Sending Messages 8

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.

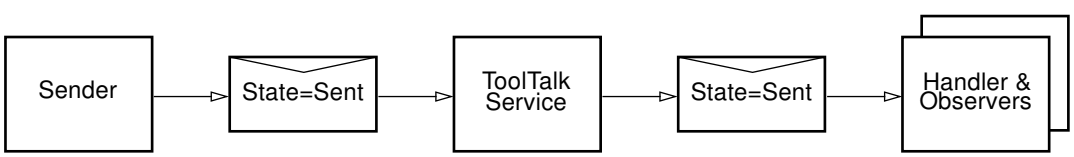

Figure 8-1 Notice Routing

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.

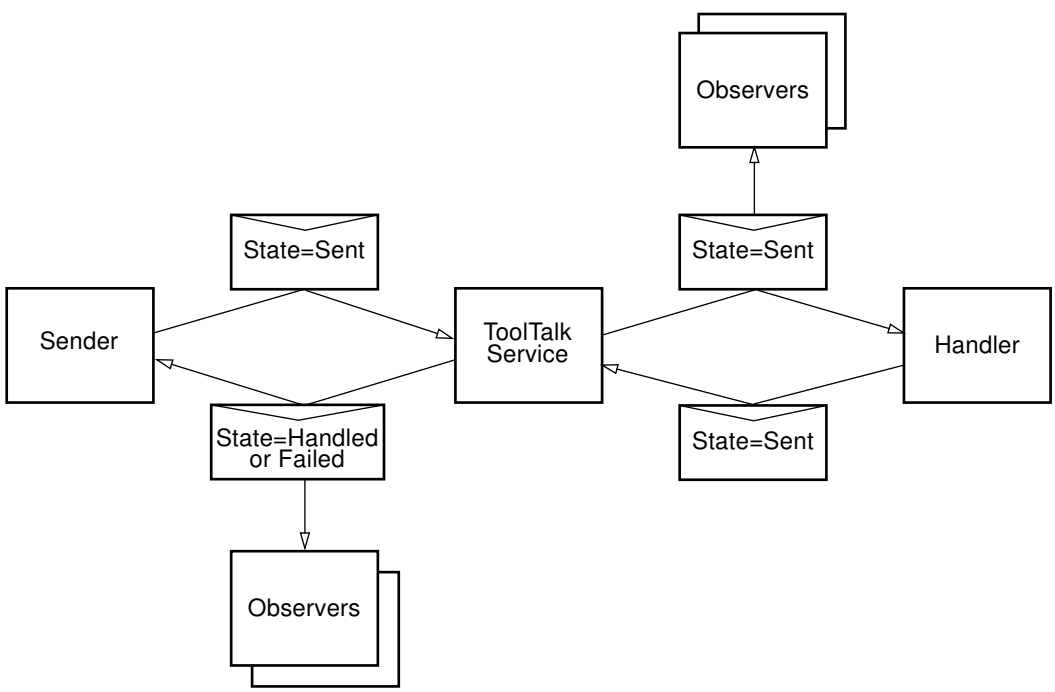

Figure 8-2 Request Routing

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 fromTable 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 `tt_session_join` call will *not* update the session attribute of a pattern that is scoped to `TT_FILE`.
TT_BOTH	Scopes to the *union* 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 *and* the session.
TT_FILE_IN_SESSION	Scopes to the *intersection* of interest in the file and the session. A pattern with only this scope will *only* 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 `tt_message_file` 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                          Tt_message m= tt_message_create();
Add scope to message                    tt_message_scope_set(m, TT_SESSION);
Add file attribute that does not        tt_message_file_set(m, file);
affect message scope
```

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                          Tt_message m= tt_message_create();
Add scope to message                    tt_message_scope_set(m, TT_FILE_IN_SESSION);
Add file to message scope               tt_message_file_set(m, file);
```

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 FinnogaCalc$_2$) 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 FinnogaCalc$_2$, FinnogaCalc$_2$ creates an object spec for cell C14 by calling a ToolTalk function. This object is identified by an objid.
6. FinnogaCalc$_2$ 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. FinnogaCalc$_2$ matches, but FinnogaCalc does not.
   d. Delivers the message to FinnogaCalc$_2$.
9. FinnogaCalc$_2$ recognizes that the message contains an object that corresponds to cell C14. FinnogaCalc$_2$ 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
tt_onotice_create(const char *objid, const char *op)	Creates an object-oriented notice.
tt_orequest_create(const char *objid, const char *op)	Creates an object-oriented request.
tt_pnotice_create(Tt_scope scope, const char *op)	Creates a process-oriented notice.
tt_prequest_create(Tt_scope scope, const char *op)	Creates a process-oriented request.
tt_message_create(void)	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
tt_message_address_set(Tt_message m, Tt_address p)	Sets addressing mode (for example, point-to-point).
tt_message_arg_add(Tt_message m, Tt_mode n, const char *vtype, const char *value)	Adds a null-terminated string argument.
tt_message_arg_bval_set(Tt_message m, int n, const unsigned char *value, int len)	Sets an argument's value to the specified byte array.
tt_message_arg_ival_set(Tt_message m, int n, int value)	Sets an argument's value to the specified integer.
tt_message_arg_val_set(Tt_message m, int n, const char *value)	Sets an argument's value to the specified null-terminated string.
tt_message_barg_add(Tt_message m, Tt_mode n, const char *vtype, const unsigned char *value, int len)	Adds a byte array argument.
tt_message_iarg_add(Tt_message m, Tt_mode n, const char *vtype, int value)	Adds an integer argument.
tt_message_context_bval(Tt_message m, const char *slotname, unsigned char **value, int *len);	Gets a context's value to the specified byte array.
tt_message_context_ival(Tt_message m, const char *slotname, int *value);	Gets a context's value to the specified integer.
tt_message_context_val(Tt_message m, const char *slotname);	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
tt_message_icontext_set(Tt_message m, const char *slotname, int value);	Sets a context to the specified integer.
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_context_set(Tt_message m, const char *slotname, const char *value);	Sets a context to the specified null-terminated string.
tt_message_class_set(Tt_message m, Tt_class c)	Sets the type of message (either notice or request)
tt_message_file_set(Tt_message m, const char *file)	Sets the file to which the message is scoped.
tt_message_handler_ptype_set(Tt_message m, const char *ptid)	Sets the ptype that is to receive the message.
tt_message_handler_set(Tt_message m, const char *procid)	Sets the procid that is to receive the message.
tt_message_object_set(Tt_message m, const char *objid)	Sets the object that is to receive the message.
tt_message_op_set(Tt_message m, const char *opname)	Sets the operation that is to receive the message.
tt_message_otype_set(Tt_message m, const char *otype)	Sets the object type that is to receive the message.
tt_message_scope_set(Tt_message m, Tt_scope s)	Sets the recipients who are to receive the message (file, session, both).
tt_message_sender_ptype_set(Tt_message m, const char *ptid)	Sets the ptype of the application that is sending the message.
tt_message_session_set(Tt_message m, const char *sessid)	Sets the session to which the message is scoped.
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 be the receiving application.
tt_message_user_set(Tt_message m, int key, void *v)	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_request_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_request_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 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.

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