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.

11

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 *nth* 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.

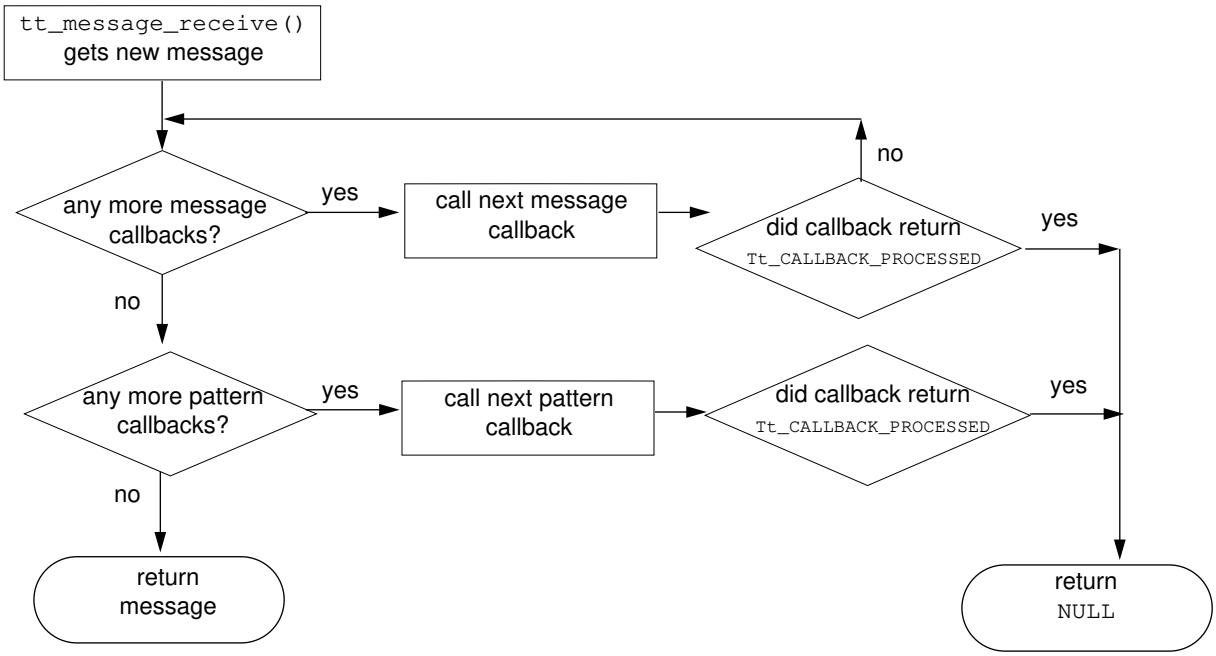

Figure 11-4 How Callbacks Are Invoked

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
tt_message_arg_mode(Tt_message m, int n)	The argument mode (in, out, inout). †
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 string. ‡
tt_message_arg_xval_set(Tt_message m, int n, xdrproc_t xdr_proc, void *value)	‡

† 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 be 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.

11