

ToolTalk User's Guide

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.



© 1991–1994 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX[®] and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

Sun, the Sun logo, Sun Microsystems, Sun Microsystems Computer Corporation, SunSoft, the SunSoft logo, Solaris, SunOS, OpenWindows, DeskSet, ONC, ONC+, NFS, NetISAM, and ToolTalk are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCcompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK[®] and Sun[™] Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



Table of Contents

1. Introducing the ToolTalk Service	1
ToolTalk Scenarios	2
Using the ToolTalk Desktop Services Message Set	2
Using the ToolTalk Document and Media Exchange Message Set	4
Using the CASE Interoperability Message Sets	6
How Applications Use ToolTalk Messages	8
Sending ToolTalk Messages	8
Message Patterns	8
Receiving ToolTalk Messages	9
ToolTalk Message Distribution	10
Process-Oriented Messages	10
Object-Oriented Messages	10
Determining Message Delivery	10
Modifying Applications to Use the ToolTalk Service	12
2. An Overview of the ToolTalk Service	13

ToolTalk Architecture	13
Starting a ToolTalk Session	15
Background and Batch Sessions.	17
X Window System	17
Locating ttsession	17
Maintaining ToolTalk Files and Databases	18
Demonstration Programs	18
3. Message Patterns.	19
Message Pattern Attributes	20
Scope Attributes.	22
Scoping to a Session Only.	22
Scoping to a File Only	23
Scoping to a File in a Session	24
Scoping to a File and/or a Session	25
Adding Files to Scoped Patterns	26
Context Attributes	27
Disposition Attributes.	28
4. Setting Up and Maintaining the ToolTalk Processes	29
Location of the ToolTalk Service Files	29
Version.	31
Requirements	31
Environment Variables	31
ToolTalk Environment Variables	31
Other Environment Variables.	33

Environment Variables Required to Start Programs on Remote Hosts	33
Using Context Slots to Create Environment Variables	35
Installing the ToolTalk Database Server	35
When the ToolTalk Service is Installed Elsewhere on the System.	36
Running the New ToolTalk Database Server	37
Redirecting the ToolTalk Database Server	37
Redirecting the Host Machine	37
Redirecting the File System Partition	38
5. Maintaining Application Information	41
Installing Application Types.	41
Examining ToolTalk Type Information	43
Removing ToolTalk Type Information.	43
Updating the ToolTalk Service	44
Process Type Errors	44
Using ttsnoop to Debug Messages and Patterns	45
Composing and Sending Messages.	48
Composing and Registering Patterns	50
Displaying Message Components	52
Sending Pre-Created Messages	53
Receiving Messages.	54
Stop Receiving Messages	54
6. Maintaining Files and Objects Referenced in ToolTalk Messages	55

ToolTalk-Enhanced Shell Commands	56
Maintaining and Updating ToolTalk Databases.	57
Displaying, Checking, and Repairing Databases.	57
7. Participating in ToolTalk Sessions.	59
Including the ToolTalk API Header File	59
Registering with the ToolTalk Service	60
Registering in the Initial Session	61
Registering in a Specified Session	62
Registering in Multiple Sessions	63
Setting Up to Receive Messages.	63
Sending and Receiving Messages in the Same Process.	64
Sending and Receiving Messages in a Networked Environment	65
Unregistering from the ToolTalk Service.	65
8. Sending Messages.	67
How the ToolTalk Service Routes Messages.	67
Sending Notices.	67
Sending Requests.	68
Changes in State of Sent Message	69
Message Attributes	69
Address Attribute	70
Scope Attributes.	70
Serialization of Structured Data.	74
ToolTalk Message Delivery Algorithm	74
Process-Oriented Message Delivery	74

Object-Oriented Message Delivery	77
Otype Addressing	80
Modifying Applications to Send ToolTalk Messages.	80
Creating Messages.	80
Adding Message Callbacks	87
Sending a Message	90
Examples.	91
9. Dynamic Message Patterns	93
Defining Dynamic Messages	93
Creating a Message Pattern	95
Adding a Message Pattern Callback	95
Registering a Message Pattern	96
Deleting and Unregistering a Message Pattern	96
Updating Message Patterns with the Current Session or File	97
Joining the Default Session.	97
Joining Multiple Sessions	98
Joining Files of Interest	99
10. Static Message Patterns	101
Defining Static Messages	101
Defining Process Types	101
Signatures	102
Creating a Ptype File	103
Automatically Starting a Tool	106
Defining Object Types	107

Signatures	108
Creating Otype Files	109
Installing Type Information	112
Checking for Existing Process Types	112
Declaring Process Type	113
Undeclaring Process Types	115
11. Receiving Messages	117
Retrieving Messages	117
Identifying and Processing Messages Easily	119
Recognizing and Handling Replies Easily	120
Checking Message Status	120
Examining Messages	120
Callback Routines	123
Callbacks for Messages Addressed to Handlers	124
Attaching Callbacks to Static Patterns	125
Handling Requests	125
Replying to Requests	125
Rejecting or Failing a Request	126
Destroying Messages	127
12. Objects	129
Object-Oriented Messaging	129
Creating Object Specs	131
Assigning Otypes	131
Determining Object Specification Properties	132

Storing Spec Properties	132
Adding Values to Properties	132
Writing Object Specs	132
Updating Object Specs	132
Maintaining Object Specs	133
Examining Spec Information	134
Comparing Object Specs	134
Querying for Specific Specs in a File	134
Moving Object Specs	136
Destroying Object Specs	137
Managing Object and File Information	137
Managing Files that Contain Object Data	137
Managing Files that Contain ToolTalk Information	138
An Example of Object-Oriented Messaging	139
13. Managing Information Storage	143
Information Provided to the ToolTalk Service	143
Information Provided by the ToolTalk Service	143
Calls Provided to Manage the Storage of Information	144
Marking and Releasing Information	144
Allocating and Freeing Storage Space	145
Special Case: Callback and Filter Routines	146
Callback Routines	146
Filter Routines	147
14. Handling Errors	149

Retrieving ToolTalk Error Status	150
Checking ToolTalk Error Status	150
Returned Value Status	150
Returned Pointer Status	151
Returned Integer Status	152
Broken Connections	153
Error Propagation	154
A. Migrating from the Classing Engine Database to the ToolTalk Types Database	155
The ttce2xdr Script	155
Converting the User Database	155
Converting the System Database	156
Converting the Network Database	157
B. A Simple Demonstration of How the ToolTalk Service Works	159
Inter-Application Communication Made Easy	159
Adding Inter-Operability Functionality	160
Modifying the Xedit Application	160
Modifying the Xfontsel Application	161
We Have Tool Communication!	163
Adding ToolTalk Code to the Demonstration Applications . . .	166
Adding ToolTalk Code to the Xedit Files	166
Adding ToolTalk Code to the Xfontsel Files	174
C. The ToolTalk Standard Message Sets	181
The ToolTalk Desktop Services Message Set	181

Why the ToolTalk Desktop Services Message Set was Developed	182
Key Benefits of the ToolTalk Desktop Services Message Set	182
The ToolTalk Document and Media Exchange Message Set. . .	182
Why the ToolTalk Document and Media Exchange Message Set was Developed	183
Key Benefits of the ToolTalk Document and Media Exchange Message Set	183
General ToolTalk Message Definitions and Conventions	184
Errors.	187
General ToolTalk Development Guidelines and Conventions.	188
Always Make Anonymous Requests.	189
Let Tools Be Started as Needed	189
Reply When Operation has been Completed.	190
Avoid Statefulness Whenever Possible.	190
Declare One Process Type per Role	190
Developing ToolTalk Applications	191
Messaging Alliances	193
D. Frequently Asked Questions	195

Figures

Figure 1-1	Applications Using the ToolTalk Service.....	1
Figure 2-1	ToolTalk Service Architecture	14
Figure 5-1	ttsnoop Display Window	46
Figure 5-2	Popup Messages Panel	48
Figure 5-3	Editing Send Contexts.....	49
Figure 5-4	Popup Patterns Panel	50
Figure 5-5	Editing Message Contexts in Patterns	51
Figure 5-6	Checkboxes to Display Message Component Values	52
Figure 5-7	Display of Message Components	53
Figure 5-8	Incoming Message Displayed	54
Figure 8-1	Notice Routing.....	67
Figure 8-2	Request Routing.....	68
Figure 8-3	Destroying a Message.....	88
Figure 8-4	Creating and Sending a Pnotice	91
Figure 8-5	Creating and Sending an Orequest.....	92
Figure 10-1	Example of a Ptype File	103

Figure 10-2	Ptype Syntax	104
Figure 10-3	Automatically Starting a Tool	107
Figure 10-4	Otype Syntax	109
Figure 10-5	Example of an Otype File	111
Figure 10-6	Example of a Ptype Being Registered During Initialization . .	114
Figure 11-1	Receiving a Message	118
Figure 11-2	Code Checking the Message State	119
Figure 11-3	Allocating, Examining, and Releasing Storage	121
Figure 11-4	How Callbacks Are Invoked	124
Figure 12-1	ToolTalk Object Data.	130
Figure 12-2	Obtaining a List of Specifications	135
Figure 14-1	Obtaining an Error Explanation	151
Figure 14-2	Retrieving a Returned Pointer Status	152
Figure 14-3	Checking a Returned Integer	153

Tables

Table 2-1	<code>ttsession</code> Command Line Options	15
Table 3-1	ToolTalk Message Pattern Attributes	20
Table 4-1	Environment Variables	32
Table 5-1	XDR-base Format ToolTalk Types Tables	42
Table 5-2	Classing Engine-base Format ToolTalk Types Tables	42
Table 6-1	ToolTalk-Enhanced Shell Commands	56
Table 7-1	Registering with the ToolTalk Service	60
Table 7-2	Code Used to Watch for Arriving Messages	64
Table 8-1	ToolTalk Message Attributes	69
Table 8-2	Scoping a Message with Patterns to a File	71
Table 8-3	Scoping Mechanisms for Messages	72
Table 8-4	Functions Used to Create Messages	81
Table 8-5	Functions Used to Complete Messages	81
Table 9-1	Functions for Creating, Updating, and Deleting Message Patterns	93
Table 9-2	ToolTalk Functions for Joining Default Sessions	97
Table 9-3	ToolTalk Functions for Joining Files of Interest	99

Table 11-1	Functions to Examine Message Attributes	121
Table 11-2	Functions to Reply to Requests	125
Table 11-3	Rejecting or Failing Requests	126
Table 12-1	Functions to Create	131
Table 12-2	Functions to Maintain Object Specifications	133
Table 12-3	Functions to Copy, Move, or Remove Files that Contain Object Data	137
Table 12-4	ToolTalk-Wrapped Shell Commands	138
Table 13-1	Managing ToolTalk Storage	144
Table 14-1	Retrieving ToolTalk Error Status	150
Table 14-2	ToolTalk Error Macros	150

Preface

This manual describes the ToolTalk[®] service and how you modify your application to send and receive ToolTalk messages. Topics in this manual include:

- General concepts of the ToolTalk service
- What the ToolTalk service is and how it works
- Requirements to set up and maintain the ToolTalk service
- What is required to integrate your application with the ToolTalk service
- How to modify your application to send messages addressed to processes or ToolTalk objects
- How to register message pattern information for the messages your application wants to receive
- How to receive and handle messages delivered to your application by the ToolTalk service
- How to create and manage ToolTalk objects in your application data
- How to maintain ToolTalk objects at the system administrator level and what users must do to maintain these objects and the files in which they are stored
- How the ToolTalk service enables an application to communicate with other applications

Who Should Use this Book

This guide is for developers who create or maintain applications that use the ToolTalk service to inter-operate with other applications; it is also useful for system administrators who set up workstations. This guide assumes familiarity with operating system commands, system administrator commands, and system terminology.

How This Manual Is Organized

This manual is organized as follows:

Chapter 1, “Introducing the ToolTalk Service,” describes how the ToolTalk service works and how it uses information that your application supplies to deliver messages; and how applications use the ToolTalk service.

Chapter 2, “An Overview of the ToolTalk Service,” describes new and changed features of this release; and application and ToolTalk components.

Chapter 3, “Message Patterns,” describes message pattern attributes.

Chapter 4, “Setting Up and Maintaining the ToolTalk Processes,” describes ToolTalk file locations, hardware and software requirements, how to find ToolTalk version information, and installation instructions for the ToolTalk database server.

Chapter 5, “Maintaining Application Information,” describes how to maintain application information.

Chapter 6, “Maintaining Files and Objects Referenced in ToolTalk Messages,” describes how to maintain files references in ToolTalk messages; how system administrators and users maintain ToolTalk objects; and how to perform maintenance on ToolTalk databases.

Chapter 7, “Participating in ToolTalk Sessions,” describes the location of the ToolTalk API header file; how you initialize your application and start a session with the ToolTalk service; how you provide file and session information to the ToolTalk service; how to manage storage and handle errors; and how to unregister your message patterns and close your communication with the ToolTalk service when your process is ready to quit.

Chapter 9, “Dynamic Message Patterns,” describes how to create a dynamic message pattern and register it with the ToolTalk service; and how to add callbacks to your dynamic message patterns.

Chapter 10, “Static Message Patterns,” describes how to provide process and object type information at installation time; how to make a static message pattern available to the ToolTalk Service; how to declare a ptype; and register it with the SunSoft Vendor Type Registration program.

Chapter 8, “Sending Messages,” describes the complete ToolTalk message structure; the ToolTalk message delivery algorithm; how to create, fill in, and send a ToolTalk message; and how to attach a callback to requests that will automatically call your callback routine when the reply to your request is delivered to your application.

Chapter 11, “Receiving Messages,” describes how to retrieve messages delivered to your application; how to handle the message once you have examined it; how to send replies; and when to destroy messages.

Chapter 12, “Objects,” describes how to create ToolTalk specification objects for the objects your process creates and manages.

Chapter 13, “Managing Information Storage,” describes how to manage and remove objects.

Chapter 14, “Handling Errors,” describes how to handle error conditions.

Appendix A, “Migrating from the Classing Engine Database to the ToolTalk Types Database,” describes how to convert the Classing Engine databases to the ToolTalk Types database.

Appendix B, “A Simple Demonstration of How the ToolTalk Service Works,” presents how the ToolTalk service can enable an application to communicate with other applications.

Appendix C, “The ToolTalk Standard Message Sets,” describes the ToolTalk message sets that have been developed to help you develop applications that follow the same protocol as other applications with which your application wants to inter-operate.

Appendix D, “Frequently Asked Questions,” provides the answers to some frequently asked questions about the ToolTalk service.

Related Documentation

The following is a list of related ToolTalk documentation:

- *ToolTalk Reference Manual*
- ToolTalk Message Sets
 - ∞ CASE Inter-Operability Message Sets
- ToolTalk and Open Protocols, ISBN 013-031055-7
Published by SunSoft Press/Prentice Hall

Introducing the ToolTalk Service

The ToolTalk service enables independent applications to communicate with each other without having direct knowledge of each other. Applications create and send ToolTalk messages to communicate with each other. The ToolTalk service receives these messages, determines the recipients, and then delivers the messages to the appropriate applications, as shown in Figure 1-1.

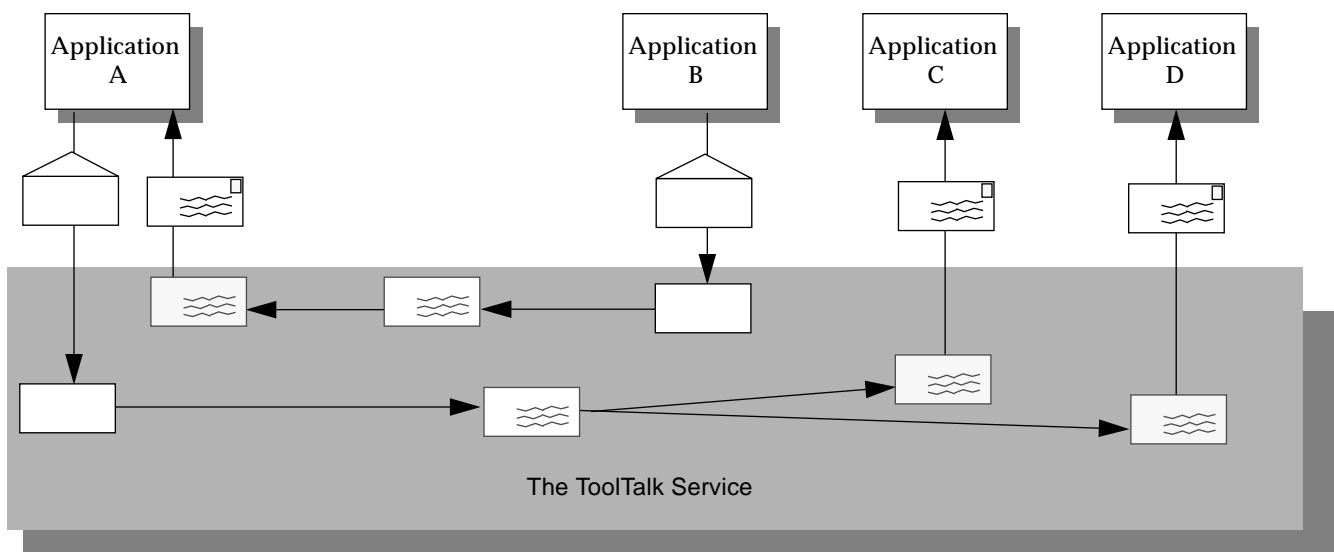


Figure 1-1 Applications Using the ToolTalk Service

1.1 *ToolTalk Scenarios*

The scenarios in this section illustrate how the ToolTalk service helps users solve their work problems. The message protocols used in these scenarios are hypothetical.

Using the ToolTalk Desktop Services Message Set

The ToolTalk Desktop Services Message Set allows an application to integrate and control other applications without user intervention. This section illustrates two scenarios that show how the Desktop Services Message Set might be implemented.

The Smart Desktop

A common user requirement for a graphic user interface (GUI) front-end is the ability to have data files be aware (or “know”) of their applications. To do this, an application-level program is needed to interpret the user’s requests. Examples of this application-level program (known as *smart desktops*) are the Apple Macintosh finder, Microsoft Windows File Manager, and the Solaris File Manager. The key common requirements for smart desktops are:

1. Takes a file
2. Determines its application
3. Invokes the application

The ToolTalk Service encompasses additional flexibility by allowing classes of tools to edit a specific data type. The following scenario illustrates how the Desktop Services Message Set might be implemented as a smart desktop transparent to the end-user.

- 1. Diane double clicks on the File Manager icon.**
 - The File Manager opens and displays the files in Diane’s current directory.
- 2. Diane double clicks on an icon for a data file.**
 - a. The File Manager requests that the file represented by the icon be displayed. The File Manager encodes the file type in the *display* message.

-
- b. The ToolTalk session manager matches the pattern in the *display* message to a registered application (in this case, the Icon Editor), and finds an instance of the application running on Diane's desktop.

Note – If the ToolTalk session manager does not find a running instance of the application, it checks the statically-defined ptypes and starts an application that best matches the pattern in the message. If none of the ptypes match, it returns failure to the File Manager application.

- c. The Icon Editor accepts the *display* message, de-iconifies itself, and raises itself to the top of the display.

3. Diane manually edits the file.

Integrated Toolsets

Another significant application for which the Desktop Services Message Set can be implemented is *integrated toolsets*. These environments can be applied in vertical applications (such as a CASE software developer toolset) or in horizontal environments (such as compound documents). Common to both of these applications is the premise that the overall solution is built out of specialized applications designed to perform one particular task well. Examples of integrated toolset applications are text editors, drawing packages, video or audio display tools, compiler front-ends, and debuggers. The integrated toolset environment requires applications to interact by calling on each other to handle user requests. For example, to display video, an editor calls a video display program; or to check a block of completed code, an editor calls a compiler. The following scenario illustrates how Desktop Services Message Set might be implemented as an integrated toolset:

1. **George is working on a compound document using his favorite editor.**
He decides to change the some of the source code text.
2. **George double clicks on the source code text.**
 - a. The Document Editor first determines the text represents source code and then determines what file contains the source code.
 - b. The Document Editor sends an *edit* message request, using the file name as a parameter for the message.

- c. The ToolTalk session manager matches the pattern in the *edit* message to a registered application (in this case, the Source Code Editor), and finds an instance of the application running on George's desktop.

Note – If the ToolTalk session manager does not find a running instance of the application, it checks the statically-defined ptypes and starts an application that best matches the pattern in the message. If none of the ptypes match, it returns failure to the Document Editor application.

- d. The Source Code Editor accepts the *edit* message request.
 - e. The Source Code editor determines that the source code file is under configuration control, and sends a message to check out the file.
 - f. The Source Code Control application accepts the message and creates a read/write copy of the requested file. It then passes the name of the file back to the Source Code Editor.
 - g. The Source Code Editor opens a window that contains the source file.
- 3. George edits the source code text.**

Using the ToolTalk Document and Media Exchange Message Set

The ToolTalk Document and Media Exchange Message Set is very flexible and robust. This section illustrates three applications of the ToolTalk Document and Media Exchange Message Set:

- Integrating multimedia into an authoring application
- Adding multimedia extensions to an existing application
- Extending the *cut and paste* facility of X with a media translation facility

Integrating Multimedia Functionality

Integrating multimedia functionality into an application allows end-users of the application to embed various media types in their documents.

Typically, an icon that represents the media object is embedded in the document. Upon selection of an embedded object, the ToolTalk service automatically invokes an appropriate external media application and the object is played as illustrated in the following scenario.

1. Daniel opens a document that contains multimedia objects.
2. The window shows the document with several icons representing various media types (such as sound, video, and graphics).
3. Daniel double-clicks on the sound icon.
A sound application (called a *player*) is launched and the embedded recording is played.
4. To edit the recording, Daniel clicks once on the icon to select it and uses the third mouse button to bring up an Edit menu.
An editing application is launched and Daniel edits the media object.

Adding Multimedia Extensions to Existing Applications

The ToolTalk Document and Media Exchange Message Set also allows an application to use other multimedia applications to extend its features or capabilities. For example, a calendar manager can be extended to use the audiotool to play a sound file as a reminder of an appointment, as illustrated in the following scenario:

1. Karin opens her calendar manager and sets an appointment.
2. Karin clicks on an audio response button, which causes the soundtool to pop up.
3. Karin records her message; for example, “Bring the report.”

When Karin’s appointment reminder is executed, the calendar manager will start the audiotool and play Karin’s recorded reminder.

Extending the X Cut and Paste Facility

The ToolTalk Document and Media Exchange Message Set can support an extensible, open-ended translation facility. The following scenario illustrates how an extensible multimedia *cut and paste* facility could work:

1. Maria opens two documents that are different media types.
2. Maria selects a portion of *Document A* and cuts the portion using the standard X-windowing *cut* facility.
3. Maria then pastes the cut portion into *Document B*.
 - a. *Document B* negotiates the transfer of the cut data with *Document A*.
 - b. If *Document B* does not understand any of the types offered by *Document B*, it requests a *tagged media type*. *Document B* uses the tagged media type to broadcast a ToolTalk message requesting a translation of the media type to a media type it understands.
 - c. A registered translation utility accepts the request and returns the translated version of the media type to *Document B*.
 - d. The paste of the translated data into *Document B* is performed.

Using the CASE Interoperability Message Sets

The CASE Interoperability Message Sets allow an application to integrate and control other applications without user intervention. This section illustrates several scenarios that show how the CASE Interoperability Message Sets might be used.

User Scenario: Fixing Bugs

This scenario steps through a complete cycle of how to fix a bug for a released application. It begins with receiving the bug report and describes the process required to fix the problem.

- 1. Larry receives a bug report that indicates his application has a problem.**
- 2. Larry invokes his CASE environment.**

A CASE user interface is displayed. The functions Larry wants to perform are available in this CASE user interface.

-
- 3. Larry writes a test case that duplicates the failure stated in the bug report.**
 - 4. Larry selects the debug function to run the application against the test case.**
 - a. A Debug request is sent.
 - b. The messaging environment selects the debugging application. It does not find an instance of the application running in Larry's CASE environment, so it automatically starts a debugger.
 - c. The debugging application receives the request and loads the binary.
 - 5. Larry tests the code and reviews the debugging state in the debugging window.**

He finds a function call that appears to be passing incorrect arguments.
 - 6. Larry selects the edit function to edit this code.**
 - a. The debugging tool sends an edit request.
 - b. The source editor receives a message to edit the specified source file.
 - 7. Larry wants to modify the source code, so he selects the checkout function.**
 - a. The source code editor sends a checkout request.
 - b. The source code editor receives the checkout notification, and changes the buffer state to modifiable.
 - 8. Larry edits the source code to fix the bug and selects the Build function to build the application.**
 - a. The build request is sent.
 - b. The build application receives the build request, and performs the build.
 - c. When the build completes, the build application sends the BuildFinished notice.
 - d. The debugger receives the BuildFinished notice and reloads the newly built application binary.

9. Larry retests the application to confirm the bug fix works.

10. Larry exits the CASE environment.

- a. Quit requests are sent to the source code editor, debugger, version manager, and builder applications.

1.2 How Applications Use ToolTalk Messages

Applications create, send, and receive ToolTalk messages to communicate with other applications. Senders create, fill in, and send a message; the ToolTalk service determines the recipients and delivers the message to the recipients. Recipients retrieve messages, examine the information in the message, and then either discard the message or perform an operation and reply with the results.

Sending ToolTalk Messages

ToolTalk messages are simple structures that contain fields for address, subject, and delivery information. To send a ToolTalk message, an application obtains an empty message, fills in the message attributes, and sends the message. The sending application needs to provide the following information:

- Is the message a notice or a request? (that is, should the recipient respond to the message?)
- What interest does the recipient share with the sender? (for example, is the recipient running in a specific user session or interested in a specific file?)

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

Message Patterns

An important ToolTalk feature is that senders need to know little about the recipients because applications that want to receive messages explicitly state what message they want to receive. This information is registered with the ToolTalk service in the form of *message patterns*.

Applications can provide message patterns to the ToolTalk service at installation time and while the application is running. Message patterns are created similarly to the way a message is created; both use the same type of

information. 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. These message patterns usually match the message protocols that applications have agreed to use. Applications can add more patterns for individual use.

When the ToolTalk service receives a message from a sending application, it compares the information in the message to the register patterns. Once matches have been found, the ToolTalk service delivers copies of the message to all recipients.

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

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 receiving application is not running, the ToolTalk service looks for instructions (provided by the application at installation time) on how to start the application.

The process retrieves the message and examines its contents.

- If the message contains a notice that an operation has been performed, the process reads the information and then discards the message.
- 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 discards the original message.

1.3 ToolTalk Message Distribution

The ToolTalk service provides two methods of addressing messages: *process-oriented messages* and *object-oriented messages*.

Process-Oriented Messages

Process-oriented messages are addressed to processes. Applications that create a process-oriented message address the message to either a specific process or to a particular type of process. Process-oriented messages are a good way for existing applications to begin communication with other applications. Modifications to support process-oriented messages are straightforward and usually take a short time to implement.

Object-Oriented Messages

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.

Note – Programs coded to the ToolTalk object-oriented messaging interface are not portable to CORBA-compliant systems without source changes.

Determining Message Delivery

To determine which groups receive messages, you *scope* your messages. Scoping limits the delivery of messages to a particular session or file.

Sessions

A *session* is a group of processes that have an instance of the ToolTalk message server in common. When a process opens communication with the ToolTalk service, a default session is located (or created if a session does not already

exist) and a *process identifier* (*procid*) is assigned to the process. Default sessions are located either through an environment variable (called process tree sessions) or through the X display (called X sessions).

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 reference 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 book.

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 reference the file without regard to the process's default session. To update message patterns with the current file path name, applications join the file.

You can also scope a message to a file within a session. The ToolTalk service will deliver the message to all processes that reference both the file and session in their message patterns.

Note – The file scoping feature is restricted to NFS and UFS file systems; it does not work, for example, across tmpfs filesystems.

1.4 *Modifying Applications to Use the ToolTalk Service*

Before you modify your application to use the ToolTalk service you must define (or locate) a ToolTalk *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.

To use the ToolTalk service, an application calls ToolTalk functions from the ToolTalk application programming interface (API). The ToolTalk API provides functions to register with the ToolTalk service, to create message patterns, to send messages, to receive messages, to examine message information, and so on. To modify your application to use the ToolTalk service, you must first include the ToolTalk API header file in your program. You also need to modify your application to:

- Initialize the ToolTalk service and join a session.
- Register message patterns with the ToolTalk service.
- Send and receive messages.
- Unregister message patterns and leave your ToolTalk session.

An Overview of the ToolTalk Service



As computer users increasingly demand that independently developed applications work together, inter-operability is becoming an important theme for software developers. By cooperatively using each other's facilities, inter-operating applications offer users capabilities that would be difficult to provide in a single application. The ToolTalk service is designed to facilitate the development of inter-operating applications that serve individuals and work groups.

2.1 ToolTalk Architecture

The following ToolTalk service components work together to provide inter-application communication and object information management:

- `ttsession` is the ToolTalk communication process.
This process joins together senders and receivers that are either using the same X server or interested in the same file. One `ttsession` 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 which 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 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 an XDR format file, which is referred to as the ToolTalk Types Database in this manual.

Figure 2-1 illustrates the ToolTalk service architecture.

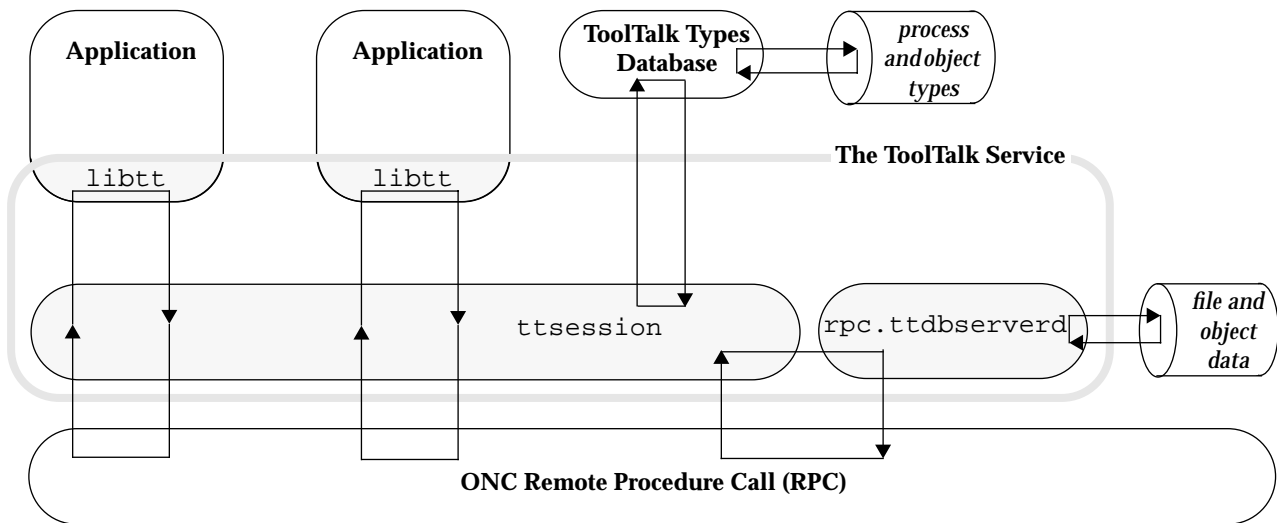


Figure 2-1 ToolTalk Service Architecture

2.2 Starting a ToolTalk Session

The ToolTalk message server `ttsession` automatically starts when you open communication with the ToolTalk server. This background process must be running before any messages can be sent or received. Each message server defines a session.

Note – A session can have more than one session identifier.

To manually start a session, enter the following command on the command line:

```
ttsession [-a level][-d display][-spStvh][-{E|X}][-c command]
```

See Table 2-1 for a description of the `ttsession` command line options.

Table 2-1 `ttsession` Command Line Options

Argument	Description
<code>-a level</code>	Sets the server authentication level. The level must be <i>unix</i> , <i>xauth</i> , or <i>des</i> .
<code>-d display</code>	Directs <code>ttsession</code> to start an <i>X</i> session for the given display. Normally, <code>ttsession</code> uses the <code>\$DISPLAY</code> environment variable.
<code>-h</code>	Prints help on how to invoke <code>ttsession</code> and exits.
<code>-p</code>	Starts a new <code>ttsession</code> and prints out its session id.
<code>-S</code>	Directs <code>ttsession</code> <i>not</i> to fork a background instance to manage its session.
<code>-s</code>	Enables silent operation; warning messages are not printed.

Note — If neither the `-c`, `-d`, or `-p` options are specified, `ttsession` starts an *X* session for the display specified in the `$DISPLAY` environment variable.

Table 2-1 `ttsession` Command Line Options (Continued)

Argument	Description
-t	Turns on trace mode. If trace mode is turned on while <code>ttsession</code> is running, messages appear on the console. Use this mode to see how messages are dispatched and delivered. Trace mode displays the state of a message when it is first seen by <code>ttsession</code> . It also displays any attempt to send the message to a given process with the success of that attempt. To toggle the trace mode on or off, use the <code>USR1</code> signal.
-v	Prints the version number and exits.
-E	Reads in the types from the Classing Engine database. This option is disabled. (The ToolTalk service now uses the XDR format databases.)
-X	Reads in the types from the XDR format database in <code>\$TTHOME/.tt/types.xdr</code> and <code>/etc/tt/types.xdr</code> . This option is the default.
-c <i>command</i>	Starts a process tree session and runs the given command. The special environment variable <code>_SUN_TT_SESSION</code> will be set to the name of this session. Any process that was started with this environment variable will default to be in this session. This option must be the last option on the command line; any characters placed after the <code>-c</code> option on the command line are taken as the command to be executed. If <i>command</i> is omitted, the value of <code>\$SHELL</code> is used instead.

Note — If neither the `-c`, `-d`, or `-p` options are specified, `ttsession` starts an `X` session for the display specified in the `$DISPLAY` environment variable.

`ttsession` responds to two signals.

- If it receives the `SIGUSR1` signal, it toggles the trace mode on or off.
- If it receives the `SIGUSR2` signal, it rereads the types file.

Background and Batch Sessions

Run your application as its own session if it runs as a background job, in a batch session, or in a session bound to a character terminal. To run your application in its own session, use the `-c` parameter with the `ttsession` command, as follows:

```
ttsession -c [ command-to-run-in-batch ]
```

This command will fork off a shell from which you can run your application.

Note – The `-c` parameter must be the last option on the command line; any characters placed after the `-c` parameter on the command line are taken as the command to be executed.

X Window System

To establish a session under the X Window System, execute `ttsession` either without arguments (which takes the display from the `$DISPLAY` environment variable) or specify the display with the `-d` parameter as follows:

```
ttsession -d :0
```

When `ttsession` is invoked, it immediately forks and the parent copy exits; the process managing the session executes in the background. The session is registered as a property, named by `_SUN_TT_SESSION` on the root window of screen 0; the host and port number is given for communication with the process managing the session.

Locating tsession

To display the `sessid` of the session for the Xdisplay:

```
xprop -root | grep _SUN_TT_SESSION
```

2.3 *Maintaining ToolTalk Files and Databases*

The ToolTalk package contains a special set of shell commands you can use to copy, move, and remove ToolTalk files (that is, files mentioned in messages and files that contain ToolTalk objects). After a standard shell command (such as `cp`, `mv`, or `rm`) is performed, the ToolTalk service is notified that a file location has changed.

The ToolTalk package also contains a database check and repair utility for the ToolTalk database, `ttdbck`, that you can use to check and repair your ToolTalk databases.

2.4 *Demonstration Programs*

The ToolTalk service source files contain several interesting OpenWindows-based demonstration programs:

- `CoEd`
A shared editor that uses ToolTalk messages to keep the shared edit session consistent across multiple editors. This program is OLIT-based.
- `ttsample1`
A simple demonstration of the ToolTalk service sending and receiving messages. This program is XView-based.
- `ttsnoop`
A demonstration program that allows you to create and send custom-constructed ToolTalk messages. You can also use this program to selectively monitor any or all ToolTalk messages on your system. This program is XView-based.
- `edit_demo`
Two programs - `cntl` and `edit` - that demonstrate ToolTalk object-oriented messaging. This program is XView-based.

This chapter describes how to provide message pattern information to the ToolTalk service. The ToolTalk service uses message patterns to determine message recipients. After receiving a message, the ToolTalk service compares the message to all current message patterns to find a matching pattern. Once a match is made, the message is delivered to the application that registered the message pattern.

You can provide message pattern information to the ToolTalk service using either dynamic or static methods, or both. The method you choose depends on the type of messages you want to receive.

- If the types of messages you want to receive will vary while your application is running, the *dynamic method* allows you to add, change, or remove message pattern information after your application has started.
- If you want a message to start your application or to be queued if your application is not running, the *static method* provides an easy way to specify these instructions. The static method also provides an easy way to specify the message pattern information if you want to receive a defined set of messages. For more information, see Chapter 10, “Static Message Patterns.”

Regardless of the method you choose to provide message patterns to the ToolTalk service, you will want to update these patterns with each current session and file information so that you receive all messages that reference the session or file in which you are interested.

3.1 Message Pattern Attributes

The attributes in your message pattern specify the type of messages you want to receive. Although some attributes are set and have only one value, you can supply multiple values for most of the attributes you add to a pattern.

Table 3-1 provides a complete list of attributes you can put in your message patterns.

Table 3-1 ToolTalk Message Pattern Attributes

Pattern Attribute	Value	Description
Category	TT_OBSERVE TT_HANDLE	Declares whether you want to perform the operation listed in a message or only observe a message.
Scope	TT_SESSION TT_FILE TT_FILE_IN_SESSION TT_BOTH	Declares interest in messages about a session or a file, or both; Join a session or file after the message pattern is registered to update the sessid and filename.
Arguments	arguments or results	Declares the positional arguments for the operation in which you are interested.
Context	<name, value>	Declares the keyword or non-positional arguments for the operation in which you are interested
Class	TT_NOTICE TT_REQUEST	Declares whether you want to receive notices or requests, or both.
File	char *pathname	Declares the files in which you are interested. If the scope of the pattern does not require a file, the file is an attribute only.
Object	char *objid	Declares what objects in which you are interested.
Operation	char *opname	Declares the operations in which you are interested.
Otype	char *otype	Declares the type of objects in which you are interested.
address	TT_PROCEDURE TT_OBJECT TT_HANDLER TT_OTYPE	Declares the type of address in which you are interested.
disposition	TT_DISCARD TT_QUEUE TT_START TT_START+TT_QUEUE	Instructs the ToolTalk service how to handle messages to your application if an instance is not currently running.

Table 3-1 ToolTalk Message Pattern Attributes (Continued)

Pattern Attribute	Value	Description
sender	char *procid	Declares the sender in which you are interested.
sender_ptype	char *ptype	Declares the type of sending process in which you are interested.
session	char *sessid	Declares the session in which you are interested.
state	TT_CREATED TT_SENT TT_HANDLED TT_FAILED TT_QUEUED TT_STARTED TT_REJECTED	Declares the state of the message in which you are interested.

All your message patterns must at least specify:

- **Category** — Whether the application wants to perform operations listed in messages or only view messages.
 - Use TT_OBSERVE if you only want to observe messages.
 - Use TT_HANDLE if you want to perform operations requested by the messages.
- **Scope** — Whether the application is interested in messages about a particular session or file.
 - Use TT_SESSION to receive messages from other processes in your session.
 - Use TT_FILE to receive messages about the file you have joined.
 - Use TT_FILE_IN_SESSION to receive messages for the file you have joined while in this session.
 - Use TT_BOTH to receive both messages for the file, the session, or the file and the session you have joined.

The ToolTalk service compares message attributes to pattern attributes as follows:

- The ToolTalk service counts the message attribute as matched if:
 - No pattern attribute is specified.
 - The pattern does not name a context slot.

- The pattern has an empty context slot.

The fewer pattern attributes you specify, the more messages you become eligible to receive.

- If there are multiple values specified for a pattern attribute, one of the values must match the message attribute value. If no value matches, the ToolTalk service will not consider your application as a receiver.
- If context slots are contained in the message, the ToolTalk service will not consider your application as a receiver unless:
 - A value specified in a context slot of a pattern matches the value specified in the message context slot.
 - When multiple context slots are specified in a message, each context slot value in the message matches a corresponding context slot value in the pattern.

3.2 *Scope Attributes*

You can specify the following types of scopes in your message patterns:

1. Scope to a session only.
2. Scope to a file only.
3. Scope only to a file in a particular session.
4. Scope to either or both a file and a session.

Note – File scopes are restricted to NFS and UFS file systems; you cannot scope a file across other types of file systems (for example, a tmpfs file system).

Scoping to a Session Only

The type `TT_SESSION` scopes to a session only. Static session-scoped patterns require an explicit `tt_session_join` call to set the scope value; dynamic session-scoped patterns can be set with either the `tt_session_join` call or the `tt_pattern_session_add` call.

Note – The session specified by these calls must be the default session.

Code Example 3-1 shows a static session-scoped pattern; Code Example 3-2 shows a dynamic session-scoped pattern.

Code Example 3-1 Static Session-Scoped Pattern

Obtain procid	<code>tt_open();</code>
Ptype is scoped to session	<code>tt_ptype_declare(ptype);</code>
Join session	<code>tt_session_join(tt_default_session());</code>

Code Example 3-2 Dynamic Session-Scoped Pattern with a File Attribute

Obtain procid	<code>tt_open();</code>
Create pattern	<code>Tt_pattern pat = tt_create_pattern();</code>
Add scope to pattern	<code>tt_pattern_scope_add(pat, TT_SESSION);</code>
Add session to pattern	<code>tt_pattern_session_add (tt_default_session());</code>
Register pattern	<code>tt_pattern_register(pat);</code>

Scoping to a File Only

The type `TT_FILE` scopes to a file only. Code Example 3-3 shows a static file-scoped pattern; Code Example 3-4 shows a dynamic file-scoped pattern.

Code Example 3-3 Static File-Scoped Pattern

Obtain procid	<code>tt_open();</code>
Ptype is scoped to file	<code>tt_ptype_declare(ptype);</code>
Join file	<code>tt_file_join(file);</code>

Code Example 3-4 Dynamic File-Scoped Pattern

Obtain procid	<code>tt_open();</code>
Create pattern	<code>Tt_pattern pat = tt_create_pattern();</code>
Add scope to pattern	<code>tt_pattern_scope_add(pat, TT_FILE);</code>
Add file to pattern	<code>tt_pattern_file_add(pat, file);</code>
Register pattern	<code>tt_pattern_register(pat);</code>

Scoping to a File in a Session

The type `TT_FILE_IN_SESSION` scopes to the specified file in the specified session only. A pattern with this scope set will only match messages that are scoped to both the file and the session. Code Example 3-5 adds the session and then registers the pattern.

Code Example 3-5 Adding a Session to the `TT_FILE_IN_SESSION`-Scoped Pattern

Obtain procid	<code>tt_open();</code>
Create pattern	<code>Tt_pattern pat = tt_create_pattern();</code>
Add scope to pattern	<code>tt_pattern_scope_add(pat, TT_FILE_IN_SESSION);</code>
Add file to pattern	<code>tt_pattern_file_add(pat, file);</code>
Add session to pattern	<code>tt_pattern_session_add(pat, tt_default_session());</code>
Register pattern	<code>tt_pattern_register(pat);</code>

Code Example 3-6 registers the pattern and then joins a session.

Code Example 3-6 Joining a Session to Set the Session of a TT_FILE_IN_SESSION-Scoped Pattern

Obtain procid	<code>tt_open();</code>
Create pattern	<code>Tt_pattern pat = tt_create_pattern();</code>
Add scope to pattern	<code>tt_pattern_scope_add(pat, TT_FILE_IN_SESSION);</code>
Add file to pattern	<code>tt_pattern_file_add(pat, file);</code>
Register pattern	<code>tt_pattern_register(pat);</code>
Join session	<code>tt_session_join(tt_default_session());</code>

Code Example 3-7 sets the scope value for a static pattern.

Code Example 3-7 Setting the Scope Value for a TT_FILE_IN_SESSION Static Pattern

Obtain procid	<code>tt_open();</code>
Declare Ptype	<code>Tt_ptype_declare(ptype);</code>
Join File	<code>tt_file_join(file);</code>
Join session	<code>tt_session_join(tt_default_session());</code>

Scoping to a File and/or a Session

A TT_BOTH-scoped pattern will match messages that are scoped to the file, the session, or the file and the session. However, when you use this scope, you must explicitly make a `tt_file_join` call; otherwise, the ToolTalk service will only match messages that are scoped to both the file and session of the registered pattern. Code Example 3-8 and Code Example 3-9 show examples of how to use this scope.

Code Example 3-8 A Dynamic Pattern that Uses the TT_BOTH Scope

Obtain procid	<code>tt_open();</code>
Create pattern	<code>Tt_pattern pat = tt_create_pattern();</code>
Add scope to pattern	<code>tt_pattern_scope_add(pat, TT_BOTH);</code>
Add session to pattern	<code>tt_pattern_session_add(pat, tt_default_session());</code>
Add file to pattern	<code>tt_pattern_file_add(pat, file);</code>
Register pattern	<code>tt_pattern_register(pat);</code>

Code Example 3-9 A Static Pattern that Uses the TT_BOTH Scope

Obtain procid	<code>tt_open();</code>
Declare Ptype	<code>Tt_ptype_declare(ptype);</code>
Join file	<code>tt_file_join(file);</code>
Join session	<code>tt_session_join(tt_default_session());</code>

Adding Files to Scoped Patterns

To match TT_SESSION-scoped messages and TT_SESSION-scoped patterns that have the same file attributes, you can add file attributes to TT_SESSION-scoped patterns with the `tt_pattern_file_add` call, as shown in Code Example 3-10.

Note - The file attribute values do not affect the scope of the pattern.

Code Example 3-10 Adding Two File Attributes to a Session-Scoped Pattern

Obtain procid	<code>tt_open();</code>
Create pattern	<code>Tt_pattern pat = tt_create_pattern();</code>
Add scope to pattern	<code>tt_pattern_scope_add(pat, TT_SESSION);</code>
Add session to pattern	<code>tt_pattern_session_add(tt_default_session());</code>
Add first file attribute to pattern	<code>tt_pattern_file_add(pat, file1);</code>
Add second file attribute to pattern	<code>tt_pattern_file_add(pat, file2);</code>
Register pattern	<code>tt_pattern_register(pat);</code>

3.3 Context Attributes

ToolTalk *contexts* are sets of <name, value> pairs explicitly included in both messages and patterns. ToolTalk contexts allow fine-grain matching.

You can use contexts to associate arbitrary pairs with ToolTalk messages and patterns, and to restrict the set of possible recipients of a message. One common use of the restricted pattern matching provided by ToolTalk context attributes is to create sub-sessions. For example, two different programs could be debugged simultaneously with tools such as a browser, an editor, a debugger, and a configuration manager active for each program. The message and pattern context slots for each set of tools contain different values; the normal ToolTalk pattern matching of these values keep the two sub-sessions separate.

Another use for the restricted pattern matching provided by ToolTalk context attributes is to provide information in environment variables and command line arguments to tools started by the ToolTalk service.

3.4 *Disposition Attributes*

Disposition attributes instruct the ToolTalk service how to handle messages to your application if an instance of the application is not currently running.

The disposition value specified in the static type definition of a pattern is the default disposition; however, if the message disposition specifies the handler ptype the default disposition value is over-ridden. For example, a message disposition specifies a static type definition for the ptype *UWriteIt* which includes the message signature *Display*. This message signature does not match any of the static signatures in the pattern. The ToolTalk service will follow the instructions for the disposition set in the message; for example, if the message disposition is `TT_START` and the *UWriteIt* ptype specifies a start string, the ToolTalk service will start an instance of the application if one is not running.

Setting Up and Maintaining the ToolTalk Processes



Note – The ToolTalk database server program must be installed on all machines which store files that contain ToolTalk objects or files that are the subject of ToolTalk messages.

4.1 Location of the ToolTalk Service Files

The ToolTalk directories and files are:

```
/tt/bin/  
  ttsession  
  rpc.ttdbserverd
```

- `ttsession` communicates with other `ttsessions` on the network to deliver messages. `rpc.ttdbserverd` stores and manages ToolTalk object specs and information on files referenced in ToolTalk messages.

```
/tt/bin/  
  ttcp  
  ttmv  
  ttrm  
  ttrmdir  
  tttar
```

- These commands are standard operating system shell commands that inform the ToolTalk service when files that contain ToolTalk objects or files that are the subject of ToolTalk messages are copied, moved, or removed.

```
/tt/bin/  
  ttdbck
```

- **ttdbck** is a database check and recovery tool for the ToolTalk databases.

```
/tt/bin/  
  tt_type_comp
```

- **tt_type_comp** is a compiler for ptypes and otypes. It compiles the ptype and otype files and automatically installs them in the ToolTalk Types database.

```
/tt/bin/  
  ttce2xdr
```

- **ttce2xdr** converts ToolTalk type data from the Classing Engine database format to the XDR-database format.

```
/tt/lib  
  libtt.so.1  
  libtt.a
```

```
/tt/share/include/desktop/  
  tt_c.h
```

- These files are the application programming interface (API) libraries and header file that contain the ToolTalk functions used by applications to send and receive messages.

```
/tt/man/man1/  
  install_tt.1  
  tt_type_comp.1  
  ttcp.1  
  ttmv.1  
  ttrm.1  
  ttrmdir.1  
  ttsession.1  
  tttar.1
```

```
/tt/man/man1m/  
  rpc.ttdbserverd.1m  
  ttce2xdr.1m  
  ttdbck.1m  
  ttdbserverd.1m
```

```
/tt/man/man3/  
  ttapi.3
```

```
/tt/demo/  
  CoEd.6  
  edit.demo.6  
  ttsample1.6  
  ttsnoop.6
```

- These are the man pages for the ToolTalk binary files, type compiler, enhanced shell commands, API, and database check utility.

4.2 Version

All ToolTalk commands except the `ttce2xdr` command support a `-v` option that prints the version string.

4.3 Requirements

The software required by the ToolTalk service includes ONC RPC.

4.4 Environment Variables

ToolTalk Environment Variables

There are several ToolTalk environment variables that may be set:

- `SUN_TTSESSION_CMD`
- `_SUN_TT_ARG_TRACE_WIDTH`
- `_SUN_TT_FILE`
- `_SUN_TT_HOSTNAME_MAP`
- `_SUN_TT_PARTITION_MAP`
- `_SUN_TT_SESSION`
- `_SUN_TT_TOKEN`
- `TTPATH`
- `DISPLAY`
- `CEPATH`

Table 4-1 describes these variables.

Table 4-1 Environment Variables

Variable	Description
SUN_TTSESSION_CMD	Overrides the standard options specified when tools automatically start ttsession. If this variable is set, all ToolTalk clients use this command to automatically start their X sessions.
_SUN_TT_ARG_TRACE_WIDTH	Defines the number of characters of argument and context values to print when in trace mode. The default is to print the first 40 characters.
_SUN_TT_FILE	ttsession places a pathname in this variable when a tool is invoked by a message scoped to the defined file.
_SUN_TT_HOSTNAME_MAP	Points to a map file. The defined map file is read into the ToolTalk client for redirecting host machines.
_SUN_TT_PARTITION_MAP	Points to a map file. The defined map file is read into the ToolTalk client for redirecting file partitions.
_SUN_TT_SESSION	ttsession communicates its session identifier to the tools that it starts. If this variable is set, the ToolTalk client library uses its value as the default session identifier. NOTE: The string stored in this variable can be passed to tt_default_session_set.
_SUN_TT_TOKEN	Notifies the ToolTalk client library that it has been started by ttsession; the client can then confirm to ttsession that the start was successful.
TTPATH	Tells the ToolTalk service where the ToolTalk Types databases reside. The format of this variable is: userDB[:systemDB[:networkDB]]
CEPATH	Tells the Classing Engine where the ToolTalk Types databases reside.
DISPLAY	Causes ttsession to communicate its session identifier to the tools that it starts if the _SUN_TT_SESSION variable is not set. If the DISPLAY variable is set, the ToolTalk client library uses its value as the default session identifier.

A process is given a modified environment when it is automatically started by the ToolTalk service. The modified environment includes the environment variables `$_SUN_TT_SESSION`, `$_SUN_TT_TOKEN`, and any contexts in the start-message whose keyword begins with the dollar sign symbol (`$`). Optionally, the environment variable `$_SUN_TT_FILE` may also be included in the modified environment if it is a file-scoped message.

Note – If the `tt_open` call will be invoked by a child process, the parent process must propagate the modified environment to that child process.

Other Environment Variables

The `TMPDIR` environment variable is another environment variable that you can set to manipulate the ToolTalk development environment; for example:

```
TMPDIR=/var/tmp
```

redirects files to the `/var/tmp` directory.

Environment Variables Required to Start Programs on Remote Hosts

The start string is always executed on the host on which `ttsession` is running; however, the executed process can start another process on another host.

To do this, first make your start string be similar to the following:

```
# rsh farhost myprog
```

Next, to make sure `myprog` is placed in the right session and receives its initial message, you need to propagate the important ToolTalk environment variables. The `tt_rsh` shell script shown in Code Example 4-1 propagates these environment variables.

Code Example 4-1 Propagating ToolTalk Environment Variables

```
#!/bin/sh
# This command runs a command remotely in the background, by pointing
# stdout and stderr at /dev/null.
# By running this through the Bourne shell at the other end, and
# we get rid of the rsh and rshd which otherwise hand around at either
# end.
# Hacked from a posting by clyde@emx.utexas.edu to list.xpert
#
```

Code Example 4-1 Propagating ToolTalk Environment Variables (Continued)

```
#set -x
user=
debug=
HOST=${HOST-`hostname`}
if [ "$1" = "-debug" ]; then
    debug=1
    shift
fi
if [ $# -lt 2 -o "$1" = "-h" -o "$1" = "-help" ]; then
    echo "Usage: ttrsh [-debug] remotehost [-l username] remotecommand"
    echo "Usage: ttrsh [-h | -help]"
    exit 1
else
    host=$1
    shift
    if test "$1" = "-l" ; then
        shift
        user=$1
        shift
    fi
fi
xhostname=`expr "$DISPLAY" : "\([^:]*\).*"`
xscreen=`expr "$DISPLAY" : "[^:]*\(.*)"`
if test x$xscreen = x; then
    xscreen=":0.0"
fi
if test x$xhostname = x -o x$xhostname = x"unix"; then
    DISPLAY=$HOST$xscreen
fi
if [ "$user" = "" ]; then
    userOption=""
else
    userOption="-l $user"
fi
if [ $debug ]; then
    outputRedirect=
else
    outputRedirect='> /dev/null 2>&1 &'
fi
(
    echo "OPENWINHOME=$OPENWINHOME;export OPENWINHOME;\
_SUN_TT_SESSION=$_SUN_TT_SESSION;export _SUN_TT_SESSION;\
```

Code Example 4-1 Propagating ToolTalk Environment Variables (Continued)

```
_SUN_TT_TOKEN=$_SUN_TT_TOKEN;export _SUN_TT_TOKEN;_SUN_TT_FILE=$_SUN_TT_FILE;\
export _SUN_TT_FILE;DISPLAY=$DISPLAY;export DISPLAY;($*)" \
$outputRedirect | rsh $host $userOption /bin/sh &
) &
```

Using Context Slots to Create Environment Variables

Message contexts have a special meaning when the ToolTalk service starts an application. If the name of a context slot begins with a dollar sign (\$), the ToolTalk service interprets the value as an environment variable. For example,

```
start "my_application $CON1"
```

uses the value of context slot \$CON1.

4.5 Installing the ToolTalk Database Server

The ToolTalk Database server is used to store three types of information:

1. ToolTalk objects specs.
2. ToolTalk session IDs of sessions with clients that have joined a file using the `tt_file_join` call.
3. File-scoped messages that are queued because the message disposition is `TT_QUEUED` and a handler that can handle the message has not yet been started.

Note – The ToolTalk database server does *not* store messages that are scoped to file-in-session.

The ToolTalk service requires that a database server run on each machine that stores files that contain ToolTalk objects or files that are the subject of ToolTalk messages. When an application attempts to reference a file on a machine that does not contain a database server, an error similar to the following message is displayed:

```
% Error: Tool Talk database server on integral is not running: tcp
```

where *integral* is the hostname and *tcp* is the application protocol. This error message indicates that the connection failed. A failed connection can also be caused by network problems.

When the ToolTalk Service is Installed Elsewhere on the System

To install the ToolTalk database server from another machine on the system that already has the ToolTalk service installed:

1. **Login as superuser.**
2. **Copy the database server image from a machine that has the ToolTalk software installed to the machine that does not.**

For example:

```
# rcp client:/tt/bin/rpc.ttdbserverd /tt/bin/rpc.ttdbserverd
```

where *client* is the machine where the ToolTalk binaries are located.



Caution – Before you copy the database server image, verify that the operating systems are the same levels.

3. **Check that the `/etc/inetd.conf` file contains the following line.**

```
# 100083/1 stream rpc/tcp wait root /tt/rpc.ttdbserverd rpc.ttdbserverd
```

4. **Cause `inetd` to reread the configuration file.**
(*inetd-pid* in the following example is from the `ps` listing.)

```
# ps -ef | grep inetd  
# kill -HUP inetd-pid
```


4.6 *Running the New ToolTalk Database Server*

Once the new version of the ToolTalk database server has been run on a machine, you cannot revert to a previous version of the ToolTalk database server. Any attempt to run a previous version of the ToolTalk database server displays the following error message:

```
rpc.ttdbserverd: Any data written using a ToolTalk 1.0.x DB server
after using a new ToolTalk DB server will be ignored.
```

4.7 *Redirecting the ToolTalk Database Server*

You can redirect both database host machines and the file system partitions.

- Redirecting a database host machine allows a ToolTalk client to physically access ToolTalk data from a machine that is not running a ToolTalk database server.
- Redirect a file system partition allows a ToolTalk database to logically read and write ToolTalk data from and to a read-only file system partition (for example, a CD-Rom) by physically accessing a different file system partition.

Redirecting the Host Machine

When you redirect a database host machine, a ToolTalk client can physically access ToolTalk data from a machine that is not running a ToolTalk database server. To redirect the host machine, you need to map the hostnames of the machines the ToolTalk client is to access. On the machine running the ToolTalk client that is making the database query:

1. Create a `hostname_map` file.

For example:

```
# Map first host machine
oldhostname1 newhostname1

# Map second host machine
oldhostname2 newhostname2
```

where *oldhostname* is the name of the machine the ToolTalk client needs to access and *newhostname* is the name of a machine that is running the ToolTalk database server.

2. Store the file in the same location at which the ToolTalk Types databases are stored.

The map files have the same order of precedence as the ToolTalk Types (*usr*, *system*, or *network*) databases.

Note – A file defined in the `_SUN_TT_HOSTNAME_MAP` environment variable has a higher precedence than the map in the *user* database.

The map file is read into a ToolTalk client when the client makes a `tt_open` call.

Redirecting the File System Partition

When you redirect a file system partition, a ToolTalk database can logically read and write ToolTalk data from and to a read-only file system partition by physically accessing a different file system partition. To redirect a file partition, you need to map the partitions to where the ToolTalk database will write. On the machine running the ToolTalk database server:

1. Create a `partition_map` file.

For example:

```
# Map first partition
/cdrom /usr

# Map second partition
/sr0/export/home /export/home
```

maps the read-only partition `/cdrom` to `/usr`, a read-write partition; and maps the read-only partition `/sr0/export/home` to `/export/home`, a read-write partition.

2. Store the map file in the same location at which the system ToolTalk Types databases are stored.

Note – A file partition defined in the `_SUN_TT_PARTITION_MAP` environment variable has a higher precedence than the file partition defined in this map file.

The map file is read when the ToolTalk database server is started, or when the database receives a `USR2` signal.

Maintaining Application Information



Applications that want to receive ToolTalk messages provide information to the ToolTalk service that describes what kind of messages they want to receive. This information, known as message patterns, is provided dynamically either by applications as they run, or through ptype and otype files.

5.1 Installing Application Types

Installing application types is an occasional task; you only need to install type information when an application error condition exists. Ptype and otype files are run through the ToolTalk type compiler at installation time.

tt_type_comp merges the information into the ToolTalk Types Database. The application then tells the ToolTalk service to read the type information in the database.

To install an application's ptype and otype files, follow these steps:

1. Run `tt_type_comp` on your type file.

```
% tt_type_comp <your-file>
```

tt_type_comp runs *your-file* through `cpp`, compiles the type definitions, and merges the information into a ToolTalk Types table. Table 5-1 describes location of the XDR-base format tables; Table 5-2 describes the location of the Classing Engine-base format table.

Table 5-1 XDR-base Format ToolTalk Types Tables

Database	Uses XDR Table
user	~/tt/types.xdr
system	/etc/tt/types.xdr
network	\$TTHOME/etc/tt/types.xdr

Table 5-2 Classing Engine-base Format ToolTalk Types Tables

Database	Uses Classing Engine Table
user	~/cetables/cetables
system	/etc/cetables/cetables
network	\$OPENWINHOME/lib/cetables/cetables

By default, `tt_type_comp` uses the *user* database. To specify another database, use the `-d` option; for example:

```
% tt_type_comp -d user/system/network <your-file>
```

Note - When you run `tt_type_comp` on your `ptype` or `otype` files, it first runs `cpp` on the file and then checks the syntax before it places the data into the ToolTalk Types Database format. If syntax errors are found, a message is displayed that indicates the line number of the `cpp` file. To find the line, enter `cpp -P source-file temp-file` and view the *temp-file* to find the error on the line reported by `tt_type_comp`.

2. Force `ttsession` to reread the ToolTalk Types Database.

To force `ttsession` to reread the ToolTalk Types Database, see the “Updating the ToolTalk Service” section.

5.2 Examining ToolTalk Type Information

You can examine all type information in a specified ToolTalk Types Database, only the ptype information, or only the otype information. To specify the database you want to examine, use the `-d` option and supply the name of the *user*, *system*, or *network* to indicate the desired database. If the `-d` option is not used, `tt_type_comp` will use the *user* database by default.

- ◆ **To examine all the ToolTalk type information in a ToolTalk Types Database, enter the following line**

```
% tt_type_comp -d user/system/network -p
```

The type information will be printed out in source format.

- ◆ **To list all ptypes in a ToolTalk Types Database, enter the following line:**

```
% tt_type_comp -d user/system/network -P
```

The names of the ptypes will be printed out in source format.

- ◆ **To list all otypes in a ToolTalk Types Database, enter the following line:**

```
% tt_type_comp -d user/system/network -O
```

The names of the otypes will be printed out in source format.

5.3 Removing ToolTalk Type Information

You can remove both ptype and otype information from the ToolTalk Types Databases.

- ◆ **Use `tt_type_comp` to remove type information. Enter the following line:**

```
% tt_type_comp -d user/system/network -r type
```

For example, to remove a ptype called *EditDemo* from the ToolTalk Types network database of a sample application, enter the line:

```
% tt_type_comp -d network -r EditDemo
```

After you remove type information, force any running `ttsessions` to reread the ToolTalk Types Database again to bring the ToolTalk service up-to-date. See “Updating the ToolTalk Service” for more information.

5.4 Updating the ToolTalk Service

When you make changes to the ToolTalk Types Database, you must force any ToolTalk sessions that are already running to reread it or the changes will not be in effect. Use `ttsession` to force the ToolTalk service to read the type information.

1. Enter the `ps` command to find the process identifier (*pid*) of the `ttsession` process

```
% ps -ef | grep ttsession
```

2. Enter the `kill` command to send a `SIGUSR2` signal to `ttsession`.

```
% kill -USR2 <ttsession pid>
```

5.5 Process Type Errors

One or both of the following conditions exists if applications report the error:

```
Application is not an installed ptype.
```

1. The ToolTalk service has not been instructed by the application to reread the recently updated type information in the ToolTalk Types Database. See “Updating the ToolTalk Service” for instructions on how to force the ToolTalk service to reread type information from the ToolTalk Types Database.

2. The application's ptypes and otypes have not been compiled and merged into the ToolTalk Types Database. See "Installing Application Types" for instructions on how to compile and merge type information.

5.6 Using *ttsnoop* to Debug Messages and Patterns

ttsnoop is a tool provided to create and send custom-constructed ToolTalk messages. You can also use *ttsnoop* as a tool to selectively monitor any or all ToolTalk messages. The *ttsnoop* program resides in the directory `$OPENWINHOME/share/src/tooltalk/demo/ttsnoop`. To compile the program, enter the following commands on the command line:

```
cd $OPENWINHOME/share/src/tooltalk/demo/ttsnoop
make
```

Note - To run *ttsnoop* as a debugging tool, you must be running the OpenWindows software. Set the `$OPENWINHOME` environment variable to where the OpenWindows software is installed.

To start the program, enter the following command on the command line:

```
ttsnoop [ -t ]
```

The **-t** option displays the ToolTalk API calls that are being used to construct a particular pattern or message. Figure 5-1 shows the window that is displayed when *ttsnoop* starts.

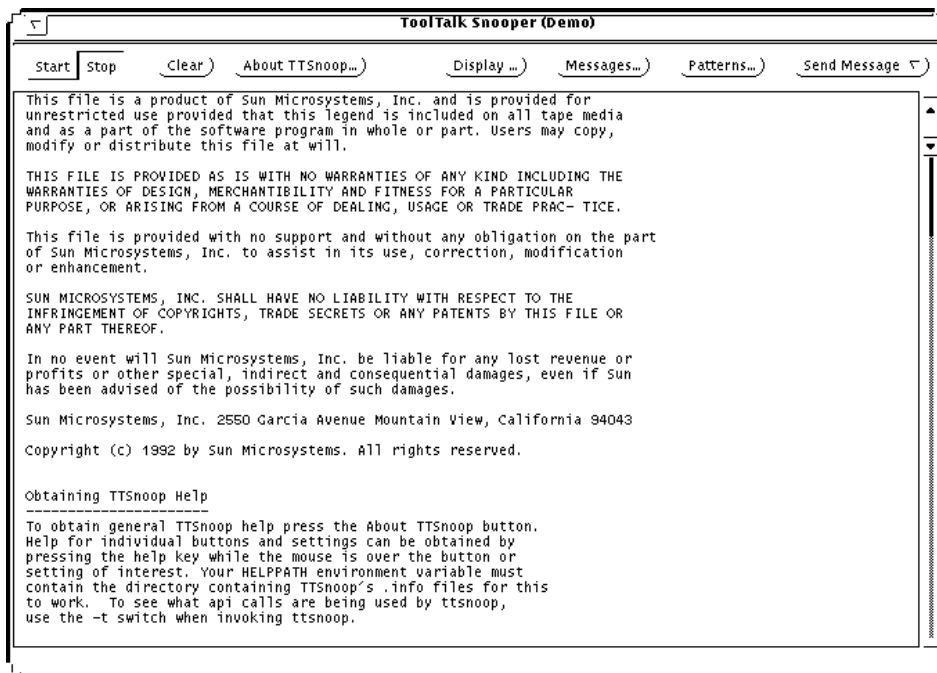


Figure 5-1 ttsnoop Display Window

- ◆ **Start**
Press this button to activate message reception. *ttsnoop* will display any incoming messages which match the patterns you register.
- ◆ **Stop**
Press this button to stop receiving messages.
- ◆ **Clear**
Press this button to clear the window.

◆ About TTSnoop

Press this button to obtain general help for *ttsnoop*.

Note – To obtain help for individual buttons and settings, place the mouse over the button or setting and press the F1 key or Help key on your keyboard. Your `HELPPATH` environment variable must contain the directory that contains the TTSnoop `.info` files.

◆ Display

Press this button to popup a panel of checkboxes to highlight specific ToolTalk message components on the *ttsnoop* display subwindow.

◆ Messages

Press this button to popup a panel which allows you to create, store, and send ToolTalk messages.

◆ Patterns

Press this button to popup a panel which allows you to compose and register ToolTalk patterns.

◆ Send Messages

Press this button to send messages that were stored using the Messages popup.

Composing and Sending Messages

When you press the Messages button on the initial display window, the popup panel shown in Figure 5-2 is displayed.

Send Message

Add Message Change Message Delete Message Edit Send Contexts...

Messages

Description: _____

Address: TT_PROCEDURE TT_OBJECT TT_HANDLER TT_OTYPE

Handler: _____

Handler ptype: _____

Object: _____

Otype: _____

OP: _____

Scope: TT_SESSION TT_FILE TT_BOTH TT_FILE_IN_SESSION

Session: X 129.144.153.7 0

File Name: _____

Class: TT_NOTICE TT_REQUEST

Disposition: TT_DISCARD TT_QUEUE TT_START TT_QUEUE+TT_START

Sender ptype: _____

Arguments

Mode: TT_IN TT_OUT TT_INOUT Kind of Add: Add Badd ladd

Vtype: _____

Value: _____

Add Delete Change

Send Message

Figure 5-2 Popup Messages Panel

◆ Add Message

Press this button to store the current message settings. Once the messages are stored, you can recall and send these messages using the Send Message button on the initial display window.

◆ Edit Contexts

Press this button to add, change, and delete send message contexts. The popup window displayed, shown in Figure 5-3, allows you to edit contexts to be sent with your messages.

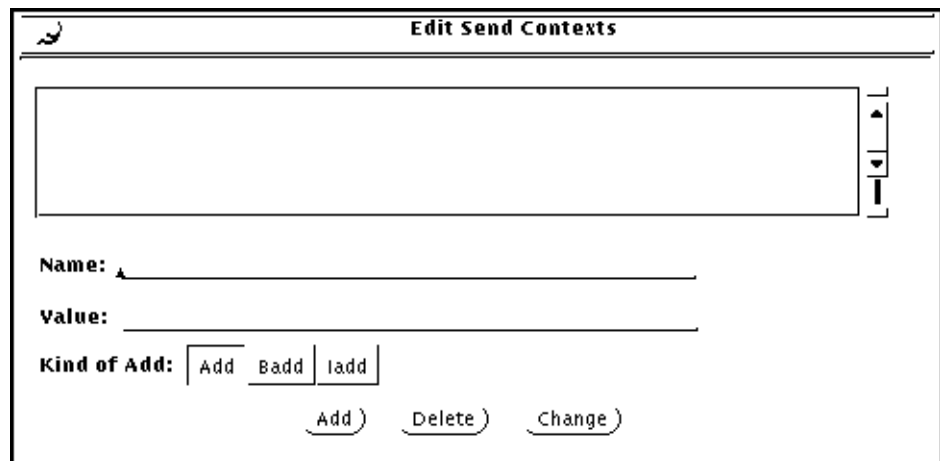


Figure 5-3 Editing Send Contexts

◆ Send Message

Press this button to send the newly created message.

Note – This button performs the same function as the Send Message button on the main menu.

Composing and Registering Patterns

When you press the Patterns button on the initial display window, the popup panel shown in Figure 5-4 is displayed.

Pattern Match

Edit Receive Contexts...)

Address: TT_PROCEDURE TT_OBJECT TT_HANDLER TT_OTYPE

Object:

Otype:

OP:

Scope: TT_SESSION TT_FILE TT_BOTH TT_FILE_IN_SESSION

File Name:

Session: X 129.144.153.7 0

Category: TT_OBSERVE

Class: TT_NOTICE TT_REQUEST

State: TT_CREATED TT_SENT TT_HANDLED TT_FAILED TT_QUEUED TT_STARTED TT_REJECTED

Disposition: TT_DISCARD TT_QUEUE TT_START TT_QUEUE+TT_START

Sender:

Sender ptype:

Arguments

Mode: TT_IN TT_OUT TT_INOUT **Kind of Add:** Add Badd Iadd

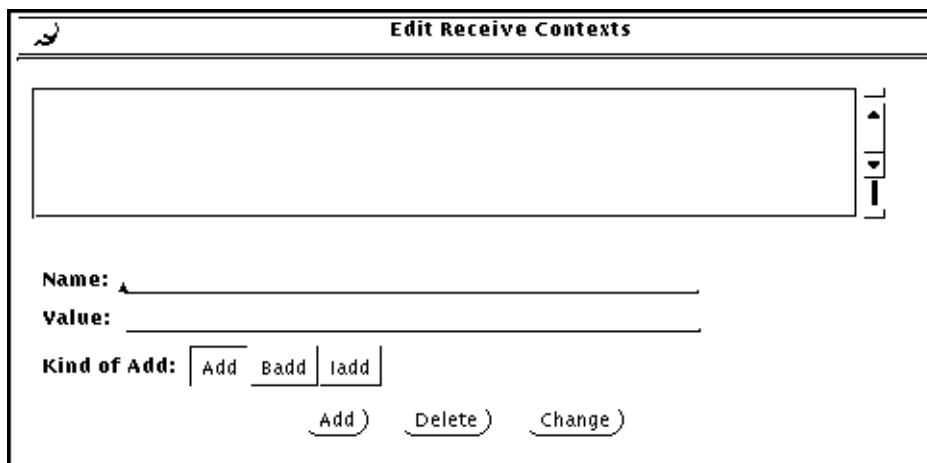
VType: **Value:**

Figure 5-4 Popup Patterns Panel

Press the Apply button to register your pattern. Once a pattern is registered, you can use *ttsnoop* as a debugging tool to observe what messages are being sent by other applications.

◆ **Edit Receive Contexts**

Press this button to add, change, and delete receive message contexts in patterns. The popup window displayed, shown in Figure 5-5, allows you to edit contexts to be registered with your patterns.



Edit Receive Contexts

Name: _____

Value: _____

Kind of Add: Add Badd ladd

Figure 5-5 Editing Message Contexts in Patterns

Displaying Message Components

When you press the Display button on the initial display window, the popup a panel of checkboxes shown in Figure 5-6 is displayed.

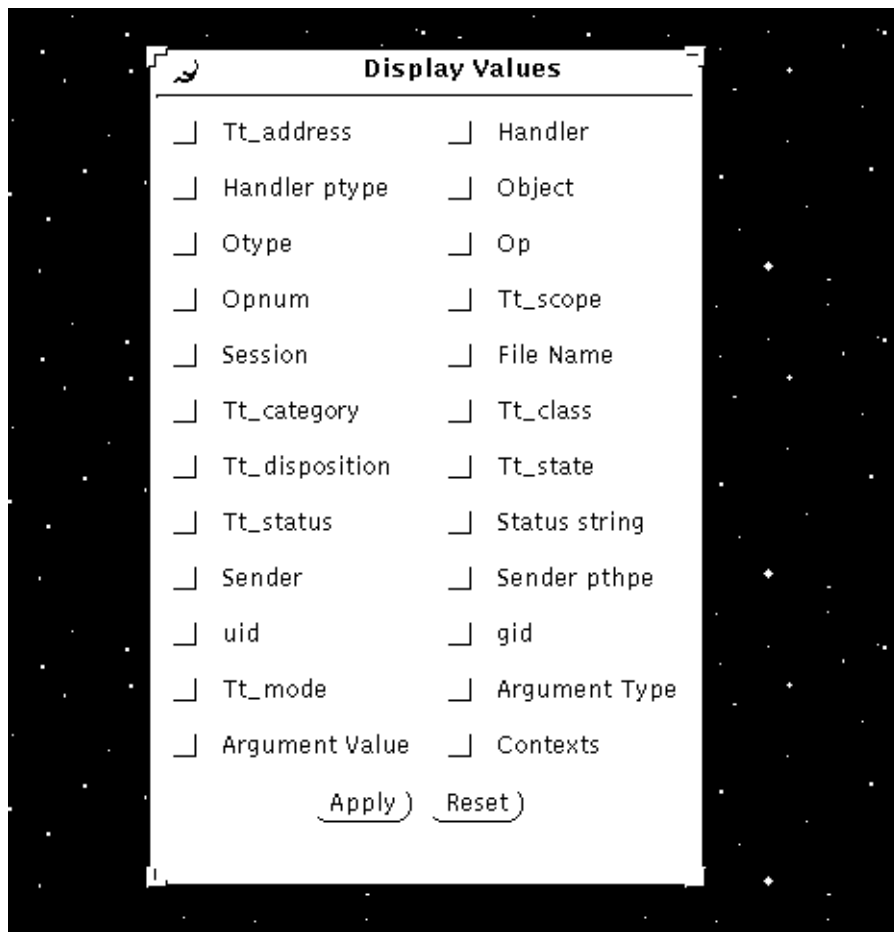


Figure 5-6 Checkboxes to Display Message Component Values

When you select a checkbox, the specified ToolTalk message component is indicated on a displayed message by an arrow (--->) to the left of the displayed message component. Figure 5-7 shows the displayed message components.

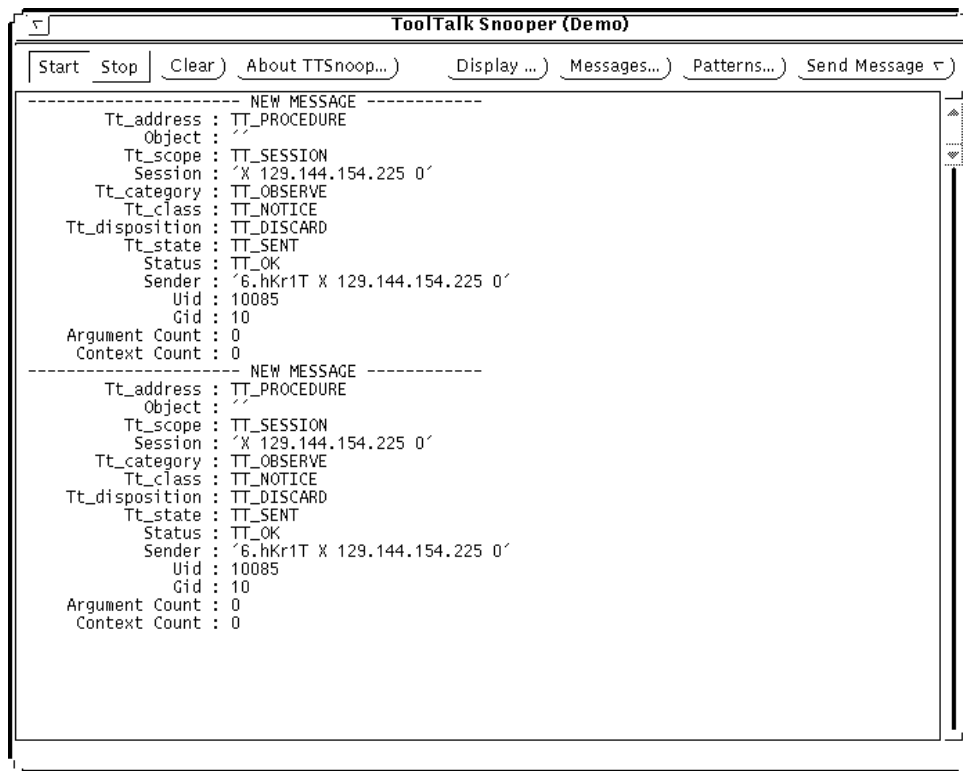


Figure 5-7 Display of Message Components

Sending Pre-Created Messages

When you press the Send Message button on the initial display window, you can send one of the messages you created and stored using the Messages popup.

Receiving Messages

When you press the Start button on the initial display window, *ttsnoop* will display any incoming messages which match the patterns you registered. Figure 5-8 is an example of a displayed incoming message.

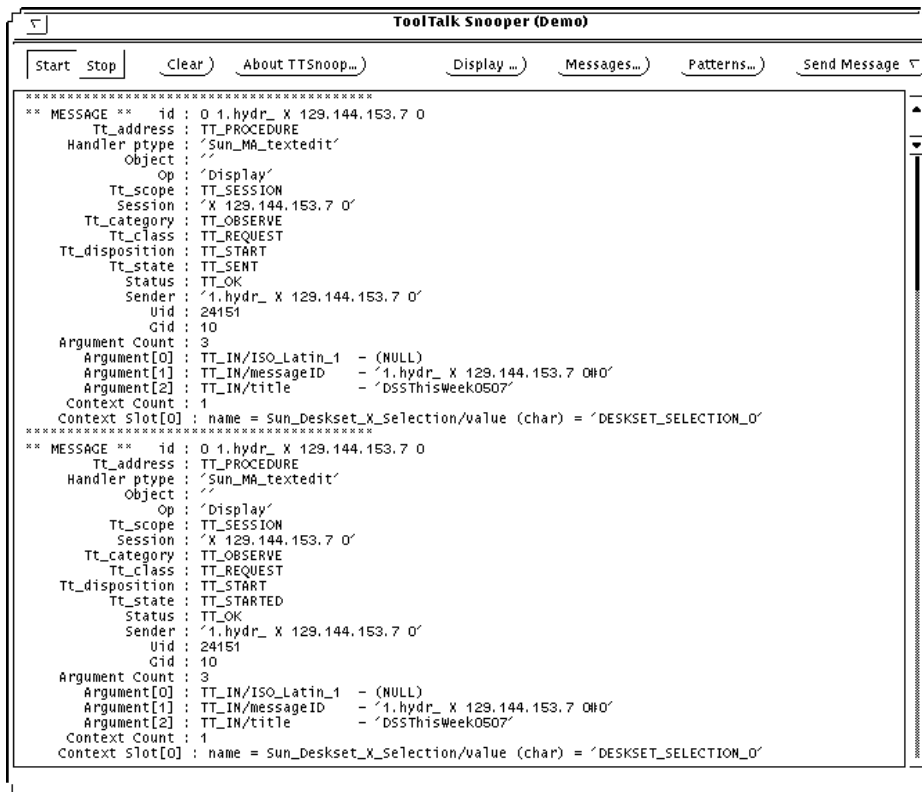


Figure 5-8 Incoming Message Displayed

Stop Receiving Messages

When you press the Stop button on the initial display window, *ttsnoop* will stop receiving messages.

Maintaining Files and Objects Referenced in ToolTalk Messages



ToolTalk messages can reference files of interest or ToolTalk objects. The ToolTalk service maintains information about files and objects, and needs to be informed of changes to these files or objects.

The ToolTalk service provides wrapped shell commands to move, copy, and remove files. These commands inform the ToolTalk service of any changes.

6.1 ToolTalk-Enhanced Shell Commands

The ToolTalk-enhanced shell commands described in Table 6-1 first invoke the standard shell commands with which they are associated (for example, `ttmv` invokes `mv`) and then update the ToolTalk service with the file changes. It is necessary to use the ToolTalk-enhanced shell commands when working with files that contain ToolTalk objects.

Table 6-1 ToolTalk-Enhanced Shell Commands

Command	Definition	Syntax
<code>ttcp</code>	Copies files that contain objects.	<code>ttcp source-file destination-file</code>
<code>ttmv</code>	Renames files that contain objects.	<code>ttmv old new</code>
<code>ttrm</code>	Removes files that contain objects.	<code>ttrm file</code>
<code>ttrmdir</code>	Removes empty directories that are associated with ToolTalk objects. You also use this command to create an object spec for a directory; for example, if a directory is mentioned in a file-scoped message. When an object spec is created, the path name of a file or directory is supplied.	<code>ttrmdir directory</code>
<code>tttar</code>	Archives and de-archives files that contain ToolTalk objects.	<code>tttar c t x pathname1 pathname2</code>

You can cause the ToolTalk-enhanced shell commands to be executed when the standard shell commands are invoked. To do this, alias the ToolTalk-enhanced shell commands in the shell startup file so that the enhanced commands appear as standard shell commands.

```
# ToolTalk-aware shell commands in .cshrc
alias mv      ttmv
alias cp      ttcp
alias rm      ttrm
alias rmdir   ttrmdir
alias tar     tttar
```

6.2 *Maintaining and Updating ToolTalk Databases*

Information about files and objects in the ToolTalk databases can become outdated if the ToolTalk-enhanced shell commands are not used to copy, move, and remove them. For example, you can remove a file *old_file* that contains ToolTalk objects from the file system with the standard `rm` command. However, because the standard shell command does not inform the ToolTalk service that *old_file* has been removed, the information about the file and the individual objects remains in the ToolTalk database.

To remove the file and object information from the ToolTalk database, use the command:

```
ttrm -L old_file
```

6.3 *Displaying, Checking, and Repairing Databases*

Use the ToolTalk database utility `ttdbck` to display, check, or repair ToolTalk databases. You also use the `ttdbck` utility for operations such as:

- Removing all ToolTalk objects of a given otype; for example, an otype that has been de-installed
- Moving specific ToolTalk objects from one file to another
- Searching for all ToolTalk object that reference nonexistent files

Note – ToolTalk databases are typically accessible only to root; therefore, the `ttdbck` utility is normally run as root.

Participating in ToolTalk Sessions



This chapter provides instructions on how to participate in a ToolTalk session. It also shows you how to manage storage of values passed in from the ToolTalk service and how to handle errors that the ToolTalk service returns.

To use the ToolTalk service, your application calls ToolTalk functions from the ToolTalk API library. To modify your application to use the ToolTalk service, you must first include the ToolTalk API header file in your program. After you have initialized the ToolTalk service and joined a session, you can join files and additional user sessions. When your process is ready to quit, you unregister your message patterns and leave your ToolTalk session.

7.1 Including the ToolTalk API Header File

To modify your application to use the ToolTalk service, first you must include the ToolTalk API header file `tt_c.h` in your program. This file resides in the `$TTHOME/include/desktop/` directory.

The following code sample shows how a program includes this file.

```
#include <stdio.h>
#include <sys/param.h>
#include <sys/types.h>

#include <desktop/tt_c.h>
```

7.2 Registering with the ToolTalk Service

Before you can participate in ToolTalk sessions, you must register your process with the ToolTalk service. You can either register in the ToolTalk session in which the application was started (the *initial session*), or locate another session and register there.

The ToolTalk functions you need to register with the ToolTalk service are shown in Table 7-1.

Table 7-1 Registering with the ToolTalk Service

Return Type	ToolTalk Function	Description
char *	tt_open(void)	Process identifier
int	tt_fd(void)	File descriptor
char *	tt_X_session(const char *xdisplay)	Return the session identifier of the specified X display server.
Tt_status	tt_default_session_set(const char *sessid)	Sets the session to which tt_open will connect.

Registering in the Initial Session

To initialize and register your process with the initial ToolTalk session, your application needs to obtain a process identifier (*procid*). You can then obtain the file descriptor (*fd*) that corresponds to the newly initialized ToolTalk process.

The following code sample first initializes and registers the sample program with the ToolTalk service, and then obtains the corresponding file descriptor.

```
int ttfd;
char  *my_procid;

/*
 * Initialize ToolTalk, using the initial default session
 */

my_procid = tt_open();

/*
 * obtain the file descriptor that will become active whenever
 * ToolTalk has a message for this process.
 */

ttfd = tt_fd();
```

`tt_open` returns the `procid` for your process and sets it as the default `procid`; `tt_fd` returns a file descriptor for your current `procid` that will become active when a message arrives for your application.



Caution – Your application must call `tt_open` before other `tt_` calls are made; otherwise, errors may occur. However, there are two exceptions: `tt_default_session_set` and `tt_X_session` can be called before `tt_open`.

When `tt_open` is the first call made to the ToolTalk service, it sets the initial session as the default session. The default session identifier (*sessid*) is important to the delivery of ToolTalk messages. The ToolTalk service automatically fills in the default `sessid` if an application does not explicitly set the session message attribute. If the message is scoped to `TT_SESSION`, the message will be delivered to all applications in the default session that have registered interest in this type of message.

Registering in a Specified Session

To register in a session other than the initial session, your program must find the name of the other session, set the new session as the default, and register with the ToolTalk service.

The following code sample shows how to join an X session named `somehost:0` that is not your initial session.

```
char    *my_session;
char    *my_procid;

my_session = tt_X_session("somehost:0");
tt_default_session_set(my_session);
my_procid = tt_open();
ttfd = tt_fd();
```

Note – The required calls must be in the specified order.

1. `tt_X_session()`;
This call retrieves the name of the session associated with an X display server. `tt_X_session()` takes the argument

```
char *xdisplay_name
```

where `xdisplay_name` is the name of an X display server (in this example, `somehost:0, :0`).
2. `tt_default_session_set()`;
This call sets the new session as the default session.
3. `tt_open()`;
This call returns the `procid` for your process and sets it as the default `procid`.
4. `tt_fd()`;
This call returns a file descriptor for your current `procid`.

Registering in Multiple Sessions

There may be cases when you want to send and receive your messages in different sessions. To register in multiple sessions, your program must find the identifiers of the sessions to which it wants to connect, set the new sessions, and register with the ToolTalk service.

The following code sample shows how to connect *procid* to *sessid1*, and *procid2* to *sessid2*.

```
tt_default_session_set(sessid1);
my_procid1 = tt_open();
tt_default_session_set(sessid2);
my_procid2 = tt_open();
tt_fd2 = tt_fd();
```

You can then use `tt_default_procid_set` to switch between the sessions.

7.3 Setting Up to Receive Messages

Before your application can receive messages from other applications, you must set up your process to watch for arriving messages. When a message arrives for your application, the file descriptor becomes active. The code you use to alert your application that the file descriptor is active depends on how your application is structured.

For example, a program that uses the XView notifier, through the `xv_main_loop` or `notify_start` calls, can have a callback function invoked when the file descriptor becomes active. The following code sample invokes `notify_set_input_func` with the handle for the message object as a parameter.

```
/*
 * Arrange for XView to call receive_tt_message when the
 * ToolTalk file descriptor becomes active.
 */
notify_set_input_func(base_frame,
    (Notify_func)receive_tt_message,
    ttfd);
```

Table 7-2 describes various window toolkits and the call used to watch for arriving messages.

Table 7-2 Code Used to Watch for Arriving Messages

Window Toolkits	Code Used
XView	notify_set_input_func()
X Window System Xt (Intrinsics)	XtAddInput() or XtAddAppInput()
Other toolkits including Xlib structured around <code>select(2)</code> or <code>poll(2)</code> system calls	The file descriptor returned by <code>tt_fd()</code> Note: Once the file descriptor is active and the <code>select</code> call exits, use <code>tt_message_receive()</code> to obtain a handle for the incoming message.

7.4 Sending and Receiving Messages in the Same Process

Normally, the receiver deletes the message when it has completed the requested operation. However, the ToolTalk service uses the same message ID for both the receiver and the requestor. When sending and receiving messages in the same process, these features cause the message underneath the requestor to be deleted as well.

One workaround is to put a refcount on the message. To do this, use the `tt_message_user[_set]()` function.

Another workaround is to destroy the message in the receiver only if the sender is not the current `procid`; for example:

```
Tt_callback_action
my_pattern_callback(Tt_message m, Tt_pattern p)
{
    /* normal message processing goes here */

    if (0!=strcmp(tt_message_sender(m),tt_default_procid()) {
        tt_message_destroy(m);
    }
    return TT_CALLBACK_PROCESSED;
}
```

7.5 Sending and Receiving Messages in a Networked Environment

You can use the ToolTalk service in a networked environment; for example, you can start a tool on a different machine or join a session that is running on a different machine. To do so, invoke a `ttsession` with either the `-c` or `-p` option.

- The `-c` option will invoke the named program and place the right session id in its `_SUN_TT_SESSION` environment variable. For example, the command

```
ttsession -c cmdtool
```

defines `_SUN_TT_SESSION` in that `cmdtool` and any ToolTalk client you run with the environment variable `$_SUN_TT_SESSION` set to its value will join the session owned by this `ttsession`.

- The `-p` option prints the session id to standard output. `ttsession` then forks into the background to run that session.

To join the session, an application must either pass the session id to `tt_default_session_set` or place the session id in the environment variable `_SUN_TT_SESSION` before it calls the `tt_open` function. `tt_open` will check the environment variable `_SUN_TT_SESSION` and join the indicated session (if it has a value).

7.6 Unregistering from the ToolTalk Service

When you want to stop interacting with the ToolTalk service and other ToolTalk session participants, you must *unregister* your process before your application exits.

```
/*
 * Before leaving, allow ToolTalk to clean up.
 */
tt_close();

exit(0);
}
```

`tt_close` returns `Tt_status` and closes the current default `procid`.

Sending Messages



This chapter explains how messages are routed, and describes the ToolTalk message attributes and algorithm. It also describes how to create messages, fill in message contents, attach callbacks to requests, and send messages.

8.1 How the ToolTalk Service Routes Messages

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.

Sending Notices

When you send an informational message, the notice takes a one-way trip, as shown in Figure 8-1.

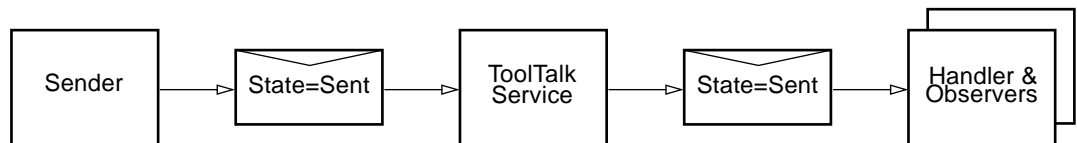


Figure 8-1 Notice Routing

The sending process creates a message, fills in attribute values, and sends it. The ToolTalk service matches message and pattern attribute values, then 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.

Sending Requests

When you send a message that is a request, the request takes a round-trip from sender to handler and back; copies of the message take a one-way trip to interested observers. Figure 8-2 illustrates the request routing procedure.

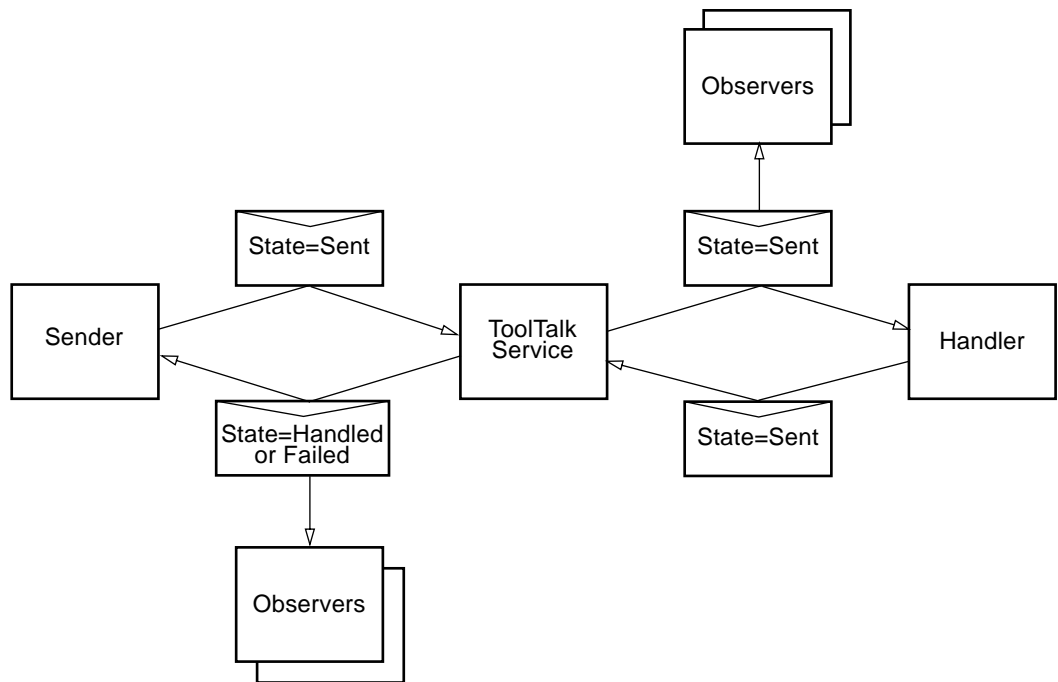


Figure 8-2 Request Routing

The ToolTalk service delivers a request to only one handler. The handler adds results to the message and sends it back. Other processes can observe a request before or after it is handled, or at both times; observers absorb a request without sending it back.

Changes in State of Sent Message

To allow you to track the progress of a request you sent, you will receive a message every time the request changes state. You will receive these state change messages even if no patterns have been registered, or no message callbacks have been specified.

8.2 Message Attributes

ToolTalk messages contain attributes that store message information and provide delivery information to the ToolTalk service. This delivery information is used to route the messages to the appropriate receivers.

ToolTalk messages are simple structures that contain attributes for address, subject (such as *operation* and *arguments*), and delivery information (such as *class* and *scope*.) Each message contains attributes from Table 8-1.

Table 8-1 ToolTalk Message Attributes

Message Attribute	Value	Description	Who Can Complete
Arguments	arguments or results	Specifies arguments used in the operation. If the message is a reply, these arguments contain the results of the operation.	Sender, receiver
Class	TT_NOTICE, TT_REQUEST	Specifies whether the recipient needs to perform an operation.	Sender
File	char *pathname	Specifies the file involved in the operation. If the scope of the message does not require a file, the file is an attribute only.	Sender, ToolTalk
Object	char *objid	Specifies the object involved in the operation.	Sender, ToolTalk
Operation	char *opname	Specifies the name of operation to be performed.	Sender
Otype	char *otype	Specifies the type of object involved in the operation.	Sender, ToolTalk
Address	TT_PROCEDURE, TT_OBJECT, TT_HANDLER, TT_OTYPE	Specifies where the message should be sent.	Sender

Table 8-1 ToolTalk Message Attributes (Continued)

Message Attribute	Value	Description	Who Can Complete
Handler	char *procid	Specifies the receiving process.	Sender, ToolTalk
Handler_ptype	char *ptype	Specifies the type of receiving process.	Sender, ToolTalk
Disposition	TT_DISCARD, TT_QUEUE, TT_START TT_START+TT_QUEUE	Specifies what to do if the message cannot be received by any running process.	Sender, ToolTalk
Scope	TT_SESSION, TT_FILE, TT_BOTH, TT_FILE_IN_SESSION	Specifies the applications that will be considered as potential recipients based on their registered interest in a session or file.	Sender, ToolTalk
Sender_ptype	char *ptype	Specifies the type of the sending process.	Sender, ToolTalk
Session	char *sessid	Specifies the sending process's session.	Sender, ToolTalk
Status	int status, char *status_str	Specifies additional information about the status of the message.	Receiver, ToolTalk

Address Attribute

Messages addressed to other applications can be addressed to a particular process or to any process that has registered a pattern that matches your message. When you address a message to a process, you need to know the process identifier (procid) of the other application. However, processes do not usually know each other's procid; more often, a sender does not care which process performs an operation (request message) or learns of an event (notice message).

Scope Attributes

Applications that use 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. To register this interest, applications join sessions or files (or both) with the ToolTalk service. This file and session information is used by the ToolTalk service with the message patterns to determine which applications should receive a message.

Note – The scope attributes are restricted to NFS and UFS files systems; file scoping does not work across file systems (for example, a tmpfs file system.)

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.

File-based Scoping in Patterns

Table 8-2 describes the types of scopes that use files which you can use to scope messages with patterns.

Table 8-2 Scoping a Message with Patterns to a File

Type of Scope	Description
TT_FILE	Scopes to the specified file only. You can set a session attribute on this type of pattern to provide a file-in-session-like scoping but a <code>tt_session_join</code> call will <i>not</i> update the session attribute of a pattern that is scoped to TT_FILE.
TT_BOTH	Scopes to the <i>union</i> of interest in the file and the session. A pattern with only this scope will match messages that are scoped to the file, or scoped to the session, or scoped to both the file <i>and</i> the session.
TT_FILE_IN_SESSION	Scopes to the <i>intersection</i> of interest in the file and the session. A pattern with only this scope will <i>only</i> match messages that are scoped to both the file and session.

To scope to the union of TT_FILE_IN_SESSION and TT_SESSION, add both scopes to the same pattern, as shown in Code Example 8-1.

Code Example 8-1 Scoping to Union of TT_FILE_IN_SESSION and TT_SESSION

```
tt_open();

Tt_pattern pat = tt_create_pattern();
tt_pattern_scope_add(pat, TT_FILE_IN_SESSION);
tt_pattern_scope_add(pat, TT_SESSION);
```

Code Example 8-1 Scoping to Union of TT_FILE_IN_SESSION and TT_SESSION

```
tt_pattern_file_add(pat, file);
tt_pattern_session_add(pat, tt_default_session());
tt_pattern_register(pat);
```

File-based Scoping in Messages

Messages have the same types of file-based scoping mechanisms as patterns. Table 8-3 describes these scopes.

Table 8-3 Scoping Mechanisms for Messages

Type of Scope	Description
TT_FILE	Scopes the message to all clients that have registered interest in a file.
TT_BOTH	Scopes the message to all clients that have registered interest in the message's session, the message's file, or the message's session and file.
TT_FILE_IN_SESSION	Scopes the message to all clients that have registered interest in both the message's file and session.
TT_SESSION + tt_message_file_set()	Scopes the message to every client that has registered interest in the message's session. When the message is received by a client whose pattern matches, the receiving client can call <code>tt_message_file</code> to get the file name.

When a message is scoped to `TT_FILE` or `TT_BOTH`, the ToolTalk client library checks the database server for all sessions that have clients that are interested in the file and sends the message to all of the interested ToolTalk sessions. The ToolTalk sessions then match the messages to the appropriate clients. The message sender is *not* required to explicitly call to `tt_file_join`.

If a message that is scoped to `TT_FILE_IN_SESSION` or `TT_SESSION` contains a file, the database server is not contacted and the message is sent only to clients that are scoped to the message's session.

Session Scope

When a message is scoped to a session, only those applications that have connected to that session are considered as potential recipients.

Code Example 8-1 Setting a Session

Create message	<code>Tt_message m= tt_message_create();</code>
Add scope to message	<code>tt_message_scope_set(m, TT_SESSION);</code>
Add file attribute that does not affect message scope	<code>tt_message_file_set(m, file);</code>

File-In-Session Scope

Applications can be very specific about the distribution of a message by specifying `TT_FILE_IN_SESSION` for the message scope. Only those applications that have joined both the file and the session indicated are considered potential recipients.

Applications can also scope a message to every client that has registered interest in the message's session by specifying `TT_SESSION` with `tt_message_file_set` for the message scope. When the message is received by a client whose pattern matches, the receiving client can get the file name by calling `tt_message_file`.

Code Example 8-2 Setting a File

Create message	<code>Tt_message m= tt_message_create();</code>
Add scope to message	<code>tt_message_scope_set(m, TT_FILE_IN_SESSION);</code>
Add file to message scope	<code>tt_message_file_set(m, file);</code>

Serialization of Structured Data

The ToolTalk service supports three types of data for message arguments: integers, null-terminated strings, and byte strings.

To send any other data type in a ToolTalk message, the client must serialize the data into a string or byte string and then deserialize it on receipt. The new XDR argument API calls provided with the ToolTalk service now handles these serialization and deserialization functions. The client only needs to provide an XDR routine and a pointer to the data. After serializing the data into the internal buffer, the ToolTalk service treats the data in the same manner as it treats a byte stream.

8.3 *ToolTalk Message Delivery Algorithm*

To help you further understand how the ToolTalk service determines message recipients, this section describes the creation and delivery of both process-oriented messages and object-oriented messages.

Process-Oriented Message Delivery

For some process-oriented messages, the sending application knows the ptype or the procid of the process that should handle the message. For other messages, the ToolTalk service can determine the handler from the operation and arguments of the message.

1. Initialize.

The sender obtains a message handle and fills in the *address*, *scope*, and *class* attributes.

The sender fills in the *operation* and *arguments* attributes.

If the sender has declared only one ptype, the ToolTalk service fills in *sender_ptype* by default; otherwise, the sender must fill it in.

If the scope is `TT_FILE`, the file name must be filled in or defaulted. If the scope is `TT_SESSION`, the session name must be filled in or defaulted. If the scope is `TT_BOTH` or `TT_FILE_IN_SESSION`, both the file name and session name must be filled in or defaulted.

Note – The set of patterns checked for delivery depends on the scope of the message. If the scope is `TT_SESSION`, only patterns for processes in the same session are checked. If the scope is `TT_FILE`, patterns for all processes observing the file are checked. If the scope is `TT_FILE_IN_SESSION` or `TT_BOTH`, both sets of processes are checked.

The sender may fill in the *handler_ptype* if known. However, this greatly reduces flexibility because it does not allow processes of one ptype to substitute for another. Also, the disposition attribute must be specified by the sender in this case.

2. Dispatch to handler.

The ToolTalk service compares the *address*, *scope*, *message class*, *operation*, and *argument* modes and types to all signatures in the Handle section of each ptype.

Only one ptype will usually contain a message pattern that matches the operation and arguments and specifies a handle. If a handler ptype is found, then the ToolTalk service fills in *opnum*, *handler_ptype*, and *disposition* from the ptype message pattern.

If the address is `TT_HANDLER`, the ToolTalk service looks for the specified procid and adds the message to the handler's message queue. `TT_HANDLER` messages cannot be observed because no pattern matching is done.

3. Dispatch to observers.

The ToolTalk service compares the *scope*, *class*, *operation*, and *argument* types to all message patterns in the Observe section of each ptype.

For all observe signatures that match the message and specify `TT_QUEUE` or `TT_START`, the ToolTalk service attaches a record (called an "observe promise") to the message that specifies the ptype and the queue or start options. The ToolTalk service then adds the ptype to its internal `ObserverPtypeList`.

4. Deliver to handler.

If a running process has a registered handler message pattern that matches the message, the ToolTalk service delivers the message to the process; otherwise, the ToolTalk service honors the disposition (start or queue) options.

If more than one process has registered a dynamic pattern that matches the handler information, the more specific pattern (determined by counting the number of non-wildcard matches) is given preference. If two patterns are equally specific, the choice of handler is arbitrary.

5. Deliver to observers.

The ToolTalk service delivers the message to all running processes that have registered Observer patterns that match the message. As each delivery is made, the ToolTalk service checks off any observe promise for the ptype of the observer. After this process is completed and there are observe promises left unfulfilled, the ToolTalk service honors the start and queue options in the promises.

Example

In this example, a debugger uses an editor to display the source around a breakpoint through ToolTalk messages.

The editor has the following Handle pattern in its ptype:

```
(HandlerPtype: TextEditor;
Op: ShowLine;
Scope: TT_SESSION;
Session: my_session_id;
File: /home/butterfly/astrid/src/ebe.c)
```

1. When the debugger reaches a breakpoint, it sends a message that contains the *op* (ShowLine), *argument* (the line number), *file* (the file name), *session* (the current session id), and *scope* (TT_SESSION) attributes.
2. The ToolTalk service matches this message against all registered patterns and finds the pattern registered by the editor.
3. The ToolTalk service delivers the message to the editor.
4. The editor then scrolls to the line indicated in the argument.

Object-Oriented Message Delivery

Many messages handled by the ToolTalk service are directed at objects but are actually delivered to the process that manages the object. The message signatures in an otype, which include the ptype of the process that can handle each specific message, help the ToolTalk service determine process to which it should deliver an object-oriented message.

1. Initialize.

The sender fills in the *class*, *operation*, *arguments*, and the target *objid* attributes.

The sender attribute is automatically filled in by the ToolTalk service. The sender can either fill in the *sender_ptype* and *session* attributes or allow the ToolTalk service to fill in the default values.

If the scope is `TT_FILE`, the file name must be filled in or defaulted. If the scope is `TT_SESSION`, the session name must be filled in or defaulted. If the scope is `TT_BOTH` or `TT_FILE_IN_SESSION`, both the file name and session name must be filled in or defaulted.

Note – The set of patterns checked for delivery depends on the scope of the message. If the scope is `TT_SESSION`, only patterns for processes in the same session are checked. If the scope is `TT_FILE`, patterns for all processes observing the file are checked. If the scope is `TT_FILE_IN_SESSION` or `TT_BOTH`, both sets of processes are checked.

2. Resolve.

The ToolTalk service looks up the *objid* in the ToolTalk database and fills in the otype and file attributes.

3. Dispatch to handler.

The ToolTalk service searches through the otype definitions for Handler message patterns that match the message's *operation* and *arguments* attributes. When a match is found, the ToolTalk service fills in *scope*, *opnum*, *handler_ptype*, and *disposition* from the otype message pattern.

4. Dispatch to object-oriented observers.

The ToolTalk service compares the message's *class*, *operation*, and *argument* attributes against all Observe message patterns of the otype. When a match is found, if the message pattern specifies TT_QUEUE or TT_START, the ToolTalk service attaches a record (called an "observe promise") to the message that specifies the ptype and the queue or start options.

5. Dispatch to procedural observers.

The ToolTalk service continues to match the message's *class*, *operation*, and *argument* attributes against all Observe message patterns of all ptypes. When a match is found, if the signature specifies TT_QUEUE or TT_START, the ToolTalk service attaches an observe promise record to the message, specifying the ptype and the queue or start options.

6. Deliver to handler.

If a running process has a registered Handler pattern that matches the message, the ToolTalk service delivers the message to the process; otherwise, the ToolTalk service honors the disposition (queue or start) options.

If more than one process has registered a dynamic pattern that matches the handler information, the more specific pattern (determined by counting the number of non-wildcard matches) is given preference. If two patterns are equally specific, the choice of handler is arbitrary.

7. Deliver to observers.

The ToolTalk service delivers the message to all running processes that have registered Observer patterns that match the message. As each delivery is made, the ToolTalk service checks off any observe promise for the ptype of the observer. After this process is completed and there are observe promises left unfulfilled, the ToolTalk service honors the disposition (queue or start) options in the promises.

Example

In this example, a hypothetical spreadsheet application named FinnogaCalc is integrated with the ToolTalk service.

1. FinnogaCalc starts and registers with the ToolTalk service by declaring its ptype, FinnogaCalc, and joining its default session.
2. FinnogaCalc loads a worksheet, hatsize.wks, and tells the ToolTalk service it is observing the worksheet by joining the worksheet file.

3. A second instance of `FinnogaCalc` (called `FinnogaCalc2`) starts, loads a worksheet, `wardrobe.wks`, and registers with the ToolTalk service in the same way.
4. The user assigns the value of cell B2 in `hatsize.wks` to also appear in cell C14 of `wardrobe.wks`.
5. So that `FinnogaCalc` can send the value to `FinnogaCalc2`, `FinnogaCalc` creates an object spec for cell C14 by calling a ToolTalk function. This object is identified by an `objid`.
6. `FinnogaCalc2` then gives this `objid` to `FinnogaCalc` (for example, through the clipboard).
7. `FinnogaCalc` remembers that its cell B2 should appear in the object identified by this `objid` and sends a message that contains the value.
8. ToolTalk routes the message. To deliver the message, the ToolTalk service:
 - a. Examines the spec associated with the `objid` and finds that the type of the `objid` is `FinnogaCalc_cell` and that the corresponding object is in the file `wardrobe.wks`.
 - b. Consults the `otype` definition for `FinnogaCalc_cell`. From the `otype`, the ToolTalk service determines that this message is observed by processes of `ptype FinnogaCalc` and that the scope of the message should be `TT_FILE`.
 - c. Matches the message against registered patterns and locates all processes of this `ptype` that are observing the proper file. `FinnogaCalc2` matches, but `FinnogaCalc` does not.
 - d. Delivers the message to `FinnogaCalc2`.
9. `FinnogaCalc2` recognizes that the message contains an object that corresponds to cell C14. `FinnogaCalc2` updates the value in `wardrobe.wks` and displays the new value.

Otype Addressing

Sometimes you may need to send an object-oriented message without knowing the objid. To handle these cases, the ToolTalk service provides otype addressing. This addressing mode requires the sender to specify the operation, arguments, scope, and otype. The ToolTalk service looks in the specified otype definition for a message pattern that matches the message's operation and arguments to locate handling and observing processes. The dispatch and delivery then proceed as in messages to specific objects.

8.4 Modifying Applications to Send ToolTalk Messages

To send ToolTalk messages, your application must perform several operations: it must be able to create and complete ToolTalk messages; it must be able to add message callback routines; and it must be able to send the completed message.

Creating Messages

The ToolTalk service provides three methods to create and complete messages:

1. General-purpose function
 - `tt_message_create()`
2. Process-oriented notice and request functions
 - `tt_pnotice_create()`
 - `tt_prequest_create()`
3. Object-oriented notice and request functions
 - `tt_onotice_create()`
 - `tt_orequest_create()`

The process- and object-oriented notice and request functions make message creation simpler for the common cases. They are functionally identical to strings of other `tt_message_create()` and `tt_message_<attribute>_set()` calls, but are easier to write and read. Table 8-4 and Table 8-5 list the ToolTalk functions that are used to create and complete message

Table 8-4 Functions Used to Create Messages

ToolTalk Function	Description
<code>tt_onotice_create(const char *objid, const char *op)</code>	Creates an object-oriented notice.
<code>tt_orequest_create(const char *objid, const char *op)</code>	Creates an object-oriented request.
<code>tt_pnotice_create(Tt_scope scope, const char *op)</code>	Creates a process-oriented notice.
<code>tt_prequest_create(Tt_scope scope, const char *op)</code>	Creates a process-oriented request.
<code>tt_message_create(void)</code>	Creates a message. This function is the ToolTalk general purpose function to create messages.

Note: The return type for all the create functions is `Tt_message`.

Table 8-5 Functions Used to Complete Messages

ToolTalk Function	Description
<code>tt_message_address_set(Tt_message m, Tt_address p)</code>	Sets addressing mode (for example, point-to-point).
<code>tt_message_arg_add(Tt_message m, Tt_mode n, const char *vtype, const char *value)</code>	Adds a null-terminated string argument.
<code>tt_message_arg_bval_set(Tt_message m, int n, const unsigned char *value, int len)</code>	Sets an argument's value to the specified byte array.
<code>tt_message_arg_ival_set(Tt_message m, int n, int value)</code>	Sets an argument's value to the specified integer.
<code>tt_message_arg_val_set(Tt_message m, int n, const char *value)</code>	Sets an argument's value to the specified null-terminated string.
<code>tt_message_barg_add(Tt_message m, Tt_mode n, const char *vtype, const unsigned char *value, int len)</code>	Adds a byte array argument.
<code>tt_message_iarg_add(Tt_message m, Tt_mode n, const char *vtype, int value)</code>	Adds an integer argument.
<code>tt_message_context_bval(Tt_message m, const char *slotname, unsigned char **value, int *len);</code>	Gets a context's value to the specified byte array.
<code>tt_message_context_ival(Tt_message m, const char *slotname, int *value);</code>	Gets a context's value to the specified integer.
<code>tt_message_context_val(Tt_message m, const char *slotname);</code>	Gets a context's value to the specified string.

Note: The return type for all the functions used to complete messages is `Tt_status`.

Table 8-5 Functions Used to Complete Messages (Continued)

ToolTalk Function	Description
<code>tt_message_icontext_set(Tt_message m, const char *slotname, int value);</code>	Sets a context to the specified integer.
<code>tt_message_bcontext_set(Tt_message m, const char *slotname, unsigned char *value, int length);</code>	Sets a context to the specified byte array.
<code>tt_message_context_set(Tt_message m, const char *slotname, const char *value);</code>	Sets a context to the specified null-terminated string.
<code>tt_message_class_set(Tt_message m, Tt_class c)</code>	Sets the type of message (either notice or request)
<code>tt_message_file_set(Tt_message m, const char *file)</code>	Sets the file to which the message is scoped.
<code>tt_message_handler_ptype_set(Tt_message m, const char *ptid)</code>	Sets the ptype that is to receive the message.
<code>tt_message_handler_set(Tt_message m, const char *procid)</code>	Sets the procid that is to receive the message.
<code>tt_message_object_set(Tt_message m, const char *objid)</code>	Sets the object that is to receive the message.
<code>tt_message_op_set(Tt_message m, const char *opname)</code>	Sets the operation that is to receive the message.
<code>tt_message_otype_set(Tt_message m, const char *otype)</code>	Sets the object type that is to receive the message.
<code>tt_message_scope_set(Tt_message m, Tt_scope s)</code>	Sets the recipients who are to receive the message (file, session, both).
<code>tt_message_sender_ptype_set(Tt_message m, const char *ptid)</code>	Sets the ptype of the application that is sending the message.
<code>tt_message_session_set(Tt_message m, const char *sessid)</code>	Sets the session to which the message is scoped.
<code>tt_message_status_set(Tt_message m, int status)</code>	Sets the status of the message; this status is seen by the receiving application.
<code>tt_message_status_string_set(Tt_message m, const char *status_str)</code>	Sets the text that describes the status of the message; this text is seen by the receiving application.
<code>tt_message_user_set(Tt_message m, int key, void *v)</code>	Sets a message that is internal to the sending application. This internal message is opaque data that is not seen by the receiving application.

Note: The return type for all the functions used to complete messages is `Tt_status`.

Using the General-Purpose Function to Create ToolTalk Messages

You can use the general-purpose function `tt_message_create()` to create and complete ToolTalk messages. If you create a process- or object-oriented message with `tt_message_create()`, use the `tt_message_<attribute>_set()` calls to set the attributes.

Class

- Use `TT_REQUEST` for messages that return values or status. You will be informed when the message is handled or queued, or when a process is started to handle the request.
- Use `TT_NOTICE` for messages that only notify other processes of events.

Address

- Use `TT_PROCEDURE` to send the message to any process that can perform this operation with these arguments. Fill in `op` and `args` attributes of this message.
- Use `TT_OTYPE` to send the message to this type of object that can perform this operation with these arguments. Fill in `otype`, `op`, and `args` attributes of the message.
- Use `TT_HANDLER` to send the message to a specific process. Specify the handler attribute value.

Usually, one process makes a general request, picks the handler attribute from the reply, and directs further messages to that handler. If you specify the exact `procid` of the handler, the ToolTalk service will deliver the message directly — no pattern matching is done and no other applications can observe the message. This point-to-point (PTP) message passing feature enables two processes to rendezvous through broadcast message passing and then communicate explicitly with one another.

- Use `TT_OBJECT` to send the message to a specific object that performs this operation with these arguments. Fill in `object`, `op`, and `args` attributes of this message.

Scope

Fill in the scope of the message delivery. Potential recipients could be joined to:

- `TT_SESSION`
- `TT_FILE`
- `TT_BOTH`
- `TT_FILE_IN_SESSION`

Depending on the scope, the ToolTalk service will add the default session or file, or both to the message.

Op

Fill in the operation that describes the notification or request that you are making. To determine the operation name, consult the ptype definition for the target recipient or the message protocol definition.

Args

Fill in any arguments specific to the operation. Use the function that best suits your argument's data type:

- `tt_message_arg_add()`
Adds an argument whose value is a zero-terminated character string.
- `tt_message_barg_add()`
Adds an argument whose value is a byte string.
- `tt_message_iarg_add()`
Adds an argument whose value is an integer.

For each argument you add (regardless of the value type), specify:

- `Tt_mode`
Specify `TT_IN` or `TT_INOUT`. `TT_IN` indicates that the argument is written by the sender and can be read by the handler and any observers. `TT_INOUT` indicates that the argument is written by the sender and the handler and can be read by all. If you are sending a request that requires the handler to provide an argument in return, use `TT_INOUT`.

- Value Type

The value type (*vtype*) describes the type of argument data that is to be added. The ToolTalk service uses the *vtype* name when it compares a message to registered patterns to determine a message's recipients. The ToolTalk service does not use the *vtype* to process a message or pattern argument value.

The *vtype* name helps the message receiver interpret data. For example, if a word processor rendered a paragraph into a PostScript representation in memory, it could call `tt_message_arg_add` with the following arguments:

```
tt_message_arg_add (m, "PostScript", buf);
```

In this case, the ToolTalk service would assume `buf` pointed to a zero-terminated string and send it.

Similarly, an application could send an enum value in a ToolTalk message; for example, an element of `Tt_status`:

```
tt_message_iarg_add(m, "Tt_status", (int) TT_OK);
```

The ToolTalk service sends the value as an integer but the "Tt_status" *vtype* tells the recipient what the value means.

Note – It is very important that senders and receivers define particular *vtype* names so that a receiver does not attempt to retrieve a value that was stored in another fashion; for example, a value stored as an integer but retrieved as a string.

Creating Process-Oriented Messages

You can easily create process-oriented notices and requests. To get a handle or opaque pointer to a new message object for a procedural notice or request, use the `tt_pnotice_create` or `tt_prequest_create` function. You can then use this handle on succeeding calls to reference the message.

When you create a message with `tt_pnotice_create` or `tt_prequest_create`, you must supply the following two attributes as arguments:

1. Scope

Fill in the scope of the message delivery. Potential recipients could be joined to:

- `TT_SESSION`
- `TT_FILE`
- `TT_BOTH`
- `TT_FILE_IN_SESSION`

Depending on the scope, the ToolTalk service fills in the default session or file (or both).

2. Op

Fill in the operation that describes the notice or request you are making. To determine the operation name, consult the ptype definition for the target process or other protocol definition.

You use the `tt_message_<attribute>_set` calls to complete other message attributes such as operation arguments.

Creating and Completing Object-Oriented Messages

You can easily create object-oriented notices and requests. To get a handle or opaque pointer to a new message object for a object-oriented notice or request, use the `tt_onotice_create` or `tt_orequest_create` function. You can then use this handle on succeeding calls to reference the message.

When you create a message with `tt_onotice_create` or `tt_orequest_create`, you must supply the following two attributes as arguments:

1. `Objid`
Fill in the unique object identifier.
2. `Op`
Fill in the operation that describes the notice or request you are making. To determine the operation name, consult the `pctype` definition for the target process or other protocol definition.

You use the `tt_message_<attribute>_set` calls to complete other message attributes such as operation arguments.

Adding Message Callbacks

When a request contains a message callback routine, the callback routine is automatically called when the reply is received to examine the results of the reply and take appropriate actions.

Note – Callbacks are called in reverse order of registration (for example, the most recently added callback is called first).

You use `tt_message_callback_add` to add the callback routine to your request. When the reply comes back and the reply message has been processed through the callback routine, the reply message must be destroyed before the callback function returns `TT_CALLBACK_PROCESSED`. To destroy the reply message, use `tt_message_destroy`, as illustrated in Figure 8-3.

```

Tt_callback_action
sample_msg_callback(Tt_message m, Tt_pattern p)
{
    ... process the reply msg ...

    tt_message_destroy(m);
    return TT_CALLBACK_PROCESSED;
}

```

Figure 8-3 Destroying a Message

The following code sample is a callback routine, `cntl_msg_callback`, that examines the state field of the reply and takes action if the state is started, handled, or failed.

```

/*
 * Default callback for all the ToolTalk messages we send.
 */

Tt_callback_action
cntl_msg_callback(m, p)
    Tt_message m;
    Tt_pattern p;
{
    int          mark;
    char         msg[255];
    char         *errstr;

    mark = tt_mark();
    switch (tt_message_state(m)) {
        case TT_STARTED:
            xv_set(cntl_ui_base_window, FRAME_LEFT_FOOTER,
                "Starting editor...", NULL);
            break;
        case TT_HANDLED:
            xv_set(cntl_ui_base_window, FRAME_LEFT_FOOTER, "", NULL);
            break;
        case TT_FAILED:

```

```

        errstr = tt_message_status_string(m);
        if (tt_pointer_error(errstr) == TT_OK && errstr) {
            sprintf(msg, "%s failed: %s", tt_message_op(m), errstr);
        } else if (tt_message_status(m) == TT_ERR_NO_MATCH) {
            sprintf(msg, "%s failed: Couldn't contact editor",
                    tt_message_op(m),
                    tt_status_message(tt_message_status(m)));
        } else {
            sprintf(msg, "%s failed: %s",
                    tt_message_op(m),
                    tt_status_message(tt_message_status(m)));
        }
        xv_set(cntl_ui_base_window, FRAME_LEFT_FOOTER, msg, NULL);
        break;
    default:
        break;
}
/*
 * no further action required for this message. Destroy it
 * and return TT_CALLBACK_PROCESSED so no other callbacks will
 * be run for the message.
 */
tt_message_destroy(m);
tt_release(mark);
return TT_CALLBACK_PROCESSED;
}

```

You can also add callbacks to static patterns by attaching a callback to the opnum of a signature in a ptype. When a message is delivered because it matched a static pattern with an opnum, the ToolTalk service checks for any callbacks attached to the opnum and runs them.

- Use `tt_otype_opnum_callback_add` to attach the callback routine to the opnum of an osignature.
- Use `tt_ptype_opnum_callback_add` to attach the callback routine to the opnum of a psignature.

Sending a Message

When you have completed your message, use `tt_message_send` to send it.

If the ToolTalk service returns `TT_WRN_STALE_OBJID`, it has found a forwarding pointer in the ToolTalk database that indicates the object mentioned in the message has been moved. However, the ToolTalk service will send the message with the new objid. You can then use `tt_message_object` to retrieve the new objid from the message and put it into your internal data structure.

If you will not need the message in the future (for example, if the message was a notice), you can use `tt_message_destroy` to delete the message and free storage space.

Note – If you are expecting a reply to the message, do not destroy the message until you have handled the reply.

8.5 Examples

Figure 8-4 illustrates how to create and send a pnotice.

```
/*
 * 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);
```

Figure 8-4 Creating and Sending a Pnotice

Figure 8-5 illustrates how an orequest is created and sent when the callback routine for `cntl_ui_hilite_button` is called.

```

/*
 * Notify callback function for 'cntl_ui_hilite_button'.
 */
void
cntl_ui_hilite_button_handler(item, event)
    Panel_item    item;
    Event         *event;
{
    Tt_message    msg;

    if (cntl_objid == (char *)0) {
        xv_set(cntl_ui_base_window, FRAME_LEFT_FOOTER,
              "No object id selected", NULL);
        return;
    }
    msg = tt_orequest_create(cntl_objid, "hilite_obj");
    tt_message_arg_add(msg, TT_IN, "string", cntl_objid);
    tt_message_callback_add(msg, cntl_msg_callback);
    tt_message_send(msg);
}

```

Figure 8-5 Creating and Sending an Orequest

Dynamic Message Patterns

9.1 Defining Dynamic Messages

The dynamic method provides message pattern information while your application is running. You create a message pattern and register it with the ToolTalk service. You can add callback routines to dynamic message patterns that the ToolTalk service will call when it matches a message to the pattern.

To create and register a dynamic message pattern, you allocate a new pattern object, fill in the proper information, and register it. When you are done with the pattern (that is, when you are no longer interested in messages that match it), either unregister or destroy the pattern. You can register and unregister dynamic message patterns as needed.

The ToolTalk functions used to create, register, and unregister dynamic message patterns are listed in Table 9-1.

Table 9-1 Functions for Creating, Updating, and Deleting Message Patterns

ToolTalk Function	Description
tt_pattern_create(void)	Create Pattern
tt_pattern_arg_add(Tt_pattern p, Tt_mode n, const char *vtype, const char *value)	Add string arguments
tt_pattern_barg_add(Tt_pattern m, Tt_mode n, const char *vtype, const unsigned char *value, int len)	Add byte array arguments

Note: The return type for all functions except tt_pattern_create is Tt_status; tt_pattern_create returns Tt_pattern.

Table 9-1 Functions for Creating, Updating, and Deleting Message Patterns (Continued)

ToolTalk Function	Description
<code>tt_pattern_iarg_add(Tt_pattern m, Tt_mode n, const char *vtype, int value)</code>	Add integer arguments
<code>tt_pattern_xarg_add(Tt_pattern m, Tt_mode n, const char *vtype, xdrproc_t xdr_proc, void *value)</code>	Adds an xdr argument to a byte array
<code>tt_pattern_bcontext_add(Tt_pattern p, const char *slotname, const unsigned char *value, int length);</code>	Add byte array contexts
<code>tt_pattern_context_add(Tt_pattern p, const char *slotname, const char *value);</code>	Add string contexts
<code>tt_pattern_icontext_add(Tt_pattern p, const char *slotname, int value);</code>	Add integer contexts
<code>tt_pattern_address_add(Tt_pattern p, Tt_address d)</code>	Add address
<code>tt_pattern_callback_add(Tt_pattern p, Tt_message_callback_action f)</code>	Add message callback
<code>tt_pattern_category_set(Tt_pattern p, Tt_category c)</code>	Set category
<code>tt_pattern_class_add(Tt_pattern p, Tt_class c)</code>	Add class
<code>tt_pattern_disposition_add(Tt_pattern p, Tt_disposition r)</code>	Add disposition
<code>tt_pattern_file_add(Tt_pattern p, const char *file)</code>	Add file
<code>tt_pattern_object_add(Tt_pattern p, const char *objid)</code>	Add object
<code>tt_pattern_op_add(Tt_pattern p, const char *opname)</code>	Add operation
<code>tt_pattern_opnum_add(Tt_pattern p, int opnum)</code>	Add operation number
<code>tt_pattern_otype_add(Tt_pattern p, const char *otype)</code>	Add object type
<code>tt_pattern_scope_add(Tt_pattern p, Tt_scope s)</code>	Ad scope
<code>tt_pattern_sender_add(Tt_pattern p, const char *procid)</code>	Add sending process identifier
<code>tt_pattern_sender_ptype_add(Tt_pattern p, const char *ptid)</code>	Add sending process type
<code>tt_pattern_session_add(Tt_pattern p, const char *sessid)</code>	Add session identifier
<code>tt_pattern_state_add(Tt_pattern p, Tt_state s)</code>	Add state
<code>tt_pattern_user_set(Tt_pattern p, int key, void *v)</code>	Set user
<code>tt_pattern_register(Tt_pattern p)</code>	Register pattern
<code>tt_pattern_unregister(Tt_pattern p)</code>	Unregister pattern
<code>tt_pattern_destroy(Tt_pattern p)</code>	Destroy message pattern

Note: The return type for all functions except `tt_pattern_create` is `Tt_status`; `tt_pattern_create` returns `Tt_pattern`.

Creating a Message Pattern

To create message patterns, use the `tt_pattern_create` function. You can use this function to get a handle or opaque pointer to a new pattern object, and then use this handle on succeeding calls to reference the pattern.

To fill in pattern information, use the `tt_pattern_<attribute>_add` and `tt_pattern_<attribute>_set` calls. You can supply multiple values for each attribute you add to a pattern. The pattern attribute matches a message attribute if any of the values in the pattern match the value in the message. If no value is specified for an attribute, the ToolTalk service assumes that you want any value to match. Some attributes are set and, therefore, can only have one value.

Adding a Message Pattern Callback

To add a callback routine to your pattern, use the `tt_pattern_callback_add` function.

Note – Callbacks are called in reverse order of registration (for example, the most recently added callback is called first).

When the ToolTalk service matches a message, it automatically calls your callback routine to examine the message and take appropriate actions. When a message that matches a pattern with a callback is delivered to you, it is processed through the callback routine. When the routine is finished, it returns `TT_CALLBACK_PROCESSED` and the API objects involved in the operation are freed. You can then use `tt_message_destroy` to destroy the message, which frees the storage used by the message, as illustrated in the following code sample.

```
Tt_callback_action
sample_msg_callback(Tt_message m, Tt_pattern p)
{
    ... process the reply msg ...

    tt_message_destroy(m);
    return TT_CALLBACK_PROCESSED;
}
```

Registering a Message Pattern

To register the completed pattern, use the `tt_pattern_register` function. After you register your pattern, you join the sessions or files of interest.

The following code sample creates and registers a pattern.

```
/*
 * 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);
```

Deleting and Unregistering a Message Pattern

Note - If delivered messages that matched the deleted pattern have not been retrieved by your application (for example, the messages might be queued), the ToolTalk service does not destroy these messages.

To delete a message pattern, use the `tt_pattern_destroy` function. This function first unregisters the pattern and then destroys the pattern object.

To stop receiving messages that match a message pattern without destroying the pattern object, use the `tt_pattern_unregister` to unregister the pattern.

The ToolTalk service will automatically unregister and destroy all message pattern objects when you call `tt_close`.

9.2 Updating Message Patterns with the Current Session or File

To update your message patterns with the session or file in which you are currently interested, join the session or file.

Joining the Default Session

When you join a session, the ToolTalk service updates your message pattern with the sessid. For example, if you have declared a ptype or registered a message pattern that specifies `TT_SESSION` or `TT_FILE_IN_SESSION`, use `tt_session_join` to join the default session. The following code sample shows how to join the default session.

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

tt_session_join(tt_default_session());
```

Table 9-2 lists the ToolTalk functions you use to join the session in which you are interested.

Table 9-2 ToolTalk Functions for Joining Default Sessions

Return Type	ToolTalk Function	Description
char *	<code>tt_default_session(void)</code>	Return default session id
Tt_status	<code>tt_default_session_set(const char *sessid)</code>	Set default session
char *	<code>tt_initial_session(void)</code>	Return initial session id
Tt_status	<code>tt_session_join(const char *sessid)</code>	Join this session
Tt_status	<code>tt_session_quit(const char *sessid)</code>	Quit session

Once your patterns are updated, you will begin to receive messages scoped to the session you joined.

Note – If you had previously joined a session and then registered a ptype or a new message pattern, you must again join the same session or a new session to update your pattern before you will receive messages that match your new pattern.

When you no longer want to receive messages that reference the default session, use the `tt_session_quit` function. This function removes the `sessid` from your session-scoped message patterns.

Joining Multiple Sessions

When you join multiple sessions, you will automatically get responses to requests and point-to-point messages but you will not get notices unless you explicitly join the new session. The following code sample shows how to join the multiple sessions.

```
tt_default_session_set(new_session_identifier);
tt_open();
tt_session_join(new_session);
```

In order to effectively use multiple sessions, you must store the session ids of the sessions in which you are interested in order to pass these identifiers to `tt_default_session_set` prior to opening a new session with `tt_open`; that is, you need to place the values (which `ttsession` stores in the environment variable `_SUN_TT_SESSION`) in a file on the system so that other ToolTalk clients can access the value of a session id contained in that file and use it to open the non-default session. For example, you can store the session ids in a “well-known” file and then send a file-scoped message (indicating this file) to all clients which have registered an appropriate pattern. The client will then know to open the scoped-to file, read one or more session ids from it, and use these session ids (with `tt_open`) to open a non-default session. An alternative method is advertising the session ids by means of, for example, a name service or a third-party database.

Note – How `ttsession` session ids are stored and passed to interested clients is beyond the scope of the ToolTalk protocol and must be determined based on the architecture of the system.

Joining Files of Interest

When you join a file, the ToolTalk service automatically adds the name of the file to your file-scoped message patterns. For example, if you have declared a process type or registered a message pattern that specifies `TT_FILE` or `TT_FILE_IN_SESSION`, use the `tt_file_join` function to join files of interest. Table 9-3 lists the ToolTalk functions you use to express your interest in specific files.

Table 9-3 ToolTalk Functions for Joining Files of Interest

Return Type	ToolTalk Function	Description
char *	<code>tt_default_file(void)</code>	Join default file
Tt_status	<code>tt_default_file_set(const char *docid)</code>	Set default file
Tt_status	<code>tt_file_join(const char *filepath)</code>	Join this file
Tt_status	<code>tt_file_quit(const char *filepath)</code>	Quit file

When you no longer want to receive messages that reference the file, use the `tt_file_quit` function to remove the file name from your file-scoped message patterns.

10.1 Defining Static Messages

The static messaging method provides an easy way to specify the message pattern information if you want to receive a defined set of messages.

To use the static method, you define your process types and object types and compile them with the ToolTalk type compiler, `tt_type_comp`. When you declare your process type, the ToolTalk service creates message patterns based on that type. These static message patterns remain in effect until you close communication with the ToolTalk service.

10.2 Defining Process Types

Your application can still be considered a potential message receiver even when no process is running the application. To do this, you provide message patterns and instructions on how to start the application in a process type (*ptype*) file. These instructions tell the ToolTalk service to perform one of the following actions when a message is available for an application but the application is not running:

- Start the application and deliver the message
- Queue the message until the application is running
- Discard the message

To make the information available to the ToolTalk service, the ptype file is compiled with the ToolTalk type compiler, `tt_type_comp`, at application installation time.

When an application registers a ptype with the ToolTalk service, the message patterns listed in it are automatically registered, too.

Ptypes provide application information that the ToolTalk service can use when the application is not running. This information is used to start your process if necessary to receive a message or queue messages until the process starts.

A ptype begins with a process-type identifier (*ptid*). Following the *ptid* are:

1. An optional start string — The ToolTalk service will execute this command, if necessary, to start a process running the program.
2. Signatures — Describes the `TT_PROCEDURE`-addressed messages that the program wants to receive. Messages to be observed are described separately from messages to be handled.

Signatures

Signatures describe the messages that the program wants to receive. A signature is divided by an arrow (`=>`) into two parts. The first part of a signature specifies matching attribute values. The more attribute values specified in a signature, the fewer messages the signature will match. The second part of a signature specifies receiver values that the ToolTalk service will copy into messages that match the first part of the signature.

A ptype signature can contain values for disposition and operation numbers (*opnum*). The ToolTalk service uses the disposition value (start, queue, or the default discard) to determine what to do with a message that matches the signature when no process is running the program. The *opnum* value is provided as a convenience to message receivers. When two signatures have the same operation name but different arguments, different *opnums* makes incoming messages easy to identify.

Creating a Ptype File

Figure 10-1 illustrates a ptype file.

```
#include "Sun_EditDemo_opnums.h"

ptype Sun_EditDemo {
    /* setenv Sun_EditDemo_HOME to install dir for the demo */
    start "${Sun_EditDemo_HOME}/edit";
    handle:
    /* edit file named in message, start editor if necessary */
    session Sun_EditDemo_edit(void)
        => start opnum=Sun_EditDemo_EDIT;

    /* tell editor viewing file in message to save file */
    session Sun_EditDemo_save(void)
        => opnum=Sun_EditDemo_SAVE;

    /* save file named in message to new filename */
    session Sun_EditDemo_save_as(in string new_filename)
        => opnum=Sun_EditDemo_SAVE_AS;

    /* bring down editor viewing file in message */
    session Sun_EditDemo_close(void)
        => opnum=Sun_EditDemo_CLOSE;
};
```

Figure 10-1 Example of a Ptype File

Figure 10-2 shows the syntax for a ptype file.

```

ptype      ::= 'ptype' ptid '{'
              property*
              ['observe:' psignature*]
              ['handle:' psignature* ]
              '}' [';']

property   ::= property_id value ';'
property_id ::= 'start'
value      ::= string
ptid       ::= identifier
psignature ::= [scope] op args [contextdcl]
              ['=>'
              ['start']['queue']
              ['opnum='number]]
              ';'

scope      ::= 'file'
              | 'session'
              | 'file_in_session'
args       ::= '(' argspec {, argspec}* ')'
              | '(void)'
              | '()'

contextdcl ::= 'context' '(' identifier {, identifier}* ')' ';'
argspec    ::= mode type name
mode       ::= 'in' | 'out' | 'inout'
type       ::= identifier
name       ::= identifier

```

Figure 10-2 Ptype Syntax

Property_id Information

ptid

process type identifier (ptid). Identifies the process type. A ptid must be unique for every installation. Because this identifier cannot be changed after installation time, each chosen name must be unique. For example, you can use a name that includes the trademarked name of your product or company, such as Sun_EditDemo. The ptid cannot exceed 32 characters and should not be one of the reserved identifiers: ptype, otype, start, opnum, queue, file, session, observe, or handle.

start

Start string for the process. If the ToolTalk service needs to start a process, it executes this command; `/bin/sh` is used as the shell.

Before executing the command, the ToolTalk service defines `TT_FILE` as an environment variable with the value of the file attribute of the message that started the application. This command runs in the environment of `ttsession`, not in the environment of the sender of the message that started the application, so any context information must be carried by message arguments or contexts.

Psignature Matching Information

scope

This pattern attribute is matched against the scope attribute in messages.

op

Operation name. This name is matched against the op attribute in messages.

Note – If you specify message signatures in both your ptype and otypes, use unique operation names in each. For example, do not specify a display operation in both your ptype and otype.

args

Arguments for the operation. If the args list is `(void)`, the signature matches only messages with no arguments. If the args list is empty (that is, `()`), the signature matches without regard to the arguments.

contextdcl

Context name. When a pattern with this named context is generated from the signature, it contains an empty value list.

Psignature Actions Information

start

If the psignature matches a message and no running process of this ptype has a pattern that matches the message, start a process of this ptype.

queue

If the `psignature` matches a message and no running process of this `ptype` has a pattern that matches the message, queue the message until a process of this `ptype` registers a pattern that matches it.

opnum

Fill in the message's `opnum` attribute with the specified number to enable you to identify the signature that matched the message.

When the message matches the signature, the `opnum` from the signature is filled into the message. Your application can then retrieve the `opnum` with the `tt_message_opnum` call. By giving each signature a unique `opnum`, you can quickly determine which signature matched the message.

You can attach a callback routine to the `opnum` with the `tt_ptype_opnum_callback_add` call. When the message is matched, the ToolTalk service will check for any callbacks attached to the `opnum` and, if any are found, run them.

In Figure 10-1, the `Sun_EditDemo_opnums.h` file defines symbolic definitions for all the `opnums` used by `edit.c`, allowing both the `edit.types` file and `edit.c` file to share the same definitions.

Automatically Starting a Tool

Figure 10-3 is a simple example of a `ptype` declaration that causes the ToolTalk service to automatically start a tool. The example code states:

If a message to display, edit, or compose is received and there is no current instance of the tool running that can handle the message, start `"/home/toone/tools/mytest"` and deliver the message.



Caution – This example causes the ToolTalk service to search indefinitely for a handler.

```
ptype My_Test {
    start "/home/toone/tools/mytest";
    handle:

    session Display (in Ascii text) => start;
    session Edit (inout Ascii text) => start;
    session Compose (out Ascii text) => start;

    file Display (in Ascii file_name) => start;
    file Edit (inout Ascii file_name) => start;
    file Compose (out Ascii file_name) => start;
};
```

Figure 10-3 Automatically Starting a Tool

10.3 Defining Object Types

When a message is addressed to a specific object or a type of object, the ToolTalk service must be able to determine to which application the message is to be delivered. Applications provide this information in an *object type (otype)*. An otype names the ptype of the application that manages the object and describes message patterns that pertain to the object.

These message patterns also contain instructions that tell the ToolTalk service what to do if a message is available but the application is not running. In this case, ToolTalk performs one of the following instructions:

- Start the application and deliver the message
- Queue the message until the application is running
- Discard the message

To make the information available to the ToolTalk service, the otype file is compiled with the ToolTalk type compiler `tt_type_comp` at application installation time. When an application that manages objects registers with the ToolTalk service, it declares its ptype. When a ptype is registered, the ToolTalk service checks for otypes that mention the ptype and registers the patterns found in these otypes.

The otype for your application provides addressing information that the ToolTalk service uses when delivering object-oriented messages. The number of otypes you have, and what they represent, depends on the nature of your application. For example, a word processing application might have otypes for characters, words, paragraphs, and documents; a diagram editing application might have otypes for nodes, arcs, annotation boxes, and diagrams.

An otype begins with an object-type identifier (*otid*). Following the otid are:

1. An optional start string — ToolTalk will execute this command, if necessary, to start a process running the program.
2. Signatures — Code that defines the messages that can be addressed to objects of the type (that is, the operations that can be invoked on objects of the type).

Signatures

Signatures defines the messages that can be addressed to objects of the type. A signature is divided by an arrow ($=>$) into two parts. The first part of a signature define matching criteria for incoming messages. The second part of a signature defines receiver values which the ToolTalk service adds to each message that matches the first part of the signature. These values specify the ptid of the program that implements the operation and the message's scope and disposition.

Creating Otype Files

Figure 10-4 shows the syntax for an otype file.

```
otype          ::= obj_header '{' objbody* '}' [';']
obj_header     ::= 'otype' otid [':' otid*]
objbody        ::= 'observe:' osignature*
               | 'handle:' osignature*

osignature     ::= op args [contextdcl] [rhs][inherit] ';'
rhs            ::= ['=>' ptid [scope]]
               ['start']['queue']
               ['opnum='number]
inherit        ::= 'from' otid
args           ::= '(' argspec {, argspec}* ')'
               | '(void)'
               | '()'
contextdcl     ::= 'context' '(' identifier {, identifier}* ')' ';'
argspec        ::= mode type name
mode           ::= 'in' | 'out' | 'inout'
type           ::= identifier
name           ::= identifier
otid           ::= identifier
ptid           ::= identifier
```

Figure 10-4 Otype Syntax

Obj_Header Information

otid

object type identifier (otid). Identifies the object type. An otid must be unique for every installation. Because this identifier cannot be changed after installation time, each chosen name must be unique. For example, begin with the ptid of the tool that implements the otype. The otid is limited to 64 characters and should not be one of the reserved identifiers: ptype, otype, start, opnum, start, queue, file, session, observe, or handle.

Osignment Information

The object body portion of the otype definition is a list of osignatures for messages about the object that your application wants to observe and handle.

op

Operation name. This name is matched against the op attribute in messages.

Note – If you specify message signatures in both your ptype and otypes, use unique operation names in each. For example, do not specify a display operation in both your ptype and otype.

args

Arguments for the operation. If the args list is (void), the signature matches only messages with no arguments. If the args list is empty (just “”), the signature matches messages without regard to the arguments.

contextdcl

Context name. When a pattern with this named context is generated from the signature, it contains an empty value list.

ptid

Process type identifier for the application that manages this type of object.

opnum

Fill in the message’s opnum attribute with the specified number to enable you to identify the signature that matched the message.

When the message matches the signature, the opnum from the signature is filled into the message. Your application can then retrieve the opnum with the tt_message_opnum call. By giving each signature a unique opnum, you can quickly determine which signature matched the message.

You can attach a callback routine to the opnum with the tt_otype_opnum_callback_add call. When the message is matched, the ToolTalk service will check for any callbacks attached to the opnum and, if any are found, run them.

inherit

Otypes form an inheritance hierarchy in which operations can be inherited from base types. The ToolTalk service requires the otype definer to explicitly name all inherited operations and the otype from which to inherit. This

explicit naming prevents later changes (such as adding a new level to the hierarchy, or adding new operations to base types) from unexpectedly affecting the behavior of an otype.

scope

This pattern attribute is matched against the scope attribute in messages. It appears on the rightmost side of the arrow and is filled in by the ToolTalk service during message dispatch. This means the definer of the otype can specify the attributes instead of requiring the message sender to know how the message should be delivered.

Osignature Actions Information

start

If the osignature matches a message and no running process of this otype has a pattern that matches the message, start a process of this otype.

queue

If the osignature matches a message and no running process of this otype has a pattern that matches the message, queue the message until a process of this otype registers a pattern that matches it.

Figure 10-5 illustrates an otype file.

```
#include "Sun_EditDemo_opnums.h"

otype Sun_EditDemo_object {
    handle:
    /* hilite object given by objid, starts an editor if necessary */
    hilite_obj(in string objid)
        => Sun_EditDemo session start opnum=Sun_EditDemo_HILITE_OBJ;
};
```

Figure 10-5 Example of an Otype File

The `Sun_EditDemo_opnums.h` file defines symbolic definitions for all the opnums used by `edit.c`, allowing both the `edit.types` file and `edit.c` file to share the same definitions.

10.4 Installing Type Information

The ToolTalk Types Database makes `ptype` and `otype` information available on the host that executes the sending process, the host that executes the receiving process, and the hosts that run the sessions to which the processes are joined.

- To start applications and to queue messages, the `ptype` definition must be placed into the ToolTalk Types Database.
- To receive messages addressed to objects your application creates and manages, the `otype` definitions must also be installed in the ToolTalk Types Database.

To place your type information into the ToolTalk Types Database and make it available to the ToolTalk service, you compile your type files with the ToolTalk type compiler, `tt_type_comp`. This compiler creates ToolTalk types definitions for your type information and stores them in the ToolTalk Types Database. See Chapter 5, “Maintaining Application Information,” for detailed information.

This version of the ToolTalk service provides a function to merge a compiled ToolTalk type file into the currently running `ttsession`:

```
tt_session_types_load(current_session, compiled_types_file)
```

where `current_session` is the current default ToolTalk session and `compiled_types_file` is the name of the compiled ToolTalk types file. This function adds new types and replaces existing types of the same name; other existing types remain unchanged.

10.5 Checking for Existing Process Types

The ToolTalk service provides a simple function to test if a given `ptype` is already registered in the current session.

```
tt_ptype_exists(const char *ptid)
```

where `ptid` is the identifier of the session to test for registration.

10.6 Declaring Process Type

Since type information is only specified once (when your application is installed), your application needs to only declare its ptype each time it starts.

To declare your ptype, use `tt_ptype_declare` during your application's ToolTalk initialization routine. The ToolTalk service will create the message patterns listed in your ptype and any otypes that reference the specified ptype.

The message patterns created when you declare your ptype exist in memory until your application exits the ToolTalk session.

Note - The message patterns created when you declare your ptype information cannot be unregistered with `tt_pattern_unregister`; however, you can unregister these patterns with `tt_ptype_undeclare`.

Figure 10-6 illustrates how a ptype is registered during a program's initialization.

```

/*
 * Initialize our ToolTalk environment.
 */
int
edit_init_tt()
{
    int      mark;
    char     *procid = tt_open();
    int      ttfd;
    void     edit_receive_tt_message();

    mark = tt_mark();

    if (tt_pointer_error(procid) != TT_OK) {
        return 0;
    }
    if (tt_ptype_declare("Sun_EditDemo") != TT_OK) {
        fprintf(stderr, "Sun_EditDemo is not an installed ptype.\n");
        return 0;
    }
    ttfd = tt_fd();
    tt_session_join(tt_default_session());
    notify_set_input_func(edit_ui_base_window,
                          (Notify_func)edit_receive_tt_message,
                          ttfd);

    /*
     * Note that without tt_mark() and tt_release(), the above
     * combination would leak storage -- tt_default_session() returns
     * a copy owned by the application, but since we don't assign the
     * pointer to a variable we could not free it explicitly.
     */

    tt_release(mark);
    return 1;
}

```

Figure 10-6 Example of a Ptype Being Registered During Initialization

10.7 Undeclaring Process Types

There may be cases when you need to retract a declared ptype; for example, in the CASE environment:

- An installation sets up a compile server which declares itself willing to accept compilation requests when it comes up. Once the server has accepted a request, it changes state and will no longer accept new compilation requests.
- A generic encapsulation process declares itself as multiple ptypes and then forwards requests to underlying tools. If an underlying tool exits, the generic wrapper no longer wants to declare itself as the ptype associated with that tool.

To unregister a ptype, use `tt_ptype_undeclare`. This call reverses the effect of the `tt_ptype_declare` call; that is, all patterns generated from the ptype are unregistered and the process is removed from the session's list of active processes with this ptype. This call returns a status of `TT_ERR_PTYPE` if the named ptype was not declared by the calling process.



Caution – One invocation of `tt_type_undeclare` will *completely* unregister the ptype regardless of how many times the process has declared the ptype; that is, multiple declarations of the ptype are the same as declaring it once.

Code Example 10-1 is an example of how to retract a declared ptype.

Code Example 10-1 Undeclaring a Ptype

Obtain procid	<code>tt_open();</code>
Undeclare Ptype	<code>tt_ptype_undeclare(ptype);</code>

Receiving Messages

11 

This chapter describes how to retrieve messages delivered to your application and how to handle the message once you have examined it. It also shows you how to send replies to requests that you receive.

To retrieve and handle ToolTalk messages, your application must perform several operations: it must be able to retrieve ToolTalk messages; it must be able to examine messages; it must provide callback routines; it must be able to respond to requests; and it must be able to destroy the message when it is no longer needed.

11.1 Retrieving Messages

When a message arrives for your process, the ToolTalk-supplied file descriptor becomes active. When notified of the active state of the file descriptor, your process must call `tt_message_receive` to get a handle for the incoming message.

Figure 11-1 illustrates how to receive a message.

```
/*
 * 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;
```

Figure 11-1 Receiving a Message

Handles for messages remain constant. For example, when a process sends a message, both the message and any replies to the message have the same handle as the sent message. Figure 11-2 is an example of how you can check the message state for `TT_HANDLED`.

```
Tt_message m, n;
m = tt_message_create();
...
tt_message_send(m);

... wait around for tt_fd to become active

n = tt_message_receive();
if (m == n) {
    /* This is the reply to the message we sent */
    if (TT_HANDLED == tt_message_state(m) ) {
        /* The receiver has handled the message so we can go on */
        ...
    }
} else {
    /* This is a new message coming in */
}
```

Figure 11-2 Code Checking the Message State

Identifying and Processing Messages Easily

To easily identify and process messages received by you:

- Add a callback to a dynamic pattern with `tt_pattern_callback_add`. When you retrieve the message, the ToolTalk service will invoke any message or pattern callbacks. See Chapter 9, “Dynamic Message Patterns,” for more information on placing callbacks on patterns.
- Retrieve the message’s `opnum` if you are receiving messages that match your `ptype` message patterns.

Recognizing and Handling Replies Easily

To easily recognize and handle replies to messages sent by you:

- Place specific callbacks on requests before you send them with `tt_message_callback_add`. See Chapter 8, “Sending Messages,” for more information on placing callbacks on messages.
- Compare the handle of the message you sent with the message you just received. The handles will be the same if the message is a reply.
- Add information meaningful to your application on the request with the `tt_message_user_set` call.

11.2 Checking Message Status

When you receive a message, you must check its status. If the status is `TT_WRN_START_MESSAGE`, you must either reply, reject, or fail the message even if the message is a notice, or issue a `tt_message_accept` call.

11.3 Examining Messages

When your process receives a message, you examine the message and take appropriate action.

Before you start to retrieve values, obtain a mark on the ToolTalk API stack so that you can release the information the ToolTalk service returns to you all at once. Figure 11-3 allocates storage, examines message contents, and releases the storage.

```

/*
 * 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))) {
    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;

```

Figure 11-3 Allocating, Examining, and Releasing Storage

Table 11-1 lists the ToolTalk functions you use to examine the attributes of a message you have received.

Table 11-1 Functions to Examine Message Attributes

Return Type	ToolTalk Function	Description
Tt_address	tt_message_address(Tt_message m)	The address of the message.
Tt_status	tt_message_arg_bval(Tt_message m, int n, unsigned char **value, int *len)	The argument value as a byte array.
Tt_status	tt_message_arg_ival(Tt_message m, int n, int *value)	The argument value as an integer.
Tt_status	tt_message_arg_xval(Tt_message m, int n, xdrproc_t xdr_proc, void *value)	The argument value as an xdr.
Tt_mode	tt_message_arg_mode(Tt_message m, int n)	The argument mode (in, out, inout).
char *	tt_message_arg_type(Tt_message m, int n)	The argument type.
char *	tt_message_arg_val(Tt_message m, int n)	The argument value as a string.
int	tt_message_args_count(Tt_message m)	The number of arguments.

Table 11-1 Functions to Examine Message Attributes (Continued)

Return Type	ToolTalk Function	Description
Tt_class	tt_message_class(Tt_message m)	The type of message (notice or request).
int	tt_message_contexts_count(Tt_message m);	The number of contexts.
char *	tt_message_context_slotname(Tt_message m, int n);	The name of a message's <i>nth</i> context.
Tt_disposition	tt_message_disposition(Tt_message m)	How to handle the message if there is no receiving application running.
char *	tt_message_file(Tt_message m)	The name of the file to which the message is scoped.
gid_t	tt_message_gid(Tt_message m)	The group identifier of the sending application.
char *	tt_message_handler(Tt_message m)	The procid of the handler.
char *	tt_message_handler_ptype(Tt_message m)	The ptype of the handler.
char *	tt_message_object(Tt_message m)	The object to which the message was sent.
char *	tt_message_op(Tt_message m)	The operation name.
int	tt_message_opnum(Tt_message m)	The operation number.
char *	tt_message_otype(Tt_message m)	The object type to which the message was sent.
Tt_pattern	tt_message_pattern(Tt_message m)	The pattern to which the message is to be matched.
Tt_scope	tt_message_scope(Tt_message m)	Who is to receive the message (FILE, SESSION, BOTH)
char *	tt_message_sender(Tt_message m)	The procid of the sending application.
char *	tt_message_sender_ptype(Tt_message m)	The ptype of the sending application.
char *	tt_message_session(Tt_message m)	The session from which the message was sent.
Tt_state	tt_message_state(Tt_message m)	The current state of the message.

Table 11-1 Functions to Examine Message Attributes (Continued)

Return Type	ToolTalk Function	Description
int	tt_message_status(Tt_message m)	The current status of the message.
char *	tt_message_status_string(Tt_message m)	Text describing the current status of the message.
uid_t	tt_message_uid(Tt_message m)	The user identifier of the sending application.
void *	tt_message_user(Tt_message m, int key)	Opaque data internal to the application.

11.4 *Callback Routines*

You can tell the ToolTalk service to invoke a callback when a message arrives because a pattern has been matched.

Note – Callbacks are called in reverse order of registration (for example, the most recently added callback is called first).

Figure 11-4 illustrates how the ToolTalk service invokes message and pattern callbacks when `tt_message_receive` is called to retrieve a new message.

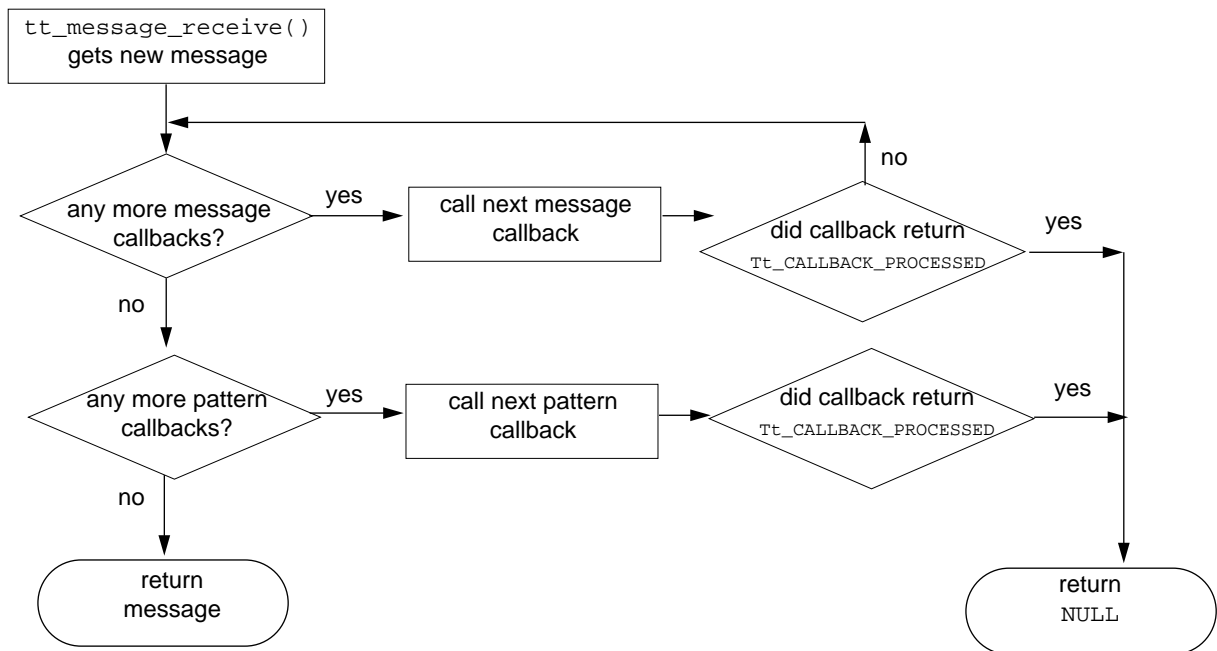


Figure 11-4 How Callbacks Are Invoked

Callbacks for Messages Addressed to Handlers

After the ToolTalk service determines the receiver for a message addressed to a handler, it matches the message against any patterns registered by the receiver. (Messages explicitly addressed to handlers are *point-to-point* messages and do not use pattern matching.)

- If the message does not match a pattern, the message is delivered in the normal manner.
- If the message is matched to a pattern, any callbacks attached to the pattern are run.

Attaching Callbacks to Static Patterns

Numeric tags (opnums) can be attached to each signature in a ptype when a static pattern is created. A callback can now be attached to the opnum. When a message is delivered because it matched a static pattern with an opnum, the ToolTalk service checks for any callbacks attached to the opnum and, if any exists, runs them.

11.5 Handling Requests

When your process receives a request (class = `TT_REQUEST`), you must either reply to the request, or reject or fail the request.

Replying to Requests

When you reply to a request, you need to:

1. **Perform the requested operation.**
2. **Fill in any argument values with modes of `TT_OUT` or `TT_INOUT`.**
3. **Send the reply to the message.**

Table 11-2 lists the ToolTalk functions you use to reply to requests.

Table 11-2 Functions to Reply to Requests

ToolTalk Function	Description
<code>tt_message_arg_mode(Tt_message m, int n)</code>	The argument mode (in, out, inout). [†]
<code>tt_message_arg_bval_set(Tt_message m, int n, const unsigned char *value, int len)</code>	Sets an argument's value to the specified byte array. [‡]
<code>tt_message_arg_ival_set(Tt_message m, int n, int value)</code>	Sets an argument's value to the specified integer. [‡]
<code>tt_message_arg_val_set(Tt_message m, int n, const char *value)</code>	Sets an argument's value to the specified string. [‡]
<code>tt_message_arg_xval_set(Tt_message m, int n, xdrproc_t xdr_proc, void *value)</code>	[‡]

[†] Return type is `Tt_mode`.

[‡] Return type is `Tt_status`.

Table 11-2 Functions to Reply to Requests (Continued)

ToolTalk Function	Description
tt_message_context_set(Tt_message m, const char *slotname, const char *value);	Sets a context to the specified string. †
tt_message_bcontext_set(Tt_message m, const char *slotname, unsigned char *value, int length);	Sets a context to the specified byte array. †
tt_message_icontext_set(Tt_message m, const char *slotname, int value);	Sets a context to the specified integer. †
tt_message_xcontext_set(Tt_message m, const char *slotname, xdrproc_t xdr_proc, void *value)	†
tt_message_reply(Tt_message m)	Replies to message ‡

† Return type is Tt_mode.
‡ Return type is Tt_status.

Rejecting or Failing a Request

If you have examined the request and your application is not currently able to handle the request, you can use the ToolTalk functions listed in Table 11-3 to reject or fail a request.

Table 11-3 Rejecting or Failing Requests

ToolTalk Function	Description
tt_message_reject(Tt_message m)	Rejects message
tt_message_fail(Tt_message m)	Fails message
tt_message_status_set(Tt_message m, int status)	Sets the status of the message; this status is seen by the receiving application.
tt_message_status_string_set(Tt_message m, const char *status_str)	Sets the text that describes the status of the message; this text is seen by the receiving application.

Return type for these requests is Tt_status.

Rejecting a Request

If you have examined the request and your application is not currently able to perform the operation but another application might be able to do so, use `tt_message_reject` to reject the request.

When you reject a request, the ToolTalk service attempts to find another receiver to handle it. If the ToolTalk service cannot find a handler that is currently running, it examines the disposition attribute, and either queues the message or attempts to start applications with ptypes that contain the appropriate message pattern.

Failing a Request

If you have examined the request and the requested operation cannot be performed by you or any other process with the same ptype as yours, use `tt_message_fail` to inform the ToolTalk service that the operation cannot be performed. The ToolTalk service will inform the sender that the request failed.

To inform the sender of the reason the request failed, use `tt_message_status_set` or `tt_message_status_string_set` before you call `tt_message_fail`.

Note - The status code you specify with `tt_message_status_set` must be greater than `TT_ERR_LAST`.

11.6 Destroying Messages

After you have processed a message and no longer need the information in the message, use `tt_message_destroy` to delete the message and free storage space.

This chapter describes how to create ToolTalk specs for objects your application creates and manages. Before you can identify the type of objects, you need to define otypes and store them in the ToolTalk Types Database. See Chapter 10, “Static Message Patterns,” for more information on otypes.

The ToolTalk service uses spec and otype information to determine object-oriented message recipients.

Note – Programs coded to the ToolTalk object-oriented messaging interface are not portable to CORBA-compliant systems without source changes.

12.1 Object-Oriented Messaging

Object-oriented messages are addressed to objects managed by applications. 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 12-1.

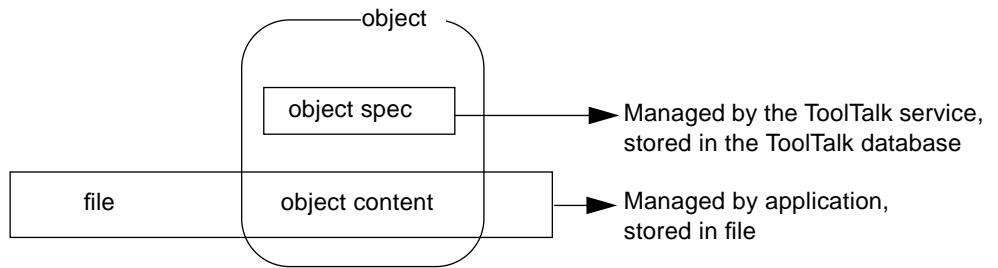


Figure 12-1 ToolTalk Object Data

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.

12.2 Creating Object Specs

To instruct the ToolTalk service to deliver messages to your objects, you create a spec that identifies the object and its otype. Table 12-1 lists the ToolTalk functions you use to create and write object spec.

Table 12-1 Functions to Create

ToolTalk Function	Description
<code>tt_spec_create(const char *filepath)</code>	Creates spec. †
<code>tt_spec_prop_set(const char *objid, const char *propname, const char *value)</code>	Sets property to specified string value. ‡
<code>tt_spec_prop_add(const char *objid, const char *propname, const char *value)</code>	Adds string property. ‡
<code>tt_spec_bprop_add(const char *objid, const char *propname, const unsigned char *value, int length)</code>	Adds byte array property. ‡
<code>tt_spec_bprop_set(const char *objid, const char *propname, const unsigned char *value, int length)</code>	Sets property to specified byte array value. ‡
<code>tt_spec_type_set(const char *objid, const char *otid)</code>	Sets object type of spec. ‡
<code>tt_spec_write(const char *objid)</code>	Writes spec to database‡

† Return type is char*.
‡ Return type is Tt_status.

To create an object spec in memory and obtain an objid for the object, use `tt_spec_create`.

Assigning Otypes

To assign an otype for the object spec, use `tt_spec_type_set`. You must set the type before the spec is written for the first time. It cannot be changed.

Note – If you create an object spec without assigning an otype or with an otype that is unknown to the ToolTalk Types Database, messages addressed to the object cannot be delivered. (The ToolTalk service does not verify that the otype you specified is known to the ToolTalk Types Database.)

Determining Object Specification Properties

You can determine what *properties* you want associated with an object; you add these properties to a spec. The ToolTalk service recognizes that it is not always possible to store information in your own internal data; for example, the objid for objects in plain ASCII text files. You can store the location of the objid in a spec property and then use this location to identify where the object is in your tool's internal data structures.

The spec properties are also a convenience for the user. A user may want to associate properties (such as a comment or object name) with the object that they can view later. Your application or another ToolTalk-based tool can search for and display these properties for the user.

Storing Spec Properties

To store properties in a spec, use `tt_spec_prop_set`.

Adding Values to Properties

To add to the list of values associated with the property, use `tt_spec_prop_add`.

12.3 Writing Object Specs

After you set the otype and add properties to an object spec, use `tt_spec_write` to make it a permanent ToolTalk item and visible to other applications. When you call `tt_spec_write`, the ToolTalk service writes the spec into the ToolTalk database.

12.4 Updating Object Specs

To update existing object spec properties, use `tt_spec_prop_set` and `tt_spec_prop_add` specifying the objid of the existing spec. Once the spec properties are updated, use `tt_spec_write` to write the changes into the ToolTalk database.

When you are updating an existing spec and the ToolTalk service returns `TT_WRN_STALE_OBJID` when you call `tt_spec_write`, it has found a forwarding pointer to the object in the ToolTalk database that indicates the object has been moved. To obtain the new `objid`, create an object message that contains the old `objid` and send it. The ToolTalk service will return the same status code, `TT_WRN_STALE_OBJID`, but updates the message `objid` attribute to contain the new `objid`. Use `tt_message_object` to retrieve the new `objid` from the message and put the new `objid` into your internal data structure.

12.5 Maintaining Object Specs

The ToolTalk service provides the functions to examine, compare, query, and move object specs. Table 12-2 lists the ToolTalk functions you use to maintain object specs.

Table 12-2 Functions to Maintain Object Specifications

Return Type	ToolTalk Function	Description
char *	<code>tt_spec_file(const char *objid)</code>	The name of the file on which the spec is located.
char *	<code>tt_spec_type(const char *objid)</code>	The object type of the spec.
char *	<code>tt_spec_prop(const char *objid, const char *propname, int i)</code>	Retrieves the <i>i</i> th (zero-based) property value as a string.
int	<code>tt_spec_prop_count(const char *objid, const char *propname)</code>	The number of values under this property name.
Tt_status	<code>tt_spec_bprop(const char *objid, const char *propname, int i, unsigned char **value, int *length)</code>	The number of byte array values under this property name.
char *	<code>tt_spec_propname(const char *objid, int i)</code>	The name of the <i>i</i> th property.
int	<code>tt_spec_propnames_count(const char *objid)</code>	The number of properties located on this spec.
char *	<code>tt_objid_objkey(const char *objid)</code>	The unique key of the spec id.
Tt_status	<code>tt_file_objects_query(const char *filepath, Tt_filter_function filter, void *context, void *accumulator)</code>	Queries the database for object specs
int	<code>tt_objid_equal(const char *objid1, const char *objid2)</code>	Checks whether two spec ids are the same.
char *	<code>tt_spec_move(const char *objid, const char *newfilepath)</code>	Moves object spec to a new file.

Examining Spec Information

You can examine the following spec information with the specified ToolTalk functions:

- Path name of the file that contains the object: `tt_spec_file`
- Otype of this object: `tt_spec_type`
- Properties stored on the spec: `tt_spec_prop` or `tt_spec_bprop`

Comparing Object Specs

To compare two objids, use `tt_objid_equal`. `tt_objid_equal` returns a value of 1 even in the case where one objid is a forwarding pointer for the other.

Querying for Specific Specs in a File

Create a filter function to query for specific specs in a file and obtain the specs in which you are interested.

Use `tt_file_objects_query` to find all the objects in the named file. As the ToolTalk service finds each object, it calls your filter function, and passes it the objid of the object and the two application-supplied pointers. Your filter function does some computation and returns a `Tt_filter_action` value (`TT_FILTER_CONTINUE` or `TT_FILTER_STOP`) to either continue the query, or to quit the search and return immediately.

Figure 12-2 illustrates how to obtain a list of specs.

```
/*
 * Called to update the scrolling list of objects for a file. Uses
 * tt_file_objects_query to find all the ToolTalk objects.
 */
int
cntl_update_obj_panel()
{
    static int list_item = 0;
    char *file;
    int i;

    cntl_objid = (char *)0;

    for (i = list_item; i >= 0; i--) {
        xv_set(cntl_ui_olist, PANEL_LIST_DELETE, i, NULL);
    }

    list_item = 0;
    file = (char *)xv_get(cntl_ui_file_field, PANEL_VALUE);
    if (tt_file_objects_query(file,
                              (Tt_filter_function)cntl_gather_specs,
                              &list_item, NULL) != TT_OK) {
        xv_set(cntl_ui_base_window, FRAME_LEFT_FOOTER,
              "Couldn't query objects for file", NULL);
        return 0;
    }

    return 1;
}
```

Figure 12-2 Obtaining a List of Specifications

Within the `tt_file_objects_query` function, the application calls `cntl_gather_specs`, a filter function that inserts objects into a scrolling list.

```

/*
 * Function to insert the objid given into the scrolling lists of objects
 * for a file. Used inside tt_file_objects_query as it iterates through
 * all the ToolTalk objects in a file.
 */
Tt_filter_action
cntl_gather_specs(objid, list_count, acc)
    char *objid;
    void *list_count;
    void *acc;
{
    int *i = (int *)list_count;

    xv_set(cntl_ui_olist, PANEL_LIST_INSERT, *i,
           PANEL_LIST_STRING, *i, objid,
           NULL);

    *i = (*i + 1);

    /* continue processing */
    return TT_FILTER_CONTINUE;
}

```

Moving Object Specs

The `objid` contains a pointer to a particular file system where the spec information is stored. To keep spec information as available as the object described by the spec, the ToolTalk service stores the spec information on the same file system as the object. Therefore, if the object moves, the spec must move, too.

Use `tt_spec_move` to notify the ToolTalk service when an object moves from one file to another (for example, through a cut and paste operation).

- If a new `objid` is not required (because both the new and old files are in the same file system), the ToolTalk service returns `TT_WRN_SAME_OBJID`.
- If the object moved to another file system, the ToolTalk service returns a new `objid` for the object and leaves a forwarding pointer in the ToolTalk database from the old `objid` to the new one.

When your process sends a message to an out-of-date objid (that is, one with a forwarding pointer), `tt_message_send` returns a special status code, `TT_WRN_STALE_OBJID`, and replaces the object attribute in the message with a new objid that points to the same object in the new location.

Note – Update any internal data structures that reference the object with the new objid.

12.6 Destroying Object Specs

Use `tt_spec_destroy` to immediately destroy an object's spec.

12.7 Managing Object and File Information



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 12-3 lists the ToolTalk functions you use to manage files that contain object data.

Table 12-3 Functions to Copy, Move, or Remove Files that Contain Object Data

ToolTalk Function	Description
<code>tt_file_move(const char *oldfilepath, const char *newfilepath)</code>	Moves the file and the ToolTalk object data
<code>tt_file_copy(const char *oldfilepath, const char *newfilepath)</code>	Copies the file and the ToolTalk object data
<code>tt_file_destroy(const char *filepath)</code>	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 12-4 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 12-4 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 path name 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.

12.8 An Example of Object-Oriented Messaging

You can run the `edit_demo` program for a demonstration of ToolTalk object-oriented messaging. This demo consists of two programs - `cntl` and `edit`. The `cntl` program uses the ToolTalk service to start an edit process with which to edit a specified file; the `edit` program allows you to create ToolTalk objects and associate the objects with text in the file. Once objects have been created and associated with text, you can use the `cntl` program to query the file for the objects and to send messages to the objects.

The following sample code 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_item    item;
    Event         *event;
{
    int           mark = tt_mark();
    char          *objid;
    char          *file;
    char          *sel;
    Textsw_index  first, last;
    char          obj_start_text[100];
    char          obj_end_text[100];
    Tt_message    msg;
```

```

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;  
}
```


Managing Information Storage

13 

To simplify your application storage management, the ToolTalk service copies all information your application provides to the ToolTalk service and also provides you with a copy of the information it returns to your application.

13.1 Information Provided to the ToolTalk Service

When you provide a pointer to the ToolTalk service, the information referenced by the pointer is copied. You can then dispose of the information you provided; the ToolTalk service will not use the pointer again to retrieve the information.

13.2 Information Provided by the ToolTalk Service

The ToolTalk service provides an allocation stack in the ToolTalk API library to store information it gives to you. For example, if you ask for the sessid of the default session with `tt_default_session`, the ToolTalk service returns the address of the character string in the allocation stack (a `char *` pointer) that contains the sessid. After you retrieve the sessid, you can dispose of the character string to clean up the allocation stack.

Note – Do not confuse the API allocation stack with your program's runtime stack. The API stack will not discard information until instructed to do so.

13.3 Calls Provided to Manage the Storage of Information

The ToolTalk service provides the calls listed in Table 13-1 to manage the storage of information in the ToolTalk API allocation stack:

Table 13-1 Managing ToolTalk Storage

Return Type	ToolTalk Function	Description
int	tt_mark(void)	Marks information returned by a series of functions.
void	tt_release(int mark)	Frees information returned by a series of functions.
caddr_t	tt_malloc(size_t s)	Reserves a specified amount of storage in the allocation stack for your use.
void	tt_free(caddr_t p)	Frees storage set aside by tt_malloc. This function takes an address returned by the ToolTalk API and frees the associated storage.

Marking and Releasing Information

The `tt_mark` and `tt_release` functions are a general mechanism to help you easily manage information storage. The `tt_mark` and `tt_release` functions are typically used at the beginning and end of a routine where the information returned by the ToolTalk service is no longer necessary once the routine has ended.

Marking Information for Storage

To ask the ToolTalk service to mark the beginning of your storage space, use `tt_mark`. The ToolTalk service returns a mark, an integer that represents a location on the API stack. All the information that the ToolTalk service subsequently returns to you will be stored in locations that come after the mark.

Releasing Information No Longer Needed

When you no longer need the information contained in your storage space, use `tt_release` and specify the mark that signifies the beginning of the information you no longer need.

Example of Marking and Releasing Information

The following code sample calls `tt_mark` at the beginning of a routine that examines the information in a message. When the information examined in the routine is no longer needed and the message has been destroyed, `tt_release` is called with the mark to free storage on the stack.

```
/*
 * 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))) {
    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;
```

Allocating and Freeing Storage Space

The `tt_malloc` and `tt_free` functions are a general mechanism to help you easily manage allotted storage allocation.

Allocating Storage Space

`tt_malloc` reserves a specified amount of storage in the allocation stack for your use. For example, you can use `tt_malloc` to create a storage location and copy the sessid of the default session into that location.

Freeing Allocated Storage Space

To free storage of individual objects that the ToolTalk service provides you pointers to, use `tt_free`. For example, you can free up the space in the API allocation stack that stores the `sessid` after you have examined the `sessid`. `tt_free` takes an address in the allocation stack (a `char *` pointer or an address returned from `tt_malloc`) as an argument.

13.4 Special Case: Callback and Filter Routines

The way that the ToolTalk service behaves toward information passed into filter functions and callbacks is a special case. Callback and filter routines called by the ToolTalk service are called with two kinds of arguments:

- Context arguments — the arguments you passed into the API call that triggered the callback. These arguments point to items owned by your application.
- Pointers to API objects — the address of message or pattern attributes in storage.

The context arguments are passed from the ToolTalk service to your application. The API objects referenced by pointers are freed by the ToolTalk service as soon as your callback or filter function returns. If you want to keep any of these objects, you must copy the objects before your function returns.

Note – The way that the ToolTalk service behaves toward information passed into filter functions and callbacks is a special case. In all other instances, the ToolTalk service stores the information in the API allocation stack until you free it.

Callback Routines

One of the features of the ToolTalk service is callback support for messages, patterns, and filters. Callbacks are routines in your program that ToolTalk calls when a particular message arrives (*message callback*) or when a message matches a particular pattern you registered (*pattern callback*).

To tell the ToolTalk service about these callbacks, add the callback to a message or pattern before you send the message or register the pattern.

Filter Routines

When you call file query functions such as `tt_file_objects_query`, you point to a filter routine that the ToolTalk service calls as it returns items from the query. For example, you could use filter routine used by the ToolTalk file query function to find a specific object. The `tt_file_objects_query` function returns all the objects in a file and runs the objects through a filter routine that you provide. Once your filter routine finds the specified object, you can use `tt_malloc` to create a storage location and copy the object into the location. When your filter function returns, the ToolTalk service will free all storage used by the objects in the file but the object you stored with the `tt_malloc` call will be available for further use.

Handling Errors

The ToolTalk service returns error status in the function's return value rather than in a global variable. ToolTalk functions return one of these error values:

- `Tt_status`
- `int`
- `char*` or opaque handle

Each return type is handled differently to determine if an error occurred. For example, the return value for `tt_default_session_set` is a `Tt_status` code. If the ToolTalk service sets the default session to the specified `sessid`:

- Without a problem — the `Tt_status` code returned is `TT_OK`.
- With a problem — the `Tt_status` code returned is `TT_ERR_SESSION`. This status code informs you that the `sessid` you passed was not valid.

14.1 Retrieving ToolTalk Error Status

You can use the ToolTalk error handling functions shown in Table 14-1 to retrieve error values.

Table 14-1 Retrieving ToolTalk Error Status

ToolTalk Function	Description
tt_pointer_error(char * return_val)	Returns an error encoded in a pointer.
tt_pointer_error((void *) (p))	Returns an error encoded in a pointer cast to VOID *.
tt_int_error(int return_val)	Returns an error encoded in an integer.

The return type for these functions is Tt_status.

14.2 Checking ToolTalk Error Status

You can use the ToolTalk error macro shown in Table 14-2 to check error values.

Table 14-2 ToolTalk Error Macros

Return Type	ToolTalk Macro	Expands to
Tt_status	tt_is_err(status_code)	(TT_WRN_LAST < (status_code))

14.3 Returned Value Status

Functions with Natural Return Values

If a ToolTalk function has a natural return value such as a pointer or an integer, a special *error value* is returned instead of the real value.

Functions with No Natural Return Values

If a ToolTalk function does not have a natural return value, the return value is an element of `Tt_status` *enum*.

To see if there is an error, use the ToolTalk macro `tt_is_err`, which returns an integer.

- If the return value is 0, the `Tt_status` *enum* is either `TT_OK` or a warning.
- If the return value is 1, the `Tt_status` *enum* is an error.

If there is an error, you can use the `tt_status_message` function to obtain the character string that explains the `Tt_status` code, as shown Figure 14-1.

```
char *spec_id, my_application_name;
Tt_status tterr;

ttterr = tt_spec_write(spec_id);
if (tt_is_err(tterr)) {
    fprintf(stderr, "%s: %s\n", my_application_name,
            tt_status_message(tterr));
}
```

Figure 14-1 Obtaining an Error Explanation

14.4 Returned Pointer Status

If an error occurs during a ToolTalk function that returns a pointer, the ToolTalk service provides an address within the ToolTalk API library that indicates the appropriate `Tt_status` code. To check whether the pointer is valid, you can use the ToolTalk macro `tt_ptr_error`. If the pointer is an error value, you can use `tt_status_message` to get the `Tt_status` character string.

Figure 14-2 checks the pointer and retrieves and prints the `Tt_status` character string if an error value is found.

```

char *old_spec_id, new_file, new_spec_id, my_application_name;
Tt_status tterr;

new_spec_id = tt_spec_move(old_spec_id, new_file);
tterr = tt_ptr_error(new_spec_id);
switch (tterr) {
    case TT_OK:
        /*
         * Replace old_spec_id with new_spec_id in my internal
         * data structures.
         */
        update_my_spec_ids(old_spec_id, new_spec_id);
        break;
    case TT_WRN_SAME_OBJID:
        /*
         * The spec must have stayed in the same filesystem,
         * since ToolTalk is reusing the spec id. Do nothing.
         */
        break;
    case TT_ERR_FILE:
    case TT_ERR_ACCESS:
    default:
        fprintf(stderr, "%s: %s\n", my_application_name,
            tt_status_message(tterr));
        break;
}

```

Figure 14-2 Retrieving a Returned Pointer Status

14.5 Returned Integer Status

If an error occurs during a ToolTalk function that returns an integer, the return value is out-of-bounds. The `tt_int_error` function returns a status of `TT_OK` if the value is not out-of-bounds.

To check if a value is out-of-bounds, you can use the `tt_is_err` macro to determine if an error or a warning occurred.

To retrieve the character string for a `Tt_status` code, you can use `tt_status_message`.

Figure 14-3 checks a returned integer.

```
Tt_message msg;
int num_args;
Tt_status tterr;
char *my_application_name;

num_args = tt_message_args_count(msg);
tt_err = tt_int_error(num_args);
if (tt_is_err(tterr)) {
    fprintf(stderr, "%s: %s\n", my_application_name,
            tt_status_message(tterr));
}
```

Figure 14-3 Checking a Returned Integer

14.6 Broken Connections

The ToolTalk service provides a function to notify processes if your tool exits unexpectedly. When you include the `tt_message_send_on_exit` call, the ToolTalk service queues the message internally until one of two events happen:

1. Your process calls `tt_close`.
In this case, the ToolTalk service deletes the message from its queue.
2. The connection between the `ttsession` server and your process is broken; for example, the application crashed.
In this case, the ToolTalk service matches the queued message to a pattern and delivers it in the same manner as if you had sent the message normally before exiting.

Your process can also send a normal message on a normal termination by calling `tt_message_send` before it calls `tt_close`. In this case, if your process sends its normal termination message but crashes before it calls `tt_close`, the ToolTalk service will deliver both the normal termination message and the `tt_message_send_on_exit` message to interested processes.

14.7 Error Propagation

ToolTalk functions that accept pointers always check the pointer passed in and return `TT_ERR_POINTER` if the pointer is an error value. This check allows you to combine calls in reasonable ways without checking the value of the pointer for every single call.

In the following code sample, a message is created, filled in, and sent. If `tt_message_create` fails, an error object is assigned to `m`, and all the `tt_message_XXX_set` and `tt_message_send` calls fail. To detect the error without checking between each call, you only need to check the return code from `tt_message_send`.

```
Tt_message m;  
  
m=tt_message_create();  
tt_message_op_set(m, "OP");  
tt_message_address_set(m, TT_PROCEDURE);  
tt_message_scope_set(m, TT_SESSION);  
tt_message_class_set(m, TT_NOTICE);  
tt_rc=tt_message_send(m);  
if (tt_rc!=TT_OK)...
```

Migrating from the Classing Engine Database to the ToolTalk Types Database



Note – In versions 1.1 and later of the ToolTalk service, `ttsession` will not read its types from the Classing Engine (CE) database; and the ToolTalk types compiler `tt_type_comp` will not merge types into the CE database.

This appendix describes how to migrate your existing ToolTalk-aware application from the CE database to the ToolTalk Types database.

A.1 The `ttce2xdr` Script

The ToolTalk service provides a script called `ttce2xdr` to convert ToolTalk types stored in the CE database (which was the default database used by versions 1.0.x of the ToolTalk service) to the XDR-format database, which is the database used by versions 1.1 and later of the ToolTalk service.

A.2 Converting the User Database

The first time a ToolTalk version 1.1.x `ttsession` is started, the `user` type database is automatically converted from the CE database to the new ToolTalk Types database. However, you can manually convert the current `user` database with the command:

```
ttce2xdr [ -xnh ] -d user
```

Table A-1 describes the options for the `ttce2xdr` script.

Table A-1 `ttce2xdr` Script Options

Option	Description
<code>-x</code>	Displays the underlying commands executed by <code>ttce2xdr</code> .
<code>-n</code>	Displays the underlying commands that <code>ttce2xdr</code> can execute.
<code>-h</code>	Describes the options for <code>ttce2xdr</code> .
<code>-d</code>	Specifies the database to be converted: <i>user</i> , <i>system</i> , or <i>network</i> .

The types are read from the CE database

```
~/.cetables/cetables
```

and written to the new ToolTalk Types database

```
~/.tt/types.xdr
```

A.3 Converting the System Database

A system CE database is the per-machine database. You will need to run the `ttce2xdr` script on each machine on which you have ToolTalk types. To determine whether there are any ToolTalk types in the system CE database, enter the following command on the command line:

```
tt_type_comp -Epd system
```

No output is generated if there are no ToolTalk types in the system CE database.

Note – You must be logged in as *root* to run the `ttce2xdr` script for the system CE databases.

To run the `ttce2xdr` script for the system CE database, enter the following commands on the command line:

```
ttce2xdr [ -xnh ] -d system
```

Table A-2 describes the options for the `ttce2xdr` script.

Table A-2 `ttce2xdr` Script Options

Option	Description
-x	Displays the underlying commands executed by <code>ttce2xdr</code> .
-n	Displays the underlying commands that <code>ttce2xdr</code> can execute.
-h	Describes the options for <code>ttce2xdr</code> .
-d	Specifies the database to be converted: <i>user</i> , <i>system</i> , or <i>network</i> .

The types are read from the CE database

```
/etc/cetables/cetables
```

and written to the new ToolTalk Types database

```
/etc/tt/types.xdr
```

A.4 Converting the Network Database

A *network* CE database is the per-OW-installation database. You need to convert each network CE database that has ToolTalk types other than those shipped with the OpenWindows Version3™ product.

Note – You must be logged in as *root* to run the `ttce2xdr` script for the network CE databases.

To convert a network-wide database, enter the following command on the command line:

```
ttce2xdr [ -xnh ] -d network [ OPENWINHOME-from [ OPENWINHOME-to ] ]
```

Table A-3 describes the options for the `ttce2xdr` script.

Table A-3 `ttce2xdr` Script Options

Option	Description
<code>-x</code>	Displays the underlying commands executed by <code>ttce2xdr</code> .
<code>-n</code>	Displays the underlying commands that <code>ttce2xdr</code> can execute.
<code>-h</code>	Describes the options for <code>ttce2xdr</code> .
<code>-d</code>	Specifies the database to be converted: <i>user</i> , <i>system</i> , or <i>network</i> .
<code>OPENWINHOME-from</code>	Reads the types from the databases under this directory. If <code>OPENWINHOME-to</code> is set, the types are written to the databases under that specified directory; otherwise, the current value of the environment variable <code>OPENWINHOME</code> is used to locate the databases to which the types are written.
<code>OPENWINHOME-to</code>	Writes the types to the databases under this directory. If <code>OPENWINHOME-from</code> is set, the types are read from the databases under that specified directory; otherwise, the current value of the environment variable <code>OPENWINHOME</code> is used to locate the databases from which the types are read.

The types are read from the CE database

```
$OPENWINHOME/lib/cetables/cetables
```

and written to the new ToolTalk Types database

```
$OPENWINHOME/etc/tt/types.xdr
```

To move ToolTalk types other than the ones shipped with the OpenWindows Version 3 product from the network CE database to the network XDR database, enter the following command on the command line:

```
ttce2xdr -d network <old_OPENWINHOME> <new_OPENWINHOME>
```

where `<old_OPENWINHOME>` is the OpenWindow installation that holds the old network CE database and `<new_OPENWINHOME>` is the OpenWindow installation in which to update the ToolTalk XDR database.

The ToolTalk Standard Message Sets



Standard message sets help developers to develop applications that will automatically integrate with applications developed by others that follow the same message protocols. Extensive work has been done with leading software suppliers and end-users to define standard message sets. The ToolTalk Standard Message Sets are higher-level interfaces of the ToolTalk API that provide common definitions and conventions to easily achieve control and data integration between applications.

See the *ToolTalk Reference Manual* for a complete description of the standard ToolTalk message sets.

C.1 The ToolTalk Desktop Services Message Set

In order to achieve basic desktop integration, applications need to support a basic set of messages to enable inter-application control. The *ToolTalk Desktop Services Message Set* is the common message set that provides this functionality for all applications. A powerful messaging protocol that benefits both developers and users of desktop applications, the ToolTalk Desktop Services Message Set allows applications to easily interact with other desktop applications. Using the ToolTalk Desktop Services Message Set, applications can communicate with each other in a transparent manner, both locally and over networks.

Why the ToolTalk Desktop Services Message Set was Developed

In order to provide integrated control of applications, 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 inter-change basic control information. This kind of functionality enables the development of smart desktops and integrated smart toolsets. Groups of applications can now call upon each other to perform tasks and to interact as one solution environment for the end-user.

Key Benefits of the ToolTalk Desktop Services Message Set

The ToolTalk Desktop Services Message Set offers developers two key benefits:

1. Allows basic control of applications without direct intervention from the user. Routine or common procedures may be automated for the convenience of the user.
2. Allows tool specialization through a common set of interactions. All ToolTalk aware applications can perform these functions.

C.2 The ToolTalk Document and Media Exchange Message Set

Multimedia is an important emerging technology. While the base of multimedia-aware applications has expanded, no single vendor provides a completely integrated solution which meets the complex needs of today's market. The *ToolTalk Document and Media Exchange™ Message Set* is a genuine breakthrough in multimedia technologies. A powerful messaging protocol designed to benefit both developers and users of multimedia technologies, the ToolTalk Document and Media Exchange Message Set allows applications to easily share each others multimedia functionality. Using the ToolTalk Document and Media Exchange Message Set, multimedia applications can communicate with each other in a transparent manner, both locally and over networks, regardless of data formats, compression technology, and other technical issues which has previously confined the use of this technology.

Why the ToolTalk Document and Media Exchange Message Set was Developed

While a few vendors have established inter-operability alliances, the range of possible end-user solutions has been restricted. The ToolTalk Document and Media Exchange Message Set allows any application to share a set of multimedia functions with any other application in a transparent manner.

This document contains specifications that have been developed by an alliance of designers from key independent multimedia hardware and software vendors, SunSoft™, and the Sun Microsystems® Computer Corporation (SMCC)™. Applications that use these simple protocols can quickly and easily create a ToolTalk™ interface to an array of multimedia services without concern for a particular service provider. Entire groups of applications can now *plug-and-play* together, integrating sound, video, graphics, telephony, and other media sources into new and exciting applications. The term *plug-and-play* means that any tool can be replaced by any other tool that follows the same protocol. That is, any tool that follows a given ToolTalk protocol can be placed (plugged) into your computing environment and perform (play) those functions indicated by the protocol. Tools can be mixed and matched, without modification and without having any specific built-in knowledge of each other. For example, you could create a word processing application that integrates a piece of video into a composition and have the video played by another application.

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.

Key Benefits of the ToolTalk Document and Media Exchange Message Set

The ToolTalk Document and Media Exchange Message Set offers developers two key benefits:

1. Ease of multimedia integration to new and existing software.

Adding multimedia functionality to any application is now vastly simplified. The ToolTalk Document and Media Exchange Message Set allows you to use other developers' multimedia technologies, thus reducing your development time and expenses while increasing your system functionality.

2. Creates a framework that extends the range of end-user solutions.

By facilitating application inter-operability, the ToolTalk Document and Media Exchange Message Set allows end-users and other developers to create new vertical solutions. These solutions, in turn, create new opportunities for your products by opening markets that were previously beyond their scope.

C.3 General ToolTalk Message Definitions and Conventions

In the ToolTalk messages there are terms used with specific ToolTalk definitions. This section defines these terms and conventions used in the ToolTalk message man pages.

Table C-1 Document and Media Exchange Message Set Descriptions

Type of Information	Description
header	A single line that describes the message in the following format: <i>MsgName(Tt_class)</i> where <i>MsgName</i> is the name of the message and <i>Tt_class</i> is either Request or Notice.
name	The name of the message and a one-line description of the message.
description	An explanation of the operation (event) that the message requests (announces).

Table C-1 Document and Media Exchange Message Set Descriptions (Continued)

Type of Information	Description
synopsis	<p>A representation of the message in the ToolTalk types-file syntax (similar to the syntax understood by the ToolTalk type compiler <code>tt_type_comp</code>) in the following format: <code><fileAttrib> <opName> (<requiredArgs> [<optionalArgs>]);</code> A synopsis entry is given for each interesting variant of the message.</p> <p><code><fileAttrib></code> - An indication of whether the file attribute of the message can/should be set.</p> <p><code><opName></code> - The name of the operation or event is called the “op name” (or “op”). It is important that different tools not use the same <code>opName</code> to mean different things. Therefore, unless a message is a standard one, its <code>opName</code> should be made unique. A good way to do this is to prefix it with: <code><Company><Product></code> e.g., “Acme_Hoarktool_My_Frammistat”.</p> <p><code><requiredArgs></code>, <code><optionalArgs></code> - The arguments that must always be included in the message. A particular argument is described in the following format: <code><mode> <vtype> <argument name></code> where <i>mode</i> is one of “in”, “out”, or “inout”, <i>vtype</i> is a programmer-defined string that describes what kind of data a message argument contains; and <i>argument name</i> is the name of the argument.</p> <p>The ToolTalk service uses vtypes to match sent message instances with registered message patterns. By convention, a <code>vtype</code> maps to a single, well-known data type.</p>

Table C-1 Document and Media Exchange Message Set Descriptions (Continued)

Type of Information	Description
required arguments	<p>The arguments that must always be in the message.</p> <p style="text-align: center;"><code><vtype> <argumentName></code></p> <p>A description of a particular argument.</p> <p>A 'vtype' is a programmer-defined string that describes what kind of data a message argument contains. ToolTalk uses vtypes for the sole purpose of matching sent message instances with registered message patterns.</p> <p>Every vtype should by convention map to a single, well-known data type. The data type of a ToolTalk argument is either integer, string, or bytes. The data type of a message or pattern argument is determined by which ToolTalk API function is used to set its value.</p> <p>The argument name is merely a comment hinting to human readers at the semantics of the argument, much like a parameter name in a C typedef.</p>
optional arguments	<p>The extra arguments that may be included in a message. Unless otherwise noted, any combination of the optional arguments, in any order, may be appended to the message after the required arguments.</p>
description	<p>An explanation of the operation that the request entails, or the event that the notice announces.</p>
errors	<p>A list of the error codes that can be set by the handler of the request (or the sender of the notice).</p>

Edict

An *edict* is a notice that looks like a request. If a request returns no data (or if the sender does not care about the returned data), it can sometimes be useful to broadcast that request to a set of tools. Since the message is a notice, no data is returned, no replies are received, and the sender is not told if any tool gets the message.

Handler

The *handler* is the distinguished recipient procid of a request. This procid is responsible for completing the indicated operation.

Notice

A *notice* is a message that announces an event. Zero or more tools may receive a given notice. The sender does not know whether any tools receive its notice. A notice cannot be replied to.

Procid

A *procid* is a principal that can send and receive ToolTalk messages. A procid is an identity, created and handed over by the ToolTalk service on demand (via `tt_open`), that a process must assume in order to send and receive messages. A single process can use multiple procsids; and a single procid can be used by a group of cooperating processes.

Request

A request is a message that asks an operation to be performed. A request has a distinguished recipient, called a handler, who is responsible for completing the indicated operation. A handler may fail, reject, or reply to a request. Any number of handlers may reject a request but ultimately only one handler can fail it or reply to it. If no running handler can be found to accept a request, the ToolTalk service can automatically start a handler. If no willing handler can be found, or if a handler fails the request, then the request is returned to the sender in the ‘failed’ state.

C.4 Errors

A `Tt_status` code can be read from a reply via `tt_message_status`. This status defaults to `TT_OK`, or can be set by the handler via `tt_message_status_set`. In extraordinary circumstances (such as no matching handler) the ToolTalk service itself sets the message status.

In addition to the `Tt_status` values defined by the ToolTalk API, the overview reference page for each set of messages lists the error conditions defined for that set of messages. For each error condition, the overview reference page provides

- Its name
- Its integer value
- A string in the “C” locale that explains the error condition

Since the ToolTalk Inter-Client Conventions (TICC) are a binary message interface, the integer and string are part of that binary interface; the name is not.

- The string may be used as a key in the `SUNW_TOOLTALK_INTERCLIENTCONVENTIONS` domain to retrieve a localized explanation of the error condition. See `dgettext(3)`.
- The integer values of these status codes begin at 1537 (`TT_ERR_APPFIRST + 1`). The first 151 codes correspond to the system error list defined in `intro(2)`.

A standard programming interface for these conventions that binds the name to the integer value does not yet exist.

The ToolTalk service allows an arbitrary status string to be included in any reply. Since a standard localized string can be derived for each status code, this status string may be used as a free-form elucidation of the status. For example, if a request is failed with `TT_DESKTOP_EPROTO`, the status string could be set to "The vtype of argument 2 was 'string'; expected 'integer'". Handling tools should try to compose the status string in the locale of the requestor. See the `Get_Locale` request.

C.5 General ToolTalk Development Guidelines and Conventions

Sun Microsystems, Inc. encourages *open protocols*. A protocol is open largely to the extent that it contains *anonymous message* (that is, messages that are sent without knowledge of who is to receive them). This section provides guidelines to help you independently develop applications that will successfully interact with any other application that supports the message protocol. These guideline and principles help ensure that two independently-developed applications will be able to initiate and maintain conventions; and, thus, interact with each other. By following these guidelines, you will enable users of your application to better control and customize their environment.

When you write a ToolTalk application, you need to follow these principles:

1. Always make requests anonymous.
2. Let tools be started as needed.
3. Reply to a request only when the requested operation has been completed.

4. Avoid statefulness whenever possible.
5. Declare one ptype for each role a tool can play.

Always Make Anonymous Requests

To design your application to be completely open, you want the requests to be completely anonymous. That is, the requesting process has no knowledge of which tool instance — or even which tool type — will perform the requested operation. If the requests are sent to a specific process, you unnecessarily restrict how users or potential message recipients can utilize their resources. If the requests are sent to a specific tool type, you unnecessarily restrict the other kinds of tools that can interact with your tool.

You want your message to describe the operation being requested or the event being reported. You do not want your message to describe the process that should receive the message. The less specific knowledge each tool encodes about the tools with which it will interact, the more flexible the overall system is for the user.

For more information about open protocols, see “Designing and Writing a ToolTalk Procedural Protocol” (SunSoft Part Number 801-3592-01).

Let Tools Be Started as Needed

To design your protocol to be completely open, you want the system to start tools only as needed. When you let a new tool instance be started only as needed, you provide the user with more flexibility and more efficient use of resources such as CPU, screen real estate, and swap space. The ToolTalk service has several features that assume the responsibility of determining when to start a new tool instance:

- The ToolTalk service allows messages and type signatures to have “start” reliability. Start reliability means that if no eligible recipient of a message is running (or willing to accept the request), the ToolTalk service will start an instance of the type of tool which is statically registered to handle or observe that message.
- The ToolTalk service allows each process type (*ptype*) to specify the maximum number of its instances that may be started in a given session.

- The ToolTalk service offers each request to all eligible running handlers before it starts a new tool instance. An eligible handler can accept or reject a request based on its own criteria (such as its ability to take on a new task; whether or not it has unsaved changes; idle time; iconic state; or whether or not the user has indicated that the tool is free to accept new work).

Reply When Operation has been Completed

To design your application to be completely open, you want to notify the sending process that its requested operation has been performed. However, the operation invoked by a request sometimes takes a relatively long time to complete compared to the very brief time it takes to send the message. Since the sending process is expecting a reply, your tool can respond in two ways:

1. It can reply immediately that it has received the request and then convey the actual results of the completed operation in a later message.
2. It can withhold the reply until the operation has been completed.

We recommend the second policy because ToolTalk messaging is entirely asynchronous: neither a tool (nor the session it is in) is blocked because it has one or more requests outstanding.

Avoid Statefulness Whenever Possible

To design your application to be open, you want each message to make sense by itself whenever possible. When a protocol is stateless, the messages in it avoid dependency on any previous messages or on some state in the assumed recipient.

Declare One Process Type per Role

A ToolTalk protocol is expressed in terms of the *roles* that each tool plays (that is, the kinds of tasks each tool is assigned to perform). A ToolTalk ptype essentially instructs the ToolTalk service how to handle any messages in which a tool is interested that are sent when that tool is not running. To design your protocol to be open, you want to declare one ptype for each role in your protocol. When you declare only one ptype per role in your protocol, you

provide users with the flexibility to interchange tools as their needs require. For example, a user may want a sophisticated sound-authoring tool for recording but also prefers a simple audio tool to perform the playback.

Thus, you will sometimes want to include only one message signature per ptype. When you include more than one message signature in the same ptype, you are requiring that any program that can handle one message can handle the other messages. For example, a ptype “UWriteIt” can include the two message signatures “Display” and “Edit” because it is expected that any tool that understands the UWriteIt document format can perform both of these operations.

C.6 Developing ToolTalk Applications

Developing ToolTalk aware-applications is a design process. You can enable your application to send and receive ToolTalk messages in a simple three-step process:

1. Determine how your application is to interact with other applications, and with users.
2. Select messages and define their use within the context of your application
3. Integrate ToolTalk calls and messages into your code.

Note – A demonstration of how you can easily add ToolTalk capability to your existing applications has been integrated with the ToolTalk software product. This demonstration is described in the paper entitled *Tool Inter-Operability: A Hands On Demonstration* (SunSoft Part Number 801-3593-01) and is part of the ToolTalk information pack available from SunSoft or your local Sun Sales office.

♦ **Define how the tools will work together and what operations must be performed.**

A clear understanding of what types of communications your application will require is a critical factor in successful application integration. The best approach to analyze this issue is to define scenarios that represent how your application will be used. From these scenarios you will be able to determine what interaction needs to take place and what information needs to be

exchanged. Detailed scenarios that show exactly what information and status is being passed will greatly help you integrate messaging into your application.

♦ **Select the appropriate messages that accomplish these tasks.**

Once you have determined how your applications will interact with other applications and users, you must determine the specific messages needed to accomplish the required tasks.

First, look at the standard message sets available from industry groups such as SunSoft, ANSI, X3H6, and CFI. Use of these messages is strongly recommended for two reasons.

1. The standard messages provide your application with a well-known and documented interface. This interface allows other developers to independently develop applications that can interface with your work. In addition, it provides an interface around which your customers can build integrated systems.
2. The standard message sets provide your application with the “universal plug-and-play” capability. This capability allows you to provide your customers with the flexibility to use multiple applications to provide a service. By giving your customers a choice of applications to use, they can pick the best tool for a particular job and you are not forced to offer features that you feel your product does not need.

If the standard message sets do not support your design, then you will need to develop custom messages.

If you use non-standard message, please contact the Document and Media Exchange Messaging Alliance at media_exchange@Sun.Com so we can consider adding your new messages to the standard message sets.

♦ **Integrate ToolTalk calls and messages into your application.**

Once you have completed the design aspect, you are ready to add the ToolTalk capabilities into your application.

First, you need to include the ToolTalk header file in all files that will use ToolTalk API calls. You will also need to register and initialize the patterns that control the sending and receiving functions. For detail information about

registering and initializing patterns, see the book entitled “The ToolTalk Service: An Inter-Operability Solution.” (The book is available in bookstores, through SunSoft Press, and directly from Prentice Hall.)

Next, add the ability to send ToolTalk messages to your code. Based on the knowledge gained from designing the scenarios, it is very straight forward to determine what routines need to send what messages, and what the arguments for each message should be.

Once the ToolTalk service is initialized, your application uses the ToolTalk API calls to create and fill in messages to be sent to other applications.

- If your application uses a windowing system, you only need to add the calls to activate the ToolTalk service in the event polling loop.
- If your application does not already use a polling loop, you need to create a simple loop that periodically checks for messages. For detailed information, see the book entitled “The ToolTalk Service: An Inter-Operability Solution.” (The book is available in bookstores, through SunSoft Press, and directly from Prentice Hall.)

C.7 Messaging Alliances

Send questions, comments, and requests for information to the Desktop Services Messaging Alliance at *ToolTalk_desktop_services@sun.com*.

Send questions, comments, and requests for information to the Document and Media Exchange Messaging Alliance at *media_exchange@Sun.Com*.



Frequently Asked Questions



This appendix contains answers to the following questions about the ToolTalk service:

<i>What is the ToolTalk service</i>
<i>Is the ToolTalk Service the SunSoft implementation of the Common Object Request Broker Architecture (CORBA)</i>
<i>What files are part of the ToolTalk service</i>
<i>Where is the initial X-based tsession started</i>
<i>Where is rpc.ttdbserverd started</i>
<i>Where are the ToolTalk type databases stored</i>
<i>Do I need X Windows to use the ToolTalk service</i>
<i>Can I use the ToolTalk service with MIT X</i>
<i>Where is the session id of the X-session</i>
<i>How does tt_open connect to a tsession</i>
<i>After calling tt_open, when does a session actually begin</i>
<i>If another session is attached, does the first session get killed</i>
<i>How can processes on different machines communicate using the ToolTalk service</i>
<i>What is the purpose of tt_default_session_set</i>
<i>How can a process connect to more than one session</i>
<i>Can you start a tsession with a known session id</i>
<i>What information does a session id contain</i>
<i>Is there a standard way to announce that a new program has joined a session</i>

<i>Where is my message going</i>
<i>What is the basic flow of a message</i>
<i>What happens when a message arrives to my application</i>
<i>How can I differentiate between messages</i>
<i>Can a process send a request to itself</i>
<i>Can I pass my own data to a function registered by <code>tt_message_callback_add</code></i>
<i>How can I send arbitrary data in a message</i>
<i>Can I transfer files with the ToolTalk service</i>
<i>How are memory (byte) ordering issues handled by the ToolTalk service</i>
<i>Can I re-use messages</i>
<i>What happens when I destroy a message</i>
<i>Can I have more than one handler per message</i>
<i>Can I run more than one handler of a given ptype</i>
<i>What value is disposition in a message</i>
<i>What are the message status elements</i>
<i>When should I use <code>tt_free</code></i>
<i>What does the ptype represent</i>
<i>Why are my new types not recognized</i>
<i>Can I declare a ptype that is not in the types database</i>
<i>Is ptype information used if a process of that ptype already exists</i>
<i>Can the ptype definition be modified to always start an instance (whether or not one is already running)</i>
<i>What does <code>tt_ptype_declare</code> do</i>
<i>What is <code>_SUN_TT_TOKEN</code></i>
<i>When are my patterns active</i>
<i>Must I register patterns to get replies</i>
<i>How can I observe requests</i>
<i>How do I match to attribute values in static patterns</i>
<i>Why am I unable to wildcard a pattern for <code>TT_HANDLER</code></i>
<i>Can I set a pattern to watch for any file scoped message</i>
<i>Is file scope in static patterns the same as <code>file_in_session</code> scope</i>
<i>What is the difference between <code>arg_add</code>, <code>barg_add</code>, and <code>iarg_add</code></i>
<i>What is the type or vtype in a message argument</i>

<i>How do I use contexts</i>
<i>How does ttsession check for matches</i>
<i>How many kinds of scope does the ToolTalk service have</i>
<i>What are the tt_db directories, and what is the difference between the types database and the tt_db directories</i>
<i>What should the tt_db databases contain</i>
<i>What does rpc.ttdbserverd do</i>
<i>Do ttsession and rpc.ttdbserverd ever communicate</i>
<i>What message bandwidth can be supported</i>
<i>Is there a limit to the message size or the number of arguments</i>
<i>What is the most time efficient method to send a message</i>
<i>What network overhead is involved</i>
<i>Does the ToolTalk service use load balancing to handle requests</i>
<i>What resources are required by a ToolTalk application</i>
<i>What happens if the ttsession exits unexpectedly</i>
<i>What happens if rpc.ttdbserverd exits unexpectedly</i>
<i>What happens if a host or a link is down</i>
<i>What does tt_close do</i>
<i>Is message delivery guaranteed on a network</i>
<i>Is there a temporal sequence of message delivery</i>
<i>What is unix, xauth, and des</i>
<i>Can my applications hide messages from each other</i>
<i>Is there protection against interception or imitation</i>
<i>Where are queued messages stored and how secure is the storage</i>
<i>Is the ToolTalk service C2 qualified</i>
<i>How can I trace my message's progress</i>
<i>How can I isolate my debugging tool from all the other tools using the ToolTalk service</i>
<i>Can I use the ToolTalk service with C++</i>
<i>Should I qualify my filenames</i>
<i>Can you tell me about ToolTalk objects</i>
<i>Is there a ToolTalk news group</i>

▼ **What is the ToolTalk service?**

The ToolTalk service enables independent applications to communicate with each other without having direct knowledge of each other. Applications create and send ToolTalk messages to communicate with each other. The ToolTalk service receives these messages, determines the recipients, and then delivers the messages to the appropriate applications.

▼ **Is the ToolTalk Service the SunSoft implementation of the Common Object Request Broker Architecture (CORBA)?**

No. The ToolTalk service is not the SunSoft CORBA-compliant Object Request Broker (ORB). The ToolTalk service was designed and shipped in 1991 — before the Object Management Group (OMG) CORBA specification was defined.

The SunSoft CORBA-compliant ORB is the Distributed Object Management Facility (DOMF), which is part of the SunSoft Project DOE product. SunSoft is publicly committed to support the ToolTalk API running on the DOMF when the DOMF becomes generally available as part of Solaris. Applications that use the ToolTalk messaging service today will transition to the distributed object environment of the future.

▼ **What files are part of the ToolTalk service?**

The ToolTalk files are normally found in the `/usr/openwin/bin`, `lib`, `share/include/desktop`, and `man/man1` directories. Table D-1 describes the files.

Table D-1 ToolTalk Files

File Name	Description
<code>ttsession</code>	Communicates on the network to deliver messages.
<code>rpc.ttdbserverd</code>	Stores and manages ToolTalk object specs and information on files referenced in ToolTalk messages.
<code>ttcp</code> , <code>ttmv</code> , <code>ttrm</code> , <code>ttrmdir</code> , <code>tttar</code>	Standard operating system shell commands. These commands inform the ToolTalk service when files that contain ToolTalk objects or files that are the subject of ToolTalk messages are copied, moved, or removed.

Table D-1 ToolTalk Files (Continued)

File Name	Description
ttdbck	Database check and recovery tool for the ToolTalk databases.
tt_type_comp	Compiles the ptype and otype files, and automatically installs them in the ToolTalk Types database.
ttce2xdr	Converts ToolTalk type data from the Classing Engine database format to the XDR database format.
libtt.a, libtt.so, and tt_c.h	Application programming interface (API) libraries and header file that contain the ToolTalk functions used by applications to send and receive messages.

▼ Where is the initial X-based ttsession started?

The first call to `tt_open` automatically starts `ttsession` if no `ttsession` is running. However, the `$OPENWINHOME/lib/openwin-sys` file may contain an entry such as the following:

```
# Start ttsession here.
ttsession -s
```

▼ Where is `rpc.ttdbserverd` started?

The `/etc/inet/inetd.conf` file contains an entry similar to the following:

```
# Sun ToolTalk Database Server
100083/1 stream rpc/tcp wait root /usr/openwin/bin/rpc.ttdbserverd rpc.ttdbserverd
```

▼ Where are the ToolTalk type databases stored?

The environment variable `TTPATH` tells the ToolTalk service where the ToolTalk Types databases reside. The format of this variable is:

```
userDB[:systemDB[:networkDB]]
```

Note – The type files are read in reverse order of `TTPATH`.

This environment variable also tells the ToolTalk service where to search for database server redirection files. The default locations are listed in Table D-2.

Table D-2 Default Locations of ToolTalk Types Database

Database	Location
user	~/.tt
system	/etc/tt
network	\$OPENWINHOME/etc/tt

▼ Do I need X Windows to use the ToolTalk service?

The ToolTalk service does not use X messages or protocols to deliver messages. The ToolTalk service is only associated with X Windows if you run an X session.

When you run an X session, the session name is advertised as a property (named `_SUN_TT_SESSION`) on the root window of the X server. Every process which names that X server as its display gets that X session as its default session. Since the X session is defined to be the group of processes displaying on a particular X display, you do need to run X Windows by definition but *not* because the ToolTalk service requires you to use it.

If there is no X server running at all (for example, you are running a session that consists entirely of character-mode applications running on a dumb terminal), use a *process tree session*. When you run a process tree session, the session name is advertised in the environment variable `_SUN_TT_SESSION`. This session is the default session for every process in the tree of processes descending from the process that set the environment variable.

▼ Can I use the ToolTalk service with MIT X?

Yes. However, the `LD_LIBRARY_PATH` must point to `$OPENWINHOME/lib` for the `libtt.so` file.

▼ Where is the session id of the X-session?

To get this identifier, enter the following command:

```
xprop -root | grep _SUN_TT_SESSION
```

Note - An X session is a session that advertises its session id on the `_SUN_TT_SESSION` property of root window.

▼ How does `tt_open` connect to a `ttsession`?

After some internal initialization, `tt_open` tries to find a `ttsession`.

1. `tt_open` checks whether the environment variable `_SUN_TT_SESSION` is set.
If this environment variable is set, it uses the value as the id of the `ttsession`.

If this environment variable is not set, it checks to see if the `DISPLAY` environment variable is set.

- If this environment variable is set, it uses the value as the id of the `ttsession`.
- If this environment variable is not set, it checks to see if the `_SUN_TT_SESSION` property on root X window (of the machine running the display) is set.

In the event that none of these environment variables are set, it will start a `ttsession` itself.

2. `tt_open` 'pings' the `ttsession` to make sure it is active.
3. `tt_open` checks the environment variable `_SUN_TT_TOKEN` to determine whether the client was started from a 'start' command for the `pty`. Once the start `pty` is determined, it creates a `procid`.

4. `tt_open` creates a TCP/IP socket on the client side to which `ttsession` connects.

Activity on the socket is noticed via the socket's associated file descriptor. `ttsession` only uses this channel to notify the client of incoming messages.

Note – Call `tt_close` on this file descriptor; do *not* call the `close` function. If you call the `close` function on the file descriptors returned by `tt_fd`, your file descriptor count will rise upon successive `tt_open` and `close` calls.

5. `tt_open` refreshes the database hostname redirection map.

▼ **After calling `tt_open`, when does a session actually begin?**

If the default session is an X session and there is no `ttsession` running, `libtt` starts one; otherwise, the `ttsession` must be started first in order to get the session name.

▼ **If another session is attached, does the first session get killed?**

No. The first session will still be running.

▼ **How can processes on different machines communicate using the ToolTalk service?**

There are two ways in which processes on different machines can communicate using the ToolTalk service.

1. They can connect to the same session.
2. They can scope to a file that is NFS mounted on the machines involved.

Connecting to the Same Session

To connect the processes to the same session, you first need to determine a common interest for the processes (for example, a scheme that associates a session name with the common interest of the processes) and then you need to determine how to propagate the session name to all of the processes. The ToolTalk service does not provide a mechanism to distribute the session address (other than the possible advertisement of a session id on the `_SUN_TT_SESSION` property of the root windows of X servers).

To get a session name, you can use the command

```
ttsession -p
```

which forks off a new session and prints its name to stdout; or you can the command:

```
ttsession -c
```

which sets the environment variable `$_SUN_TT_SESSION` to the session id.

You then need to use some mechanism to put that session name in a place where the other processes can find it. Some examples of where you can place the session name are:

- a shared file
- a .plan file
- a mail message
- a separate RPC call of your own design
- NIS

For example, one approach using a well-known file in a NFS-exported file system can be done as follows:

- a. Start `ttsession` with the following command:

```
ttsession -p >/home/foo/sessionaddress
```

- b. Ensure that the clients use the session address from the file; for example, wrap the clients in a shell script which reads the session address and sets `SUN_TT_SESSION` as follows:

```
#!/bin/csh
setenv _SUN_TT_SESSION `cat /home/foo/sessionaddress`
exec client-program
```

Alternately, the processes can use the session name in the `tt_default_session_set` call to connect to that session.

You could also send messages in the `ttsession` associated with a particular X server to advertise the newly-created `ttsessions`.

Scoping to a NFS-mounted File

File scoping is when a process registers a file scope pattern. The name of that session is placed on a list in `rpc.ttdbserverd` that is associated with the registered file. When a file-scoped message is sent, the ToolTalk service retrieves the list of sessions for the file and forwards the message to each of the sessions on the `rpc.ttdbserverd` list for that file.

Note – To scope to a file that is NFS-mounted on the machines involved requires a file system to be NFS mounted on all the systems and `rpc.ttdbserverd` to be run on the NFS server.

▼ **What is the purpose of `tt_default_session_set`?**

`tt_default_session_set` determines the `ttsession` to which a call to `tt_open` will connect.

▼ **How can a process connect to more than one session?**

Table D-3 describes several default variables that are used when communicating with the ToolTalk service.

Table D-3 Some Default Variables

Variable	Description
<code>procid</code>	Set by <code>tt_open</code> . This variable identifies the client to <code>ttsession</code> .
<code>ptype</code>	Set by <code>tt_ptype_declare</code> .
<code>file</code>	Set when you join a file. If no file is set in the message, the file attribute is set to the default file.

If you use the API functions for getting and setting the procid, your application can switch between multiple sessions. For example,

```
connect to session 1
store the default procid in filename
connect to session 2,
store the default procid filename
restore associated default procid
interact with particular_session
```

Note – The default file and ptype are part of the current default procid. Changing the default procid also changes the default file and ptype to the default file and ptype associated with that procid.

▼ **Can you start a tsession with a known session id?**

No. You have to get the session id from the ToolTalk service.

▼ **What information does a session id contain?**

The session id consists of a number of fields, including:

- Version of address format
- Unix pid of process
- RPC Transient Program Number
- Unused version (compatibility holdover)
- Authorization level
- User id
- Host IP address
- RPC version



Caution – The format of a session id is a private interface. Do *not* write ToolTalk clients that depend on the format of a session id.

▼ **Is there a standard way to announce that a new program has joined a session?**

Broadcast a notice message to notify interested processes when a new process joins a session. To observe notice messages, a process that want to be notified if a new process joins a session must register patterns to observe these notices.

Note – The Desktop Services "Started" message was developed for this purpose.

▼ **Where is my message going?**

Use the `-t` (trace mode) at start-up to observe how `ttsession` processes each message you send. You can also toggle the trace mode on and off by sending `ttsession` a `USR1` signal; for example:

```
kill -USR1 <ttsession_pid>
```

Alternatively, you can use the `ttsnoop` utility to monitor a message with very general patterns.

▼ **What is the basic flow of a message?**

Session-Scoped Message Flow

The basic flow of a session-scoped message is as follows:

1. The client builds request message and calls `tt_message_send`.
2. `ttsession` finds a handler.
The environment variable `_SUN_TT_TOKEN` is set by `ttsession` when it starts the handler.
3. The handler starts up and calls `tt_open` and `tt_fd` to establish communication to `ttsession`.
4. The handler declares its `ptype` to `ttsession`.
5. `ttsession` changes all the static patterns for the `ptype` into dynamic patterns.
At this point, the patterns are not active because the handler has not yet joined the session.

-
6. The handler joins session, activating patterns.
 7. `ttsession` notifies the handler that a message is queued.
 8. The handler notices activity on the file descriptor and calls `tt_message_receive` to retrieve the message.

If the message returned by `tt_message_receive` has the status `TT_WRN_START_MESSAGE`, the ToolTalk service started the process to deliver the message. In this case, messages for the `p`type are blocked until the process either replies, rejects, or fails the message (even if it is a notice), or calls `tt_message_accept`.

9. The handler performs the requested operation.
10. The handler returns a reply to request.
11. `ttsession` notifies the client that a (reply) message for it is in the queue. The client's file descriptor is activated.

Note – The client actually receives a message every time its request message changes state.

12. The client calls `tt_message_receive` to retrieve the result.

File-Scoped Message Flow

The basic flow of a file-scoped message is as follows:

1. A file-scoped pattern is registered.
`libtt` notifies the database server about the file and the session in which it is registering the pattern.
2. `libtt` checks with the database server to find all the sessions that have clients who have registered interest in the specified file.
 - For notices, it communicates with all these sessions directly.
 - For requests, it notifies its session about the message and the list of other sessions involved.
3. The sessions communicate amongst each other to find a handler.

▼ What happens when a message arrives to my application?

When a message arrives to your application, the following occurs:

1. The file descriptor becomes active.
2. The Xt main loop breaks out of its select and calls the function registered by the `XtAppAddInput` call.
3. The registered function calls `tt_message_receive`.
The message is read in and any callbacks associated with the message are run.
4. The message callback returns.
 - If the message callback returns `TT_CALLBACK_PROCESSED`, `tt_message_receive` returns a value of null to the input callback.
 - If the message callback returns `TT_CALLBACK_CONTINUE`, a `Tt_message` handle for the message is returned.
5. The input callback continues with any other processing.

For example, the following input callback:

```
input_callback(...)
{
    Tt_message m;
    printf ("input callback entered\n");
    m = tt_message_receive();
    printf ("input callback exiting, message handle is %d\n",
           (int)m);
}
```

and the following message callback:

```
message_callback(...)
{
    printf("message callback entered\n");
    return TT_CALLBACK_PROCESSED;
}
```

results in the following output:

```
input callback entered
message callback entered
input callback exiting, message handle is 0
```

▼ How can I differentiate between messages?

You can differentiate between messages as follows:

- Each message has an identifier that uniquely identifies the message across all running tsessions.
- You can use the `tt_message_user` call to include information on a user cell to associate the message to the application's internal state.
- Message handles remain the same. For example, Code Example D-1 tells you whether the message you received is the same as the message you sent.

Code Example D-1 Differentiating Between Messages

```
Tt_message m, n;
m = tt_message_create();
...
tt_message_send(m);

... wait around for tt_fd to become active

n = tt_message_receive();
if (m == n) {
// this is a reply to the message we sent
    if (TT_HANDLED == tt_message_state(m)) {
        // the receiver has handled the message, so we can go on
        ....
    }
} else {
    // this is some new message coming in
}
```

▼ Can a process send a request to itself?

Yes. A process can send a request that gets handled by itself. A typical pattern for this type of request is:

```
{
    ...
    tt_message_arg_val_set(m, 1, "answer");
    tt_message_reply(m);
    tt_message_destroy(m);
    return TT_CALLBACK_PROCESSED;
}
```

However, in the case where the handler and the sender are the same process, the message has already been destroyed when the reply comes back (to the same process). Any messages (such as callbacks or user data) attached to the message by the sender are also destroyed. To avoid this situation, do *not* destroy the message; for example:

```
{
    ...
    if (0!=strcmp(tt_message_sender(m),tt_default_procid())) {
        tt_message_destroy(m);
    }
}
```


▼ **Can I pass my own data to a function registered by `tt_message_callback_add`?**

To pass your own data to a function registered by `tt_message_callback_add`, use the user data cells on the message; for example:

```
x = tt_message_create();
tt_message_callback_add(x, my_callback);
tt_message_user_set(x, 1, (void *)my_data);

....

Tt_callback_action
Tt_message_callback(Tt_message m, Tt_pattern p)
{
    struct my_data_t *my_data;
    my_data = (struct my_data_t *)tt_message_user(m, 1);

    ...
}
```

Note – User data can only be seen in the client where the data is sent.

▼ **How can I send arbitrary data in a message?**

The ToolTalk service does not provide a built-in way to send structs; it only provides a way to send strings, ints, and byte arrays. To send structs, use an XDR routine to turn the struct into a byte array and put the bytes in the message. To deserialize, use the same XDR routine.

▼ Can I transfer files with the ToolTalk service?

No, not directly. You can however:

- Place the file data in a message argument.
The ToolTalk service copies the message data from the application into the library, from the library to `ttsession`, from `ttsession` to the receiver's library, and then out of the library when the receiver gets the argument value. If the data is large, this method can be very slow and use up a large amount of memory.
- Place the file name in a message argument.
This method assumes that every receiver mounts the file, and mounts it at the same mount point.
- Place the file name in the `tt_message_file` attribute.
This method also assumes that every receiver mounts the file; however, the ToolTalk service will resolve any mount point differences.

▼ How are memory (byte) ordering issues handled by the ToolTalk service?

The ToolTalk service allows you to place ints, strings, and byte vectors into messages. An XDR routine ensures that these data types are correct for each client. If you have data that is not one of these three data types, you must serialize the data into a byte vector before you place it into a message.

▼ Can I re-use messages?

No. Messages cannot be sent multiple times with different arguments. They must be iteratively created, sent, and then destroyed.

▼ What happens when I destroy a message?

When you destroy a message, you destroy the handle but *not* the underlying message. The underlying message is destroyed only when ToolTalk is done with it and all the external handles are destroyed. For example, if you destroy a handle to a message immediately after you send it, you will get a new handle when the reply comes back.

However, once you destroy a message, the ToolTalk service will not show it to you again under any circumstances. For example, if you register a pattern to observe a request you send and then destroy the message when your pattern matches it, you will not see the message when it is in state "handled" (that is, when it is a reply).

▼ **Can I have more than one handler per message?**

No, not currently. If you want multiple processes, you can use notices; or you can use message rejection to force the ToolTalk service to deliver the request to all the possible handlers — however, each of these handlers must actually perform some kind of operation.

▼ **Can I run more than one handler of a given ptype?**

Yes, you can run more than one handler of a given ptype. However, the ToolTalk service does not have a concept of load balancing; that is, the ToolTalk service will choose *one* of the handlers and deliver additional matching messages to the chosen handler only. There are several ways to force the ToolTalk service to deliver messages to other handlers:

1. Use `tt_message_reject`.

If a message comes in and a process does not want to handle it because the process is busy, the process can reject the message. The ToolTalk service will then try the next possible handler (and apply the disposition options when it runs out of registered handlers.)

This method requires the process to be in an event loop; that is, it must call `tt_message_receive` when the `tt_fd` is active. However, if the process is in a heavy computational loop, this method fails.

2. Unregister the pattern when busy. For example:

```
m = tt_message_receive();
if (m is the message that causes us to go busy) {
    tt_pattern_unregister(p);
}
```

The ToolTalk service will not route matching messages to the process when the pattern is not registered. When you want the process to receive messages again, re-register the pattern.

Note – This method causes a race condition. For example, a second message could be sent and routed to the process in the time between the first `tt_message_receive` call and the `tt_pattern_unregister` call.

3. A combination of Methods 1 and 2.
You can use a combination of the first two techniques in the following manner:

```
get the message
unregister the pattern
loop, calling tt_message_receive until it returns 0; reject
all the returned messages
handle the message
re-register the pattern
repeat
```

Note – This method assumes that the process only registers one pattern.

▼ What value is disposition in a message?

Message disposition can override the disposition specified in the static type definition. If the message specifies the handler ptype and the message does not match any of the static signatures, the disposition set in the message will be the one followed. For example, if the disposition in the message is `TT_START` and the ptype specifies a start-string, an instance will be started.

▼ What are the message status elements?

The ToolTalk service does not use `message_status_string`. This message component is for use by the applications. The ToolTalk service only sets the message status if a problem occurs with message delivery; otherwise, this message component is set and read in an application-dependent manner.

▼ When should I use `tt_free`?

`libtt` maintains an internal storage stack from which you receive data buffers. When you call a ToolTalk API routine, any `char *` or `void *` returned points to a copy that you are responsible for freeing.

Use the mark and release functions to free allocated buffers during a sequence of operations. However, the release call frees *everything* allocated since the corresponding mark call. If you want to store certain data that was returned by the ToolTalk service, make a copy of the data before you do any operations that may free it.

▼ What does the `pctype` represent?

Ptypes are programmer-defined strings that name tool kinds. (You can roughly translate `pctype` as *process type*.) Each `pctype` can be associated with a set of patterns that describe the messages in which that particular `pctype` is interested and a string for the ToolTalk service to invoke when an instance of that `pctype` needs to be started.

The main purpose of `pctype`s is to allow tools to express interest in messages even when no instance of the tool is actually running in the scope in which the message is sent. If a tool is able to perform a message's requested operation, or wants to be notified when a particular message is sent, it indicates this instruction in its `pctype` and ToolTalk will start the tool when necessary. Since the `pctype` database can also be modified by the system administrator or user, the mechanism allows the site's or user's favorite tool be designated as the tool to handle a particular message.

▼ Why are my new types not recognized?

`ttsession` only reads the types database on start-up, or on receipt of a `USR1` signal. To recognize new types, you can either restart `ttsession`, or (preferably) force your running `ttsession` to reread the types database by sending it the `USR2` signal; for example

```
kill -USR2 <ttsession pid>
```

▼ **Can I declare a ptype that is not in the types database?**

Yes. To do this, use the `tt_session_types_load` call.

▼ **Is ptype information used if a process of that ptype already exists?**

The ToolTalk service always looks for one handler and any number of observers for every message. In this case, even though the ToolTalk service finds a handler running, it will still look through the ptypes for any observe patterns that match the message. If a ptype with an observe pattern that matches does exist and there is no process of that ptype currently running, the ToolTalk service will start a new process or queue the message (as specified in the ptype pattern or in the message).

▼ **Can the ptype definition be modified to always start an instance (whether or not one is already running)?**

No. Messages to a ptype are blocked during start-up until the ptype either replies to the message, or issues a `tt_message_accept` call. However, the implementation of the ptype can include `tt_message_reject` for any request it gets that do not have a status of `TT_WRN_START_MESSAGE`. All requests will then be delivered to (and rejected by) all running instances of the ptype before a new one gets started. This method will be slow if many of these ptypes are running at the same time, or if the message contains a large amount of data. Alternatively, you could use `tt_message_accept`, which basically unblocks messages to the ptype.

▼ **What does `tt_ptype_declare` do?**

When you declare the ptype, your static patterns exist in `ttsession` memory. When a ptype is registered by an application, the ToolTalk service also checks for otypes that mention the ptype and registers the patterns found in these otypes. To activate the static patterns, your application must call the appropriate `join` functions.

Note – Multiple declarations by an application of the same ptype are ignored.

▼ What is `_SUN_TT_TOKEN`?

When processing message that requires an application to be started, the ToolTalk service sets this environment variable in the child process. When the application starts and performs `tt_open`, this information is passed back to the ToolTalk service to inform it that the application coming up is the one started or delegated to handle the message.

▼ When are my patterns active?

A pattern must be registered with the session in which it wants to be active. Patterns can be active for more than one file (for a given `procid`); the file part of the pattern will match any of the listed files.

Note – Contexts are not scopes. A pattern that is joined to contexts but not joined to any file or session cannot match any message.

▼ Must I register patterns to get replies?

No. You do not need to register patterns to get replies. However, if you do register a pattern that matches a reply, the reply will come through your event loop twice: once because it matched a pattern, and again because it is a reply.

▼ How can I observe requests?

Observers can observe requests *if* the pattern matches *and* the message is *not* point-to-point (that is, `TT_HANDLER`). If your observer pattern is not matching any requests, you can run `ttsession` in trace mode to find out why.

▼ How do I match to attribute values in static patterns?

The ToolTalk static pattern (that is, types database) mechanism does not allow you to match patterns by attribute values. You can match by file scope or argument `vtype` but you *cannot* by match by the particular filename or by argument value.

Note – This restriction also applies for matching on contexts in static patterns.

▼ Why am I unable to wildcard a pattern for TT_HANDLER?

You cannot wildcard patterns for TT_HANDLER-addressed messages because these messages are not pattern matched.

▼ Can I set a pattern to watch for any file scoped message?

No. Not specifying a file name when you use file scoping is virtually the same as specifying that you want to match to file-scoped messages about every file in the universe.

Note – A session attribute may be set on a file-scoped pattern to emulate file-in-session scoping; however, a `tt_session_join` call will not update the session attribute of a pattern that is scoped as `TT_FILE`.

▼ Is file scope in static patterns the same as file_in_session scope?

No, these scopes have different purposes.

For example, assume all sessions currently have the same static patterns and at least one pattern *P* that will match a message *M* (which you will be sending). No session has any clients that have registered interest in the file `foo.bar`.

You are connected to session *A* and issue a file-scoped message *M* for file `foo.bar`. Since no client of any session has previously expressed any interest in this file, session *A* is the only file that will get the message. (The message will match against static pattern *P* in session *A*.) Once the ptype is started, the pattern actually becomes scoped to file (within that session) and session *A* will honor all the promises.

However, if all sessions do *not* have the same static patterns, the results are different. For example, session *B* could have an extra pattern *P* that is file-scoped and that should match message *M*. When message *M* is sent in session *A*, the observer will not send the message to session *B* if no client of session *B* has previously expressed interest in the file `foo.bar`. However, if a client of session *B* has previously expressed interest in the file `foo.bar`, then the observer would know that at least one client in that session was interested in the file `foo.bar` and would send also the message to session *B*.

▼ **What is the difference between `arg_add`, `barg_add`, and `iarg_add`?**

The `barg_add` and `iarg_add` calls are basically an `arg_add` call followed by a set of the value.

▼ **What is the type or `vtype` in a message argument?**

The type or `vtype` (which is short for *value type*) in a message argument indicates the semantic domain in which the argument's value has meaning and is determined by your application.

Vtypes are analogous to typedefs in C. Every `vtype`, by convention, corresponds to only one of the three possible data types for argument values.

The `vtype` mechanism allows you to declare two values as the same type; for example, you could declare both the `vtype` `messageID` and the `vtype` `bufferID` as C strings with different semantics for each: some operations are valid on `messageID` only, some operations are valid on `bufferID` only, and some operations are valid on both `vtypes`. The pattern-matching mechanism makes sure that a request with a `bufferID` string does not get matched to a pattern for an operation that is only valid on `messageID` strings.

▼ **How do I use contexts?**

You can use contexts to restrict matching. To restrict matching, a message must have the same contexts, or a superset of the contexts, in order to possibly match. Also, if the name of a context slot begins with a dollar sign (\$) character (for example, `$ISV`) and the message causes an application to be started, the environment variable for the started application will be set to whatever value is indicated in the context slot.

▼ How does tsession check for matches?

Table D-4 describes the various ways tsession checks for matches.

Table D-4 How tsession Checks for Matches

Mechanism	Description	Match?
TT_HANDLER	This type of addressing is “point-to-point” delivery — the message is passed directly to the receiver. You cannot monitor point-to-point messages because registered patterns are never checked.	No matching required.
TT_PROCEDURE	<ul style="list-style-type: none"> Scans list of static signatures (sig) that have same operation (op) and collects lists of observers and potential handlers. If the sig has no arguments and no contexts If sig prototype (number, type and mode of args) have different values If the sig contexts are a subset of the contexts in the message Saves information for any static observers that require queuing. Scans through dynamic patterns and adds to lists of observers and potential handlers. To form the lists, tsession first uses the patterns with operations, then the patterns without operations. Checks reliability, states, class, address, handler, handler ptype, scope, object, otype, sender, sender_ptype, args, contexts. Delivers to observers first (because a handler can change state). Delivers to handler with best match — if more than one handler equally “best” matches, the handler is arbitrarily chosen. 	<p>=> Match</p> <p>=> No Match</p> <p>=> Match</p>
TT_OBJECT & TT_OTYPE	<p>Checks whether the otype argument is filled in</p> <ul style="list-style-type: none"> If sig has a different otype If sig has no otype & scope is different <p>Otherwise, matches in the same manner as for TT_PROCEDURE matching.</p>	<p>=> No Match</p> <p>=> No Match</p>

▼ How many kinds of scope does the ToolTalk service have?

Currently, the ToolTalk service has only two kinds of scope: session scope and file scope.

Note – X session is sometimes referred to as a scope; however, the X session is really a session scope.

▼ **What are the `tt_db` directories, and what is the difference between the types database and the `tt_db` directories?**

The ToolTalk types databases store the static ptype and otype definitions. These definitions declare the messages to which applications and objects respond. The ToolTalk types compiler modifies the types database when you add or change static type definitions. Upon starting, `ttsession` reads in these type files.

The `TT_DB` database is created by `rpc.ttdbserverd`. The `tt_db` directories contain the associations between files in this partition and the sessions with patterns interested in these files. It also contains all the object spec information for files in this partition.

▼ **What should the `tt_db` databases contain?**

The `tt_db` databases currently contain the following ten files:

```
access_table.ind
access_table.rec
file_object_map.ind
file_object_map.rec
file_table.ind
file_table.rec
file_table.var
property_table.ind
property_table.rec,
property_table.var
```

The permissions for these files are set to `-rw-r--r--`.

▼ **What does `rpc.ttdbserverd` do?**

The ToolTalk database server daemon performs three major functional duties:

1. It stores the ToolTalk session IDs of sessions with clients that have joined a file using the `tt_file_join` call.
2. It stores file-scoped messages that are queued because the message disposition is `TT_QUEUED` and a handler that can handle the message has not yet been started.
3. It stores ToolTalk objects specs.

▼ **Do `ttsession` and `rpc.ttdbserverd` ever communicate?**

No.

▼ **What message bandwidth can be supported?**

About 100 small messages per second. Performance mainly depends on how many recipients each message has; that is, notices that do not match any pattern are the cheapest while messages that match many observers are the most expensive.

▼ **Is there a limit to the message size or the number of arguments?**

No. However, while there is no designed limitation to the size of a ToolTalk message or the number of arguments ToolTalk does copy the data several times (both from one area in the client's address space to another area, and across the RPC connection to and from the server). For example, a megabyte of data in a ToolTalk message would be copied at least 4 times:

- From your storage to the ToolTalk library's storage
- From the ToolTalk library to the ToolTalk server
- From the ToolTalk server to the receiver's library
- From the receiver's library to the final resting place.

If there are processes observing the message, even more copying will take place. In addition, no other messages for this session can be delivered during the copy time because the `ttsession` process is single-threaded. Therefore, if you plan to send really big chunks of data very often, you probably want to consider using a non-ToolTalk way to pass the data.

▼ **What is the most time efficient method to send a message?**

Directly to process (that is, addressing the message using `TT_HANDLER`) is faster than procedural messages that match only one receiver.

▼ **What network overhead is involved?**

The ToolTalk service does *not* use hardware broadcast or multicast. The message is sent directly to the `ttsession` process for the session (whether across the network or not). When a pattern is registered, it also is sent directly to the `ttsession` process. The `ttsession` process matches the message against all the patterns and sends the message directly to only the processes that registered patterns that match the message — if no process on another machine is interested in a message, that machine does not need to wake up and look at it.

▼ **Does the ToolTalk service use load balancing to handle requests?**

No, the ToolTalk service is not a load-distribution mechanism. If two processes with identical patterns are registered, the ToolTalk service arbitrarily chooses one of process and delivers all matched messages to it. You can do load distribution if you unregister the pattern while the process is busy and reject any messages that may have been received before the pattern was unregistered.

▼ **What resources are required by a ToolTalk application?**

Coarse numbers indicate that several 100K of working set for a sending client, `ttsession`, and a receiving client is required to process messages. ToolTalk memory requirements do not grow over time.

▼ **What happens if the `ttsession` exits unexpectedly?**

When `ttsession` crashes, the `tt_fd` becomes active and most ToolTalk API calls will return the `TT_ERR_NOMP` error message

```
No Message Passer
```

Most applications assume this message means that something has happened to `ttsession` and will stop sending or receiving ToolTalk messages. Possible recovery from this situation may include:

- Recognize the `TT_ERR_NOMP` situation
- Call `tt_close` to clean up the connection from its end

- Reinitialize the ToolTalk service
- Call the sequence:

```
tt_open, tt_default_session_join, tt_fd
```

- Re-register all patterns and re-declare ptypes

Note – You may need to manipulate the setting of the environment variable `_SUN_TT_SESSION` and the value of the `_SUN_TT_SESSION` property of the root X window (if it exists) when you restart a crashed `ttsession` to take over where the last one left off. Also, you must inform other participants of the crashed session of the restarted session and the new session id so that they can recover.

When `ttsession` crashes, you will not be able to recover the following:

- Patterns registered by procsids in the crashed session
- Outstanding requests from procsids in the crashed session
- Messages that were passed the `tt_message_send_on_exit` call by procsids in the crashed session
- Session props
- Session-queued messages

▼ What happens if `rpc.ttdbserverd` exits unexpectedly?

If `rpc.ttdbserverd` exits unexpectedly, `inetd` will start a new one to replace it. Data may be temporarily unavailable but no data will be lost. However, one or more API calls may return `TT_ERR_DBAVAIL`; if the call returns `TT_OK`, the `dbserver` will update the ToolTalk databases appropriately either immediately or when a new `dbserver` reads the crash recovery log.

▼ What happens if a host or a link is down?

When TCP notices that a host or a link is down, the TCP connection breaks. When a process connection to `ttsession` breaks, `ttsession` behaves as if the process exited. All the patterns are cleaned up, and the process will receive the error message `TT_ERR_NOMP` if it attempts to send or receive messages.

▼ What does `tt_close` do?

When you call `tt_close`, `ttsession` only closes the current `procid`. If the current `procid` is the last `procid` to close, it cleans up all the ToolTalk structures created since the `tt_open` call was made. You must call `tt_close` on the file descriptor returned by `tt_fd`; otherwise, your file descriptor count will rise upon successive `tt_open` and `close` calls.

▼ Is message delivery guaranteed on a network?

Yes, delivery is reliable because messages are sent using RPC on TCP/IP.

▼ Is there a temporal sequence of message delivery?

Between a given sender and receiver, message sequence is preserved; that is, if process *A* first sends message *M1* and then later sends message *M2* and both messages are received by process *B*, process *B* will receive message *M1* before it receives message *M2*. However, there are two special exceptions:

1. If process *B* receives message *M1* and then rejects it, message *M1* is redispached to process *C*. In the meantime, (while process *B* is deciding whether to reply or reject message *M1*), the ToolTalk service continues its message delivery. These subsequent messages can appear to "pass" the first request.
2. If process *B*'s messages are queued, it will receive its queued messages when it declares a `p_type` that contains the pattern which caused the queuing. However, process *B* may not actually receive its queued messages (in this case, message *M1*) until it has already received subsequent messages from process *A*.

▼ What is `unix`, `xauth`, and `des`?

These are the three kinds of authentication:

- `unix` tells you the `uid` of the entity that is making an `rpc` call on you. The `observer` enforces security on each `RPC` call and uses this kind of authentication by default.
- `xauth` uses a read-protected file in your home directory to control access to your X display (and, thus, to your `ttsession`).
- `des` uses the Data Encryption Standard (DES) to ensure that processes who talk to `ttsession` are really who they say they are.

▼ **Can my applications hide messages from each other?**

No. The ToolTalk service intentionally does not provide a mechanism that allows one application lock out other applications from seeing its messages.

▼ **Is there protection against interception or imitation?**

No. The “plug-and-play” concept of the ToolTalk service allows applications to install and deinstall tools of choice that best perform a particular task. If application *B* responds better to protocol *X* than does application *A*, protocol *X* should be allowed to deinstall application *A* and install application *B*.

▼ **Where are queued messages stored and how secure is the storage?**

File-scoped queued messages are stored in a database on the same filesystem as the file to which they are scoped. The database is readable to the super-user only, and the ToolTalk database server (running as root) only gives the messages to processes owned by a user with read access on the file.

Session-scoped queued messages are stored in the address space of the `ttsession` that manages the session. `ttsession` only gives the messages to a process that has satisfied the authentication mode in which the `ttsession` is running.

▼ **Is the ToolTalk service C2 qualified?**

No.

▼ **How can I trace my message’s progress?**

To trace your message’s progress, turn on the trace output of the `ttsession` involved. Enter the following command:

```
kill -USR1 <unix_pid_of_the_ttsession_process>
```


▼ **How can I isolate my debugging tool from all the other tools using the ToolTalk service?**

To isolate your debugging tool, use the "process tree session" mode. This mode places the session name in an environment variable to find the ttsession process. To use this mode, do the following:

1. Start a new process tree session with trace mode turned on.

```
% ttsession -t -c $SHELL
*
* ttsession (version 1.0, library 1.0)
*
ttsession: starting
%
```

ttsession starts, sets the environment variable, and forks the given command (\$SHELL). You are now running in a subshell. All the commands run from this subshell will use the ttsession started from the command line. You can check the value of the `_SUN_TT_SESSION` environment variable for the session id of this new ttsession.

2. Inside the subshell, run the test programs:

```
% ./my_receiver &
[1] 4532
% ./my_sender &

.. and look at the output of the ttsession trace.
```

3. Exit the subshell after testing.

If you start any tool that uses the ToolTalk service in the subshell, it uses the process tree ttsession, not the X-session ttsession, which will produce undefined results.

▼ **Can I use the ToolTalk service with C++?**

Yes. The ToolTalk API header file is set up to deal with C++. When you use C++, `tt_c.h` declares all the API calls as extern C.

▼ **Should I qualify my filenames?**

No. The ToolTalk service does not allow explicit hostname qualification of pathnames. If you use a filename that contains a colon (:) symbol, the ToolTalk service searches for a filename that contains the colon symbol. The `tt_message_file` and `tt_default_file` calls return the *realpath* of the specified file as it appears on the machine on which you invoked the call. The ToolTalk service ensures that

- a. If two clients file-scope to the same file on different machines, they can talk to each other without regard to how the two files are actually mounted on each machine.
- b. A locally-valid, canonical pathname is returned back to you.

▼ **Can you tell me about ToolTalk objects?**

ToolTalk objects are somewhat different from what you normally encounter in typical object-oriented languages.

Otypes and inheritance are for implementation only. Two specs can be of the same otype but have different properties — they only share the operations as defined by the signatures in the otype declaration. For each signature in the otype declaration, a ptype must be designated. The designated ptype (process-type) is the 'execution engine' for this operation on an object of this otype. The file part of a spec is similar to a required property: every spec must have a file name; however, that file does not need to exist. The filename part of the spec performs several functions, including:

1. Allows you to specify the host and partition on which the spec will be stored.
2. Provides a grouping mechanism for objects.
3. Allows the ToolTalk-enhanced standard operating commands (such as the `ttmv` command) to keep the database's view of the world consistent with the real world.

▼ Is there a ToolTalk news group?

Yes. The ToolTalk news group is *alt.soft-sys.tooltalk*.

Glossary

CAD

Computer-aided design.

CASE

Computer-aided software engineering.

Category

Attributes of a pattern that indicate whether the application wants to handle requests that match the pattern or only observe the requests.

Classing Engine (CE)

Identifies the characteristics of desktop objects; that is, it stores attributes such as print method, icons, and file opening commands of desktop objects.

Classing Engine tables

The types database read by the OpenWindows Classing Engine.

contexts

Associates arbitrary pairs (that is, <name. value> pairs) with ToolTalk messages and patterns.

dynamic message patterns

Provides message pattern information while your application is running.

fail a request

Inform a sending application that the requested operation cannot be performed.

fd	File descriptor.
file	A container for data that is of interest to applications.
libtt	The ToolTalk application programming interface (API) library.
handle a message	To perform the operation requested by the sending application; to send a ToolTalk reply to a request.
initial session	The ToolTalk session in which the application was started.
mark	An integer that represents a location on the API stack.
message	A structure that the ToolTalk service delivers to processes. A ToolTalk message consists of an operation name, a vector of type arguments, a status value or string pair, and ancillary addressing information.
message callback	A client function. The ToolTalk service invokes this function to report information about the specified message back to the sending application; for example, the message failed or the message caused a tool to start.
message pattern	Defines the information your application wants to receive.
message protocol	A message protocol is a set of ToolTalk messages that describe operations the applications agree to perform.
notice	A notice is informational, a way for an application to announce an event.
object content	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.

object files

Files that contain object information. Applications can query for objects in a file and perform operations on batches of objects.

object-oriented messages

Messages addressed to objects managed by applications.

object specification (spec)

An object specification (known as 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.

object type (otype)

The object type (otype) for your application provides addressing information that the ToolTalk service uses when delivering object-oriented messages.

object type identifier (otid)

Identifies the object type.

observe a message

To only view a message without performing any operation that may be requested.

observe promise

Guarantees that the ToolTalk service will deliver a copy of each matching message to ptypes with an observer signature of start or queue disposition. The ToolTalk service will deliver the message either to a running instance of the ptype, by starting an instance, or by queueing the message for the ptype.

opaque pointer

A value that has meaning only when passed through a particular interface.

package

A group of components that together create some software. A package contains the executables that comprise the software, but also includes information files and scripts. Software is installed in the form of packages.

pattern callback

A client function. The ToolTalk service invokes this function when a message is received that matches the specified pattern.

process

One execution of an application, tool, or program that uses the ToolTalk service.

process-oriented messages

Messages addressed to processes.

procid

The process identifier.

ptid

The process type identifier.

ptype

The process type.

reject a request

Tells the ToolTalk service that the receiving application is unable to perform the requested operation and that the message should be given to another tool.

request

A request is a call for an action. The results of the action are recorded in the message, and the message is returned to the sender as a reply.

rpc.ttdbserverd

The ToolTalk database server process.

scope

The attribute of a message or pattern that determines how widely the ToolTalk service looks for matching messages or patterns.

sessid

Identifies the session.

session

A group of processes that are related either by the same desktop or the same process tree.

signatures

A pattern in a ptype or otype. A signature can contain values for disposition and operation numbers.

- Ptype signatures (*psignatures*) describe the procedural messages that the program wants to receive.
- Otype signatures (*osignatures*) define the messages that can be addressed to objects of the type.

spec	See object specification.
static message patterns	Provides an easy way to specify the message pattern information if you want to receive a defined set of messages.
tool manager	A program used to coordinate the development tools in the environment.
ToolTalk Types Database	The database that stores ToolTalk type information.
ttddbck	Check and repair utility for the ToolTalk database.
ttsession	The ToolTalk communication process.
tt_type_comp	The ToolTalk type compiler.
wrapped shell commands	ToolTalk-enhanced shell commands. These commands safely perform common file operations on ToolTalk files.
xdr format tables	The types database read when <code>ttsession</code> is invoked.

Index

Symbols

\$OPENWINHOME/li, 201
\$OPENWINHOME/lib/openwin-
sys, 199
() function, 64
, tt, 198
/etc/inet/inetd.conf, 199
/usr/openwin/bin, 198
tt_message_user, 64
_SUN_TT_ARG_TRACE_WIDTH, 32
_SUN_TT_FILE, 32
_SUN_TT_HOSTNAME_MAP, 32, 38
_SUN_TT_PARTITION_MAP, 32, 39
_SUN_TT_SESSION, 16, 32, 65, 200, 201,
202
_SUN_TT_TOKEN, 32, 201
_sun_tt_token, 217

A

accessing ToolTalk data from machines
not running a ToolTalk database
server, 37
accessing ToolTalk databases, 57
adding a message pattern callback, 95
adding callbacks to static patterns, 89

adding files to scoped patterns, 26
adding inter-operability functionality, 160
adding ToolTalk code, 166
Xedit files, 166
Xfontsel files, 174
adding values to spec properties, 132
address attribute, 83
address attributes, 70
addressing
otype, 80
addressing messages, methods of, 10
algorithm
object-oriented message delivery, 77
process-oriented message
delivery, 74
allocating storage space, 145
allocation stack, 143
alt.soft-sys.tooltalk, 229
API header file, including in program, 59
application programming interface
(API), 12
application programming interface (API)
libraries, 30
application types, installing, 41
architecture, 13
arg_add call, 219
args attribute, 84

assigning otype, for specs, 131
attributes
 address, 70, 83
 arg, 84
 class, 83
 op, 84
 scope, 70, 84
 setting, 83
attributes, of message patterns, 20

B

background jobs, 17
barg_add call, 219
batch sessions, 17
broken references, 137

C

C2 qualification, 226
callback routines, 146
 invoking, 123
callback routines, adding to message patterns, 95
callbacks, attached to static patterns, 125
callbacks, for handlers, 124
calls provided to manage information storage, 144
CASE Interoperability Message Sets, 6
CEPATH, 32
changing ToolTalk-enhanced shell commands, 56
checking ToolTalk databases, 18
checking ToolTalk error status, 150
class attribute, 83
close function, 202
co-edit, 18
communication process, 13
comparing objids, 134
components of the ToolTalk service, 13
computational loops, 213
connecting processes to the same session, 202

context arguments, 146
context slots, used to create environment variables, 35
contextdcl, 105, 110
contexts, defined, 27
contexts, to restrict matching, 219
convert ToolTalk type data, 30
converting ToolTalk types, script for, 155
cp command, 18
cpp command, 41
creating a ptype file, 103
creating dynamic message patterns, 95
creating general messages, 83
creating messages, 80
creating object-oriented messages, 87
creating otype files, 109
creating process-oriented messages, 86
creating specs, 131

D

database
 check and recovery tool, 30
 records, 13
database server
 installing ToolTalk, 35
 process, 13
 redirecting, 37
 redirecting file system partitions, 38
 redirecting host machines, 37
database server redirection files, 200
database utility ttdbck, 57
databases
 accessing ToolTalk, 57
 check and repair utility, ttdbck, 18
 displaying, checking, and repairing of ToolTalk, 57
 maintaining ToolTalk, 18
debugging, with ttsnoop, 45
default session
 joining, 97
 quitting, 98
delete message, 90

- deleting message patterns
 - message patterns
 - deleting, 96
- deleting messages, 127
- demonstration, 159
- demonstration programs
 - co-edit, 18
 - edit_demo, 18, 139
 - ttsample1, 18
 - ttsnoop, 18
- des, 225
- deserializing structured data, 74
- Desktop Services
 - Started, 206
- Desktop Services Message Set, 2
- destroying message patterns
 - automatically, 96
- destroying messages, 127
- destroying specs, 137
- determining spec properties, 132
- determining who receive messages, 10
- directories, list and location of, 29
- DISPLAY, 32, 201
- disposition attributes, 28
- Document and Media Exchange Message Set, 4
- dynamic message patterns, 93
 - creating, 95
- dynamic method, 19

E

- edit_demo, 18, 139
- environment variables, 31
 - _SUN_TT_ARG_TRACE_WIDTH, 32
 - _SUN_TT_FILE, 32
 - _SUN_TT_HOSTNAME_MAP, 32, 38
 - _SUN_TT_PARTITION_MAP, 32
 - _SUN_TT_PARTITION_MAP, 39
 - _SUN_TT_SESSION, 16, 32, 65
 - _SUN_TT_TOKEN, 32
 - CEPATH, 32
 - created from message contexts, 35
 - DISPLAY, 32
 - SUN_TTSESSION_CMD, 32
 - TMPDIR, 33
 - TTPATH, 32
- error handling functions, 150
- error macros, 150
- error propagation, 154
- error status, 149
 - checking, 150
 - retrieving, 150
- error value, 150
- errors, process type, 44
- event loop, 213
- examining messages, 120
- examining spec information, 134
- examining type information, 43

F

- failed connection, causes of, 36
- failing requests, 127
- features, of the ToolTalk service, 146
- features, of ToolTalk, 8
- file, 204
 - ToolTalk concept of, 11
- file information
 - managing, 138
- file query functions, 147
- file scope, 71
- file scoping, restrictions, 11
- file scoping, restrictions to, 22
- file-in-session scope, 73
- files
 - hostname_map, 37
 - list and location of, 29
 - maintaining ToolTalk, 18
 - managing with object data, 137
 - object type, 107
 - partition_map, 38
 - XDR format, 14
- files of interest
 - joining, 99

- quitting, 99
- file-scoped message flow, 207
- file-scoped messages, queued, 226
- filter routines, 147
- free storage space, 90
- freeing allocated storage space, 146
- functions
 - tt_message_user, 64
- functions with natural return values, 150
- functions without natural return values, 150

H

- handling replies easily, 120
- handling requests, 125
- header file, 30
- help on how to invoke tsession, 15
- hostname qualification of pathnames, explicit, 228
- hostname_map file, 37
- how applications use ToolTalk messages, 8
- how the ToolTalk technology works, 163

I

- iarg_add call, 219
- identifying data in existing files, 130
- identifying messages easily, 119
- information provided by the ToolTalk service, 143
- information provided to the ToolTalk service, 143
- informing sender of failed request, 127
- initial session, 60
- initializing your process, 61
- installing application types, 41
- installing the ToolTalk database server from a remote machine, 36
- installing the ToolTalk database server, 35
- installing type information, 112

- invoking callback routines, 123

J

- joining default sessions, 97
- joining files of interest, 99
- joining multiple sessions, 98

K

- kill command, 44

L

- LD_LIBRARY_PATH, 201
- li, 198
- libtt, 14, 202, 215
- libtt.a, 199
- libtt.s, 199
- libtt.so, 201
- load balancing, 213

M

- maintaining specs, 133
- maintaining ToolTalk files and databases, 18
- managing files that contain object data, 137
- managing object and file information, 138
- manpages, location of ToolTalk, 31
- manually starting a session, 15
- marking information for storage, 144
- marking the ToolTalk API stack, 120
- merging compiled ToolTalk type files into running tsession, 112
- merging type information, 112
- message
 - delete, 90
- message attributes, 69
- message attributes, comparing to pattern attributes, 21
- message callback, 146

- message callbacks, 124
- message callbacks, adding, 87
- message delivery
 - object-oriented algorithm, 77
 - process-oriented algorithm, 74
- message patterns
 - unregistering, 96
- message pattern attributes, 20
- message patterns, 9, 19
 - adding callbacks to, 95
 - automatically unregistering and destroying, 96
 - minimum specifications, 21
 - static, 101
 - updating, 97
- message protocol, 12
- message_status_string, 214
- messages
 - completing, 80
 - creating, 80
 - creating general-purpose, 83
 - deleting, 127
 - determining recipients of, 9
 - examining, 120
 - handling, 9
 - identifying and processing easily, 119
 - methods of addressing, 10
 - object-oriented, 10
 - observing, 9
 - process-oriented, 10
 - receiving, 9
 - sending, 8, 90
- messages, retrieving, 117
- MIT X, 201
- modifying applications, 160
 - xedit, 160
 - xfontsel, 161
- modifying applications to send messages, 80
- modifying makefile, 161
- modifying your application to use the ToolTalk service, 12
- moving objects between file systems, 136

- moving objects between files, 136
- multiple processes, 213
- multiple sessions
 - storing session ids of sessions, 98
- multiple sessions, joining, 98
- mv command, 18, 56

N

- network types database, converting, 157
- networked environments, 65
- news group, ToolTalk, 229
- notice, 67
- notifying processes if tool exits unexpectedly, 153

O

- object content, 130
- object data, 129
- object information
 - managing, 138
- object specification (spec), 130
- object type (otype), 107
- object-oriented message delivery, 77
- object-oriented messages, 10, 129
 - creating, 87
- objects
 - moving between file systems, 136
 - moving between files, 136
 - ToolTalk, 130
- objects, ToolTalk, 228
- objid
 - comparing, 134
 - obtaining, 131
 - obtaining new, 133
 - retrieving new, 133
- obtaining new objid, 133
- obtaining objid, 131
- op attribute, 84
- otype
 - assigning for specs, 131
- otype addressing, 80

- otype file, 107
- otype files
 - creating, 109
 - header information, 109
 - signature information, 110
- otype files, installing, 41
- otype signature, 108
- otypes, examining information, 43

P

- partition_map file, 38
- pattern attributes, comparing to message attributes, 21
- pattern callback, 146
- pattern callbacks, 124
- pointers, to API objects, 146
- point-to-point (PTP) message passing feature, 83
- point-to-point messages, 124
- process
 - communication, 13
 - database server, 13
- process identifier (procid), 61
- process type (ptype), 101
- process type errors, 44
- process type, declaring, 113
- processing messages easily, 119
- process-oriented message delivery, 74
- process-oriented messages, 10
 - creating, 86
- procid, 61, 204
 - closing default, 65
 - setting default, 61
- Project DOE, 198
- ps command, 44
- ptype, 204
- ptype file, creating, 161
- ptype files
 - creating, 103
 - property information, 104
 - registering, 102

- registering with ToolTalk, 113
- signature information, 105
- unregistering with ToolTalk, 115

- ptype files, installing, 41
- ptype signature, 102
- ptype, installing, 162
- ptypes, check for existing, 112
- ptypes, examining information, 43
- ptypes, multiple, 115
- ptypes, undeclaring, 115

Q

- quitting default session, 98
- quitting files of interest, 99

R

- read in the types from database, 16
- reading
 - hostname_map files, 38
 - partition_map files, 39
- reading ToolTalk data from read-only file system partitions, 37
- read-only file systems, 130
- read-only files, creating objects of pieces of, 130
- realpath, 228
- receiving ToolTalk messages, 9
- recipients, 8
- recognizing replies easily, 120
- records database, 13
- redirecting file system partitions, 38
- redirecting host machines, 37
- redirecting the ToolTalk database server, 37
- register file scope patterns, 204
- registering
 - in a specified session, 62
 - in the initial session, 61
 - with the ToolTalk service, 60
- registering in multiple sessions, 63
- registering ptypes, 102

rejecting requests, 127
 removing type information, 43
 repairing ToolTalk databases, 18
 replies

- recognizing and handling easily, 120

 replying to requests, 125
 request, 67
 requests

- failing, 127
- handling, 125
- informing sender of failed, 127
- rejecting, 127
- replying to, 125

 reread types file, 16
 rereading type information, 44
 retrieving new obji, 90
 retrieving new objid, 133
 retrieving ToolTalk error status, 150
 return value

- natural, 150
- no natural, 150

 returned integer, status, 152
 returned pointer, status, 151
 returned value, status, 150
 reverting to previous versions of the
 ToolTalk database, 37
 rm command, 18, 57
 routines

- callback, 146
- filter, 147

 rpc.ttdbserverd, 13, 130, 198
 running the new ToolTalk database
 server, 37
 runtime stack, 143

S

same process, sending and receiving
 messages in, 64
 scenarios illustrating the ToolTalk service
 in use, 2
 scope attribute, 84

scope attributes, 70

- file, 71
- file-in-session, 73
- session, 73

 scope, to union of TT_FILE_IN_SESSION
 and TT_SESSION, 71
 scopes, that use files, 71
 scopes, types of, 22
 scoping messages to every client with
 registered interest, 73
 scoping to file in session, 24
 scoping to file only, 23
 scoping to files and sessions, 25
 scoping to session only, 22
 senders, 8
 sending messages, 90

- modifying applications, 80

 sending notices, 67
 sending requests, 68
 sending ToolTalk messages, 8
 serializing structured data, 74
 server authentication level, 15
 session identifier (sessid), 11
 session identifiers, multiple for one
 session, 15
 session scope, 73
 session, ToolTalk concept of, 10
 sessions bound to a character terminal, 17
 session-scoped message flow, 206
 session-scoped messages, queued, 226
 setting attributes, 83
 setting up to receive messages, 63
 share/include/desktop, 198
 shell commands

- standard
 - cp, 18
 - mv, 18, 56
 - rm, 18, 57
- ToolTalk-enhanced, 18, 29, 56, 138
 - changing, 56
 - ttmv, 56

 shell commands, ToolTalk-enhanced

- ttcp, 56, 138
- ttmv, 56, 138
- ttrm, 56, 138
- ttrmdir, 56, 138
- tttar, 56, 138
- shell scripts
 - ttrsh, 33
- signals, to which ttsession responds, 16
- signatures
 - otype, 108
 - pctype, 102
- SIGUSR1 signal, 16
- SIGUSR2 signal, 16
- silent operation, 15
- spec *See object specification*
- spec, destroying an object, 137
- specs
 - adding values to properties, 132
 - assigning otype, 131
 - creating, 131
 - destroying, 137
 - determining properties, 132
 - examining information, 134
 - maintaining, 133
 - moving objects, 136
 - querying for objects, 134
 - storing properties, 132
 - updating, 133
 - updating existing properties, 132
 - writing into ToolTalk database, 132
- start a process tree session, 16
- start process tree sessions, 16
- starting a session manually, 15
- starting a ToolTalk session, 15
- starting programs on remote hosts, 33
- state change messages, 69
- static message patterns, 101
- static method, 19
- static patterns
 - adding callbacks, 89
 - attaching callbacks, 125
- storing
 - hostname_map files, 38
 - partition_map files, 38
 - storing spec properties, 132
- _SUN_TT_SESSION, 16
- SUN_TT_SESSION, 203
- SUN_TTSESSION_CMD, 32
- switching between multiple sessions, 205
- system types database, converting, 156

T

- t option, 206
- t option, of ttsnoop, 45
- TMPDIR environment variable, 33
- ToolTalk database server
 - reverting to previous versions, 37
 - running new, 37
- ToolTalk message sets
 - Desktop, 2
 - Document and Media Exchange, 4
- ToolTalk messages, 8
- ToolTalk object, 130
- ToolTalk service, 1
- ToolTalk type compiler tt_type_comp, 107
- ToolTalk types databases, moving, 158
- ToolTalk-enhanced shell commands, 138
- trace mode, 16, 206, 217, 226
- trace mode, toggling, 16
- TT_BOTH, 25
- tt_c.h, 199
- TT_CALLBACK_CONTINUE, 208
- TT_CALLBACK_PROCESSED, 208
- tt_close, 65, 96, 202
- tt_default_file, 228
- tt_default_session_set, 62, 204
- tt_fd, 61, 62
- TT_FILE, 23
- TT_FILE_IN_SESSION, 24
- tt_file_join, 99
- tt_file_objects_query, 134, 147
- tt_file_quit, 99
- TT_HANDLED, 118

tt_int_error, 152
tt_is_err, 151, 152
tt_message__set, 83
tt_message_accept, 120, 216
tt_message_callback_add, 87, 120, 211
tt_message_create, 83
tt_message_destroy, 87, 90, 127
tt_message_fail, 127
tt_message_file, 72, 228
tt_message_file attribute, 212
tt_message_file_set, 73
tt_message_object, 90, 133
tt_message_receive, 117, 124, 208, 213, 214
tt_message_reject, 127, 213, 216
tt_message_send, 137
tt_message_send_on_exit, 153
tt_message_status_set, 127
tt_message_status_string_set, 127
tt_message_user call, 209
tt_message_user_set, 120
tt_objid_equal, 134
tt_onotice_create, 87
tt_open, 61, 62, 199, 217
tt_orequest_create, 87
tt_pattern_add, 95
tt_pattern_callback_add, 95, 119
tt_pattern_create, 95
tt_pattern_destroy, 96
tt_pattern_register, 96
tt_pattern_set, 95
tt_pattern_unregister, 96, 113, 214
tt_pnotice_create, 86
tt_pointer_error, 151
tt_prequest_create, 86
tt_ptype_declare, 113
tt_ptype_undeclare, 113, 115
TT_SESSION, 22
tt_session_join, 97
tt_session_quit, 98
tt_spec_bprop, 134
tt_spec_create, 131
tt_spec_destroy, 137
tt_spec_file, 134
tt_spec_move, 136
tt_spec_prop, 134
tt_spec_prop_add, 132
tt_spec_prop_set, 132
tt_spec_type, 134
tt_spec_type_set, 131
tt_spec_write, 132
Tt_status, 65
tt_status_message, 151
tt_type_comp, 41, 102, 107, 199
TT_WRN_STALE_OBJID, 90
TT_WRN_START_MESSAGE, 120, 216
ttce2xdr, 199
ttce2xdr script, 155
ttcp, 56, 138, 198
ttdbck, 57, 199
ttdbck utility, 18
ttm, 198
ttmv, 56, 138
ttmv command, 56, 228
TTPATH, 32, 200
ttrm, 56, 138
ttrmdir, 56, 138, 198
ttrsh shell script, 33
ttsample1, 18
ttsession, 13, 29, 198
ttsession parameters, 15
ttsnoop, 18
ttsnoop utility, 206
ttsnoop, debugging with, 45
ttta, 198
tttar, 56, 138
type compiler, 30
type compiler tt_type_comp, 102
type information
 examining, 43
 examining all types, 43

- installing, 112
- merging, 112
- reading, 44
- removing, 43

types file, rereading, 16

types of scopes, 22

U

- unix, 225
- unregistering a message pattern, 96
- unregistering message patterns
 - automatically, 96
- update existing spec properties, 132
- updating existing specs, 133
- updating message patterns, 97
- updating the ToolTalk service, 44
- user data cells, 211
- user type database, converting, 155
- USR1 signal, 206

V

- v option, 31
- version number, 16
- version string, 31

W

- wildcarding patterns, 218
- writing specs, into ToolTalk database, 132
- writing ToolTalk data to read-only file
 - system partitions, 37

X

- X Window System, establishing a session
 - under, 17
- xauth, 225
- XDR format file, 14
- Xedit, 159
- Xfontsel, 159
- XtAppAddInput call, 208