the ToolTalk Service Routes Messages

A notice takes a one-way trip, as shown in Figure 6-5. The sender creates a message, fills in attribute values, and sends it. The ToolTalk service matches message and pattern attribute values, and gives a copy of the message to one handler and to all matching observers. File-scoped messages are automatically transferred across session boundaries to processes that have declared interest in the file.

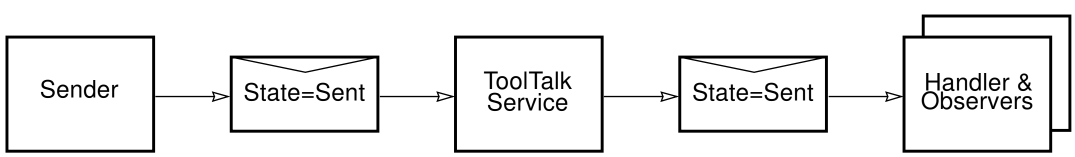

Figure 6-5 Notice Routing

A request, as shown in Figure 6-6, takes a round-trip from sender to handler and back; copies of the message take one-way side trips to observers. The ToolTalk service delivers a request to, at most, one handler. The handler adds results to the message and sends it back. Other processes can observe a request before it is handled, after, or both; observers absorb a request without sending it back.

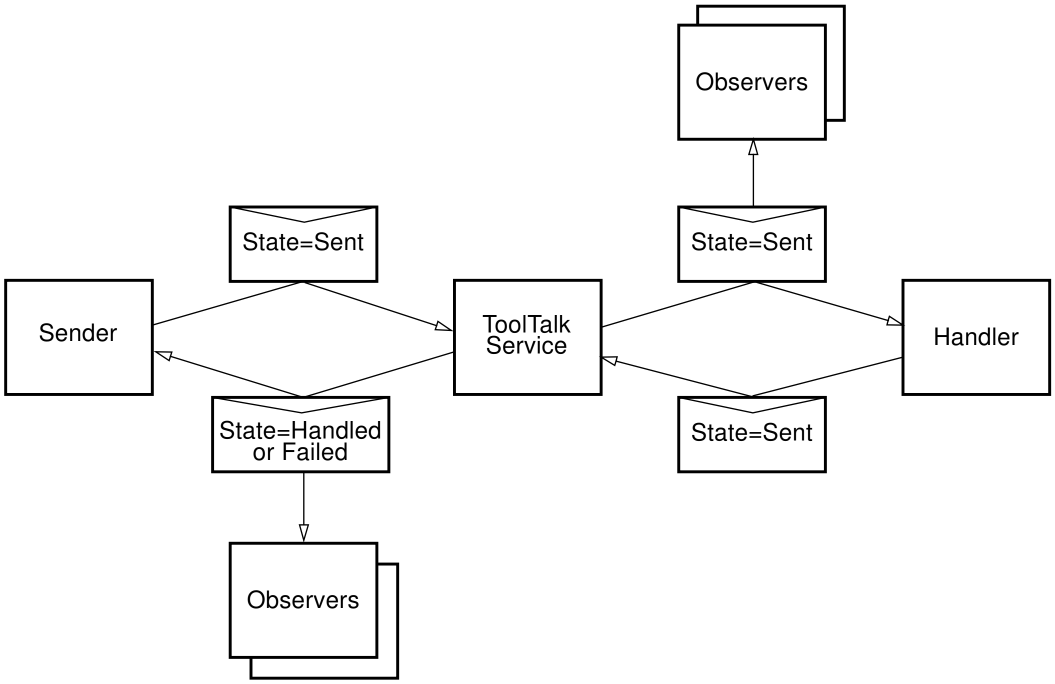

Figure 6-6 Request Routing

Using ToolTalk in Your Application

The steps you take in modifying your application to use the ToolTalk service fall into these groups:

- Including the API header files
- Preparing for interaction with the ToolTalk service
- Using message patterns
- Registering interest in a session or file
- Sending messages
- Receiving messages
- Stopping interaction with the ToolTalk service

The ToolTalk API header file is `$OPENWINHOME/share/include/desktop/tt_c.h`.

To prepare for interacting with the ToolTalk Service, you must

- open communication with the ToolTalk Service

- obtain a file descriptor that will notify you when messages have arrived
- configure your application to receive messages

To initialize your process with the ToolTalk service and get set up to receive messages, you must obtain a process identifier (`procid`) and a file descriptor (`fd`). The file descriptor is used to inform your process when messages are delivered. `tt_open()` returns the `procid` for your process, and sets this `procid` as the default `procid`.

`tt_fd()` returns a file descriptor. When a message arrives for the default `procid` in the default session, the file descriptor becomes active.

The following sample code initializes and registers with the ToolTalk service:

```
/*
 * Initialize ToolTalk, using the initial default session, and
 * obtain the file descriptor that will become active whenever
 * ToolTalk has a message for this process.
 */

my_procid = tt_open();
ttfd = tt_fd();
```

An incoming message activates the file descriptor returned by `tt_fd()`. For an XView-based program, use a callback function that will be invoked when the file descriptor becomes active. The callback is registered using `notify_set_input_func()`, as shown in the following code fragment:

```
ttfd = tt_fd();

/*
 * Arrange for XView to call receive_tt_message when the ToolTalk
 * file descriptor becomes active.
 */

notify_set_input_func(base_frame, receive_tt_message, ttfd);
```

For an OLIT-based application, use `XtAddInput()` or `XtAppAddInput()`, as shown in the following code fragment:

```
ttfd = tt_fd();
/* Arrange for ReceiveToolTalkMessage to be called */
XtAppAddInput(appcontext, ttfd, (XtPointer)XtInputReadMask,
(XtInputCallbackProc) ReceiveToolTalkMessage, (XtPointer)some_client_data);
```

Other programs may want to use the `select()` system call.

After setting up to receive messages, you need to tell the ToolTalk service what types of messages you want to receive. To create and register message patterns that the ToolTalk service will use when determining message recipients, you:

- obtain a message pattern handle
- set and add pattern attributes
- register the message pattern
- unregister the message pattern when you no longer want to receive messages that match this pattern

The following sample code fragment shows how to create and register a pattern that is:

- to be observed by the process (`TT_OBSERVE`)
- scoped to the same X or process tree session (`TT_SESSION`)
- has the pattern `ttsample1_value`

```
/*
 * Create and register a pattern so ToolTalk knows we are
 * interested in "ttsample1_value" messages within the session we
 * join.
 */

pat = tt_pattern_create();
tt_pattern_category_set(pat, TT_OBSERVE);
tt_pattern_scope_add(pat, TT_SESSION);
tt_pattern_op_add(pat, "ttsample1_value");
tt_pattern_register(pat);
```

After registering message patterns, join a session or file to update the session and file attributes in your message pattern. If you specified a scope of `TT_SESSION` in your message pattern, you need to join the session to notify the ToolTalk service to add your `sessid` to the `sessid` attribute of the message pattern. This automatic update of message patterns by joining sessions or files saves the effort of registering a new pattern for each session or file your application is interested in.

The following sample code joins the default session in which this program was started:

```
/*
 * Join the default session
 */

tt_session_join(tt_default_session());
```

To end participation in a session, use `tt_session_quit()` and the session will be removed from your session-scoped messages.

Interest in file-scoped messages is registered with `tt_file_join()` and removed with `tt_file_quit()`.

To send a message, you must first create the message and fill in its attributes; then use `tt_message_send(tt_message msg_out)` to actually send the message. Procedure addressed notice or request messages are created with `tt_pnotice_create()` or `tt_prequest_create()`. Object addressed messages are created with `tt_onotice_create()`, and `tt_orequest_create()`. These functions cause the ToolTalk service to allocate a message and return its handle. The application then uses the handle to fill in scope, operation and operation arguments for the message.

This code sample creates and sends a `pnotice` from the `ttsample1` program:

```
	/*
	 * Create and send a ToolTalk notice message
	 * ttsample1_value(in int new_value)
	 */

	msg_out = tt_pnotice_create(TT_SESSION, "ttsample1_value");
	tt_message_arg_add(msg_out, TT_IN, "integer", NULL);
	tt_message_arg_ival_set(msg_out, 0, (int)xv_get(slider,PANEL_VALUE));
	tt_message_send(msg_out);

	/*
	 * Since this message is a notice, we don't expect a reply, so
	 * there's no reason to keep a handle for the message.
	 */

	tt_message_destroy(msg_out);
}
```

When the ToolTalk-supplied file descriptor becomes active, your callback function is invoked. This is how an application is notified that a message is available. To get the message handle, use `tt_message_receive()`. The handle can then be used to retrieve attributes of the message.

When you use functions that return information such as the `tt_message_<attribute>()` calls, the ToolTalk service creates a copy of the information (the original copy is in the message structure in this case) and stores the copy in an allocation stack in the API library. The ToolTalk service actually returns a pointer to the copy to you. If you do not need the information returned by the function, use `tt_free()` to free the storage in the API stack.

The ToolTalk API also provides a more general mechanism, the `tt_mark()` and `tt_release()` functions, to mark and free information returned by a series of functions. These functions are typically used at the beginning and end of a routine where the information returned by the ToolTalk service is no longer interesting after the routine has ended.

Note – The API allocation stack should not be confused with your program's runtime stack. The ToolTalk service returns information from ToolTalk functions by providing pointers to the information in the API stack. The API stack will not discard information until you tell it to.

You can recognize replies to messages you sent by comparing the handles unless you are not interested in replies. This is why it is important not to destroy a request after you send it. When you have finished examining a message you received, be sure to destroy the message using `tt_message_destroy()` to free memory allocated for the message.

The following sample code fragment shows message retrieval and checking for the operation `ttsample1_value`:

```
/*
 * When a ToolTalk message is available, receive it; if it's a
 * ttsample1_value message, update the gauge with the new value.
 */
void
receive_tt_message()
{
    Tt_message msg_in;
    int mark;
    int val_in;

    msg_in = tt_message_receive();

    /*
     * It's possible that the file descriptor would become active
     * even though ToolTalk doesn't really have a message for us.
     * The returned message handle is NULL in this case.
     */

    if (msg_in == NULL) return;


    /*
     * Get a storage mark so we can easily free all the data
     * ToolTalk returns to us.
     */

    mark = tt_mark();

    if (0==strcmp("ttsample1_value", tt_message_op(msg_in,0))) {
        tt_message_arg_ival(msg_in, 0, &val_in);
        xv_set(gauge, PANEL_VALUE, val_in, NULL);
    }

    tt_message_destroy(msg_in);
    tt_release(mark);
    return;
}
```

Use `tt_close()` to shutdown interaction with ToolTalk and allow it to clean up. This is generally done as part of your application's exit procedures.

The ToolTalk Service, DeskSet Integration, and You

The ToolTalk services allows your application to exchange messages with DeskSet applications. These messages can be commands to start an application, load a specified file, or execute a process on a file or object. Currently, the communication is one-way in that your application may send requests to DeskSet applications, but DeskSet applications will not send requests to your application.

A common set of ToolTalk requests for many of the DeskSet applications has been specified. Each application is responsible for handling these requests. The following DeskSet applications support the same core of DeskSet requests:

- Mail Tool
- Audio Tool
- Text Editor
- Binder
- Color Changer
- Icon Editor
- PageView

DeskSet Supported Messages

All of the messages in the DeskSet protocol should be sent as point-to-point requests via `tt_message_handler_set()` to the DeskSet tool that is intended to handle them. The only exception is the `launch()` request, which must be sent first to start the handling application. After this, requests may be sent in any order. Typically, one application would `launch()` another, send the appropriate data using the `dispatch_data()` request, send the `move()` request to position the handler's base frame at a desired location and finally send an `expose()` request to force the handler to appear.

- `launch (string locale, int depth, int visual)`
  Starts the handling application. The first argument indicates which language should be used to display text. The second and third arguments indicate which X11 depth and visual the DeskSet application should use.

- `status (string status_string)`
  Send any status information to the handling process. The handling process will determine what to do with this status. An example of what status a sending process might send is the name of the sending application which the handling process could display in its footer to inform the user of the application is in control at the moment.

- `dispatch_data(string sel_name)`
  Sends data to the handling process. The `sel_name` is the selection name used for the data transfer. The handler may then access this selection and retrieve the data.
- `move(int xpos, int ypos, int width, int height, int placement)`
  The sending process would like the handling process to move its (the handling process) base frame to a new position. Arguments are the sender's frame location (x and y coordinates), sender's frame size (height and width) and a hint as to where the handler should position itself (`DS_POPUP_LEFT`, `DS_POPUP_RIGHT`, `DS_POPUP_ABOVE`, `DS_POPUP_BELOW`, `DS_POPUP_LOR`, `DS_POPUP_AOB`).
- `quit()`
  The sending process would like the handling process to exit.
- `hide()`
  The sending process would like the handling process' window to disappear, but not exit.
- `expose()`
  The sending process would like the handling process' window to appear.
- `retrieve_data(in string sel_name)`
  The sending process would like the handling process to send back any data that it has manipulated. The argument is the selection name used for the data transfer.

The sequence of steps an application should follow is:

1. Send the `launch()` message first.
2. Receive the `launch()` reply from the handler.
3. Send data using `dispatch_data()` message if any data manipulation is to be used by the DeskSet application.
4. Send an `expose()` message.

The following code fragments show an OLIT-based application that launches the TextEdit DeskSet application, handles the reply, and sends an expose message. No data is dispatched.

```
/* These are code fragments that show a program that sends a launch */
/* message to a Tooltalk-aware DeskSet application and receives the reply. */

#include <desktop/tt_c.h>

...

Tt_message tt_msg;
int ttfd;
char *my_session;
char *my_procid;

Display *display;
int visual, depth;
char *value;

main(int argc, char **argv)
{
XtAppContext appcontext;
Widget toplevelshell;

  ...

  /* set my default session, open tooltalk session,and obtain file descriptor.*/
  my_procid = tt_open();
  ttfd=tt_fd();
  my_session = tt_session_join(tt_default_session());

...

/* Arrange for handle_tooltalk to be called */
XtAppAddInput(appcontext,ttfd,(XtPointer)XtInputReadMask,(XtInputCallbackProc
)handle_tooltalk,(XtPointer)my_session);
...

/* Get X server info needed as args to "launch" message */
/* Locale info: assume "C" */
value = "C";
display = XtDisplay(toplevelshell);
depth = DefaultDepthOfScreen(XtScreen(toplevelshell));
visual = GetVisualClass(display, depth);
```

```
...

/* launch TextEdit application */
send_deskset_message("launch", 0);
/* main processing loop */
XtAppMainLoop(appcontext);
}

/* get visual class - needed as argument to "launch" message */
int GetVisualClass(Display *dsp, int depth)
{
register int i;
XVisualInfo *vinfo;
XVisualInfo vinfo_template;
int nvinfo;

/* setup template */
vinfo_template.depth = depth;
/* get visuals corresponding to depth */
vinfo = XGetVisualInfo(dsp, VisualDepthMask, &vinfo_template, &nvinfo);
/* go through list of visuals and return default visual's class */
for (i=0; i < nvinfo; i++) {
    if (vinfo[i].visual == DefaultVisualOfScreen(XtScreen(toplevelshell))) {
return(vinfo[i].class);
}
}

/* Function invoked when ToolTalk file descriptor becomes active */
void handle_tooltalk(XtPointer client_data, int *fd, XtInputId *id)
{
  Tt_message message;
  char *tt_message_name;

  if ((message = tt_message_receive()) != NULL) {
      tt_message_name = tt_message_op(message);
      switch(tt_message_state(message))
    {
    case TT_SENT:
      break;
    case TT_STARTED:
    case TT_FAILED:
    case TT_REJECTED:
    case TT_HANDLED: /* this is our reply */
```

```
                handle_message(message);
        break;
      case TT_CREATED:
          case TT_QUEUED:
          default:
                break;
      }
      }
   return;
}


/* helper function for handle_tooltalk */
void handle_message(Tt_message msg)
{
   char *message_name = tt_message_op(msg);

   /* just a simple case -- if the reply was from our "launch" */
   /* message, send an "expose" message back to the sender. */
   if (strcmp(message_name, "launch")) {
       send_deskset_message("expose", tt_message_sender(msg));
   }
}


/* function to create and send a ToolTalk message. */
void send_deskset_message(char *op, char *pt2pt_recipient)
{
   tt_msg = tt_message_create();
   tt_message_address_set(tt_msg, TT_OTYPE);
   tt_message_class_set(tt_msg, TT_REQUEST);
   tt_message_scope_set(tt_msg, TT_SESSION);
   tt_message_session_set(tt_msg, tt_default_session());
   tt_message_op_set(tt_msg, op);
   if (pt2pt_recipient == 0) {
       /* It must be a launch request. */
       tt_message_otype_set(tt_msg, "Sun_DeskSet_Object_textedit");
       tt_message_handler_ptype_set(tt_msg, "Sun_DeskSet_textedit");
       tt_message_disposition_set(tt_msg, TT_START);

       /* DeskSet tools expect three argument: locale, depth, visual class */
       tt_message_arg_add(tt_msg, TT_IN, "string", value);
       tt_message_arg_add(tt_msg, TT_IN, "int", depth);
```

```
      tt_message_arg_add(tt_msg, TT_IN, "int", visual);
  } else {
      /* It must be a point-to-point message. */
      tt_message_handler_set(tt_msg, pt2pt_recipient);
  }
  tt_message_send(tt_msg);
  tt_message_destroy(tt_msg);
}
```

The following XView-based sample code, which is an example of object-oriented messaging, creates an object for its user: it creates the object spec, sets the `otype`, writes the spec to the ToolTalk database, and wraps the user's selection with C-style comments. The application also sends out a procedure-addressed notice after it creates the new object to update other applications who observe messages with the `ToolTalk_EditDemo_new_object` operation. If other applications are displaying a list of objects in a file managed by `ToolTalk_EditDemo`, they update their list after receiving this notice.

```
/*
 * Make a ToolTalk spec out of the selected text in this textpane. Once
 * the spec is successfully created and written to a database, wrap the
 * text with C-style comments in order to delimit the object and send out
 * a notification that an object has been created in this file.
 */
Menu_item
edit_ui_make_object(item, event)
   Panel_itemitem;
   Event*event;
{
         int               mark = tt_mark();
    char    *objid;
    char    *file;
    char    *sel;
    Textsw_indexfirst, last;
    char    obj_start_text[100];
    char    obj_end_text[100];
    Tt_messagemsg;

    if (! get_selection(edit_ui_xserver, edit_ui_textpane,
              &sel, &first, &last)) {
        xv_set(edit_ui_base_window, FRAME_LEFT_FOOTER,
               "First select some text", NULL);
```

```
        tt_release(mark);
        return item;
    }
    file = tt_default_file();

    if (file == (char *)0) {
        xv_set(edit_ui_base_window, FRAME_LEFT_FOOTER,
               "Not editing any file", NULL);
        tt_release(mark);
        return item;
    }

    /* create a new spec */

    objid = tt_spec_create(tt_default_file());
    if (tt_pointer_error(objid) != TT_OK) {
        xv_set(edit_ui_base_window, FRAME_LEFT_FOOTER,
               "Couldn't create object", NULL);
        tt_release(mark);
        return item;
    }

    /* set its otype */

    tt_spec_type_set(objid, "Sun_EditDemo_object");
    if (tt_spec_write(objid) != TT_OK) {
        xv_set(edit_ui_base_window, FRAME_LEFT_FOOTER,
               "Couldn't write out object", NULL);
        tt_release(mark);
        return item;
    }

    /* wrap spec's contents (the selected text) with C-style */
    /* comments. */

    sprintf(obj_start_text," /* begin_object(%s) */", objid);
    sprintf(obj_end_text,"/* end_object(%s) */", objid);
    (void)wrap_selection(edit_ui_xserver, edit_ui_textpane,
              obj_start_text, obj_end_text);

    /* now send out a notification that we've added a new object */

    msg = tt_pnotice_create(TT_FILE_IN_SESSION,"Sun_EditDemo_new_object");
    tt_message_file_set(msg, file);
```

```
    tt_message_send(msg);

    tt_release(mark);
    return item;
}
```

ToolTalk Utility Programs

Caution – Despite the efforts of the ToolTalk service and integrated applications, object references can still be broken if you remove, move, or rename files with standard operating system commands such as `rm` or `mv`. Broken references will result in undeliverable messages.

Managing Files that Contain Object Data

To keep the ToolTalk database that services the disk partition where a file that contains object data is stored up-to-date, use the ToolTalk functions to copy, move, or destroy the file. Table 6-4 lists the ToolTalk functions you use to manage files that contain object data.

Table 6-4 Functions to Copy, Move, or Remove Files that Contain Object Data

ToolTalk Function	Description
`tt_file_move(const char *oldfilepath, const char *newfilepath)`	Moves the file and the ToolTalk object data
`tt_file_copy(const char *oldfilepath, const char *newfilepath)`	Copies the file and the ToolTalk object data
`tt_file_destroy(const char *filepath)`	Removes the file and the ToolTalk object data

The return type for these functions is `Tt_status`.

Managing Files that Contain ToolTalk Information

The ToolTalk service provides ToolTalk-enhanced shell commands to copy, move, and remove ToolTalk object and file information. Table 6-5 lists the ToolTalk-enhanced shell commands that you and users of your application should use to copy, move, and remove files referenced in messages and files that contain objects.

Table 6-5 ToolTalk-Wrapped Shell Commands

Command	Description
`ttcp`	Copies file to new location. Updates file and object location information in ToolTalk database.
`ttmv`	Renames directory or files. Updates file and object location information in ToolTalk database.
`ttrm`	Removes specified file. Removes file and object information from the ToolTalk database.
`ttrmdir`	Removes empty directories (directories that contain no files) that have ToolTalk object specs associated with them. (It is possible to create an object spec for a directory; when an object spec is created, the pathname of a file or directory is supplied.) Removes object information from the ToolTalk database.
`tttar`	Archives (or extracts) multiple files and object information into (or from) a single archive, called a tarfile. Can also be used to only archive (or extract) ToolTalk file and object information into (or from) a tarfile.

Standard Message Sets Using the ToolTalk Service

Applications usually are not developed concurrently. And, typically, they are developed by multiple groups. Application users often want to substitute their favorite applications, so they want the universal plug-and-play across all applications. The ToolTalk standard message sets provide integration benefits to users of compliant applications.

By having messages organized into discrete sets, you get a pre-arranged interface between applications with common interests. These sets are specific to an application domain; for example, accounting or computer aided software engineering. Figure 6-7 represents the relationships among the message sets.

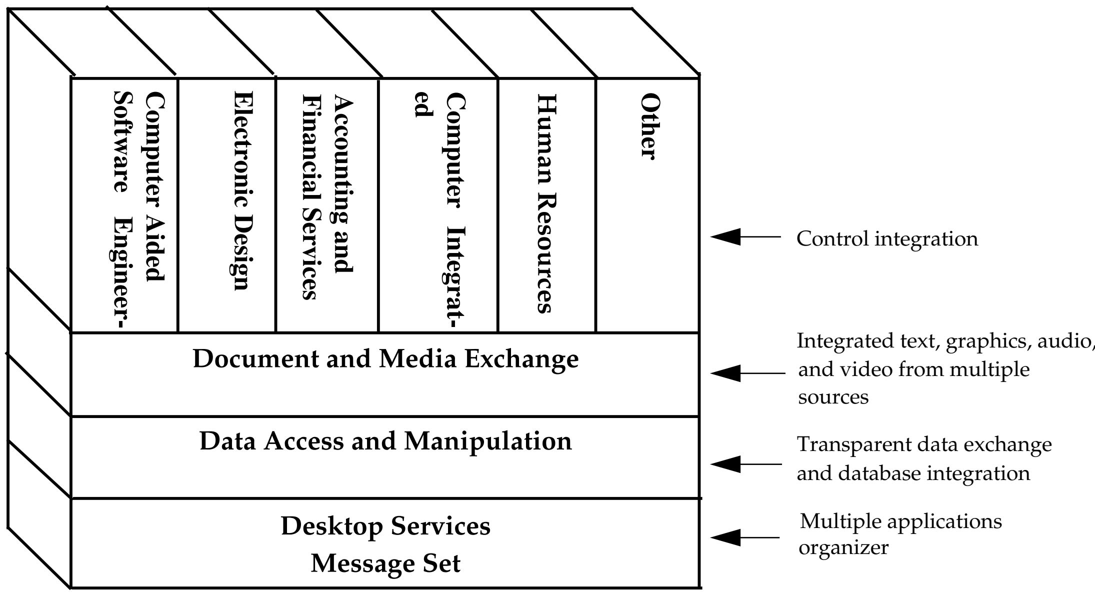

Figure 6-7 Standard ToolTalk Message Sets and Integration Benefit Examples

As the result of extensive work to define message standards with leading software suppliers and end users, SunSoft provides these well-defined, highly functional, standard message sets to anyone. They are open specifications in the public domain without royalty or license fees.

ToolTalk-compliant applications realize many integration benefits from the various message sets, which are described below.

Desktop Services Message Set

To achieve basic desktop integration, applications need to support a basic set of messages to enable inter-application control. With the Desktop Services Message Set, an application can manage the state of files, windowing, and processing for itself or any other application. A powerful messaging protocol that benefits both developers and users of desktop applications, the Desktop Services Message Set, allows applications to interact easily with other desktop applications.

For integrated control of applications to take place, certain basic features are needed to launch, halt, control display appearance, and pass information regarding input and output data. All applications need to have these facilities so that other applications in the toolset can interchange basic control information. This kind of functionality enables the development of smart desktops and integrated smart toolsets. By consistent adoption of these messages, groups of applications can now call upon one another to perform tasks and to interact as one solution environment for the end user.

All applications using the desktop should pay particular attention to the Desktop Services Message Set.

The full message set is defined in the document titled *The ToolTalk Desktop Services Message Set: Release 1.1*. Send questions, comments, and requests for information to the Desktop Services Messaging Alliance at **ToolTalk_desktop_services@sun.com**. (Note: This alias is a forum for detailed technical interactions and is not a sales/marketing forum.)

Data Access and Manipulation Message Set

End users who access databases and manipulate information in spreadsheets and decision support applications need transparent integration. The Data Access and Manipulation Message Set makes data integration seamless and transparent to users.

Document and Media Exchange Message Set

No single vendor provides a completely integrated solution that meets the complex needs of today's market. The ToolTalk Document and Media Exchange Message Set allows any application to incorporate text, graphics, audio, and video generated by another application. Virtually all applications can benefit from close integration of documentation and multimedia.

Using ToolTalk Document and Media Exchange Message Set, multimedia applications can communicate with one another in a transparent manner, both locally and over networks, regardless of data formats, compression technology, and other technical issues all of which have previously confined the use of this technology.

Applications can be made multimedia-aware without having to individually handle the complexities of numerous media types.

The ToolTalk Document and Media Exchange Message Set is an efficient set of generic message definitions that provide media control and data exchange. The protocol consists of editor messages for media players, editors and users.

The message set contains five messages oriented towards creating, editing and using documents of a certain media type. All five share common parameters (arguments) that specify the following:

- *media type*—The name of a media format. Allows messages to be dispatched to the correct media editor. For example an audiofile.
- *component* of the document in question, such as a chapter or page in a word-processing document.

- *locator*—Consists of pairs of components and component identifiers
- *region*—The portion of a document bounded by a series of locators.

It is expected that an application that uses the Document and Media Exchange Message Set will also support the Desktop Services Message Set.

The full message set is defined in the document *The ToolTalk Document and Media Exchange Message Set: Release 1.1*. Send questions, comments, and requests for information to the Document and Media Exchange Messaging Alliance at **media_exchange@Sun.Com**. (Note: This alias is a forum for detailed technical interactions and is not a sales/marketing forum.)

CASE Interoperability Message Sets

These message sets, developed jointly by Digital, SGI and SunSoft, provide CASE (Computer Aided Software Engineering) tools vendors with a protocol that encourages development of "plug-and-play" CASE tools.

To achieve control flow integration in an integrated CASE environment, applications need to support messages that enable inter-application control. This control integration allows applications to interchange state information and to provide services to each other in a controlled and well-behaved manner.

The CASE Interoperability Message Sets define a standard set of CASE functionality available through control integration services. Each message set describes an area of functionality in the CASE environment. The message sets are:

- Analysis and Design
- Browsing
- Building
- Debugging
- Editing
- Version Management

The *Analysis and Design Message set* defines messages for functions that deal with the software architecture of an application. Typical functions for manipulation include structure chart editors, data flow diagrams, and flow charts.

The *Browsing Message set* defines messages for functions that process queries on source code.

The *Building Message set* defines an interface to build utilities. This message set includes the initiation of build processes at different granularities, error notifications, and recording of dependencies.

The *Debugging Message set* defines a standard interface for a debug utility. This message set includes the common functions associated with debug utilities.

The *Editing Message set* defines a detailed collection of messages for text editing tools. These messages focus on the definition and manipulation of buffers and views and on text editing functions.

The *Version Management Message set* defines messages for tools that manage version histories of software artifacts. This message set includes checkin, checkout, and other related version control functions.

Note that the Desktop Services Message Set and the Document and Media Exchange Message Set are two sets that are part of the CASE Interoperability Message Sets. However, the CASE sets are at a high-level semantic description, whereas the others are more syntactic. That is, the CASE sets do not presume the ToolTalk service as the message distribution mechanism, but it is very easy to derive the syntax of a CASE message applied to ToolTalk.

The CASE sets are an example of a vertical market interoperability alliance. While obviously of interest to software development tool vendors, they are a good starting point for vendors in other application areas.

The full message sets are defined in the document *The CASE Interoperability Message Sets: Release 1.1*. Send questions, comments, and requests for information to the CASE Messaging Alliance at **`casemsg.ext@sun.com`**. (Note: This alias is a forum for detailed technical interactions and is not a sales/marketing forum.)

For Further Information on ToolTalk

The documents listed in Table 6-6 contain further information on the ToolTalk service. These references facilitate the process of adopting ToolTalk message sets. Note that you can use only the message sets that meet the functional integration needs of your customers. Your applications can then interoperate with any other application with the same message set.

Table 6-6 Documents Related to the ToolTalk Service

Documents	Topics	Audience
The ToolTalk Service: An Inter-Operability Solution	How to integrate with the ToolTalk service; ToolTalk API reference	Application developer, system administrator
Solaris 2.2 Desktop Integration Guide	Information on the Classing Engine, a desktop-style database used by the ToolTalk service and the ToolTalk message protocol used by the DeskSet	Application developer
OpenWindows Version 3.2 Reference Manual	man pages for ToolTalk binary files and utilities	Advanced user, system administrator, application developer
Solaris 2.2 System Configuration and Installation Guide	Installation of the Solaris 2.x operating environment including the OpenWindows software	System administrator, application developer

Bibliography

OpenWindows Version 3.1 Desktop Integration Guide, Part Number 801-2873-10, SunSoft, Inc., Mountain View, CA

X Protocol Reference Manual, O'Reilly & Associates, Sebastopol, CA.

X Toolkit Intrinsics Programming Manual, O'Reilly & Associates, Sebastopol, CA.

Xlib Programming Manual, O'Reilly & Associates, Sebastopol, CA.

X Toolkit Intrinsics Programming Manual, O'Reilly & Associates, Sebastopol, CA.

The X Window System Programming & Applications with Xt: OPEN LOOK Edition, Douglas A. Young & John A. Pew, Prentice Hall, Englewood Cliffs, NJ.

OpenWindows Version 3.0.1 Programmer's Guide, Part Number 800-6388-10, SunSoft, Inc., Mountain View, CA.

OLIT 3.1 Reference Manual, Part Number 800-3436-10, SunSoft, Inc., Mountain View, CA.

*OPEN LOOK Graphical User Interface Functional Specificatio*n, Addison-Wesley Publishing Co., Reading, MA.

XView Programming Manual, O'Reilly & Associates, Sebastopol, CA.

The ToolTalk Service: An Inter-Operability Solution, SunSoft Press/Prentice Hall, Englewood Cliffs, NJ.

The ToolTalk Desktop Services Message Set: Release 1.1, A White Paper, by Ronald Lee, Brian Holtz, and Astrid Julienne, Part Number 92206-002, SunSoft, Inc., Mountain View, CA.

The ToolTalk Document and Media Exchange Message Set: Release 1.1, A White Paper by Dan Clark, Ronald Lee, Brian Holtz, and Astrid Julienne, Part Number 92204-002, SunSoft, Inc., Mountain View, CA.

CASE Interoperability Message Sets: Release 1.0, SunSoft, Inc., Mountain View, CA.