Frequently Asked Questions D

This appendix contains answers to the following questions about the ToolTalk service:

What is the ToolTalk service
Is the ToolTalk Service the SunSoft implementation of the Common Object Request Broker Architecture (CORBA)
What files are part of the ToolTalk service
Where is the initial X-based ttsession started
Where is rpc.ttdbserverd started
Where are the ToolTalk type databases stored
Do I need X Windows to use the ToolTalk service
Can I use the ToolTalk service with MIT X
Where is the session id of the X-session
How does tt_open connect to a ttsession
After calling tt_open, when does a session actually begin
If another session is attached, does the first session get killed
How can processes on different machines communicate using the ToolTalk service
What is the purpose of tt_default_session_set
How can a process connect to more than one session
Can you start a ttsession with a known session id
What information does a session id contain
Is there a standard way to announce that a new program has joined a session

D

Where is my message going
What is the basic flow of a message
What happens when a message arrives to my application
How can I differentiate between messages
Can a process send a request to itself
Can I pass my own data to a function registered by tt_message_callback_add
How can I send arbitrary data in a message
Can I transfer files with the ToolTalk service
How are memory (byte) ordering issues handled by the ToolTalk service
Can I re-use messages
What happens when I destroy a message
Can I have more than one handler per message
Can I run more than one handler of a given ptype
What value is disposition in a message
What are the message status elements
When should I use tt_free
What does the ptype represent
Why are my new types not recognized
Can I declare a ptype that is not in the types database
Is ptype information used if a process of that ptype already exists
Can the ptype definition be modified to always start an instance (whether or not one is already running)
What does tt_ptype_declare do
What is _SUN_TT_TOKEN
When are my patterns active
Must I register patterns to get replies
How can I observe requests
How do I match to attribute values in static patterns
Why am I unable to wildcard a pattern for TT_HANDLER
Can I set a pattern to watch for any file scoped message
Is file scope in static patterns the same as file_in_session scope
What is the difference between arg_add, barg_add, and iarg_add
What is the type or vtype in a message argument

D

How do I use contexts
How does ttsession check for matches
How many kinds of scope does the ToolTalk service have
What are the tt_db directories, and what is the difference between the types database and the tt_db directories
What should the tt_db databases contain
What does rpc.ttdbserverd do
Do ttsession and rpc.ttdbserverd ever communicate
What message bandwidth can be supported
Is there a limit to the message size or the number of arguments
What is the most time efficient method to send a message
What network overhead is involved
Does the ToolTalk service use load balancing to handle requests
What resources are required by a ToolTalk application
What happens if the ttsession exits unexpectedly
What happens if rpc.ttdbserverd exits unexpectedly
What happens if a host or a link is down
What does tt_close do
Is message delivery guaranteed on a network
Is there a temporal sequence of message delivery
What is unix, xauth, and des
Can my applications hide messages from each other
Is there protection against interception or imitation
Where are queued messages stored and how secure is the storage
Is the ToolTalk service C2 qualified
How can I trace my message's progress
How can I isolate my debugging tool from all the other tools using the ToolTalk service
Can I use the ToolTalk service with C++
Should I qualify my filenames
Can you tell me about ToolTalk objects
Is there a ToolTalk news group

D

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 (ROB). 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
ttsession	Communicates on the network to deliver messages.
rpc.ttdbserverd	Stores and manages ToolTalk object specs and information on files referenced in ToolTalk messages.
ttcp, ttmv, ttrm, ttrmdir, tttar	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
```

D

▼ 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 ptype. Once the start ptype is determined, tt 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
procid	Set by `tt_open`. This variable identifies the client to ttsession.
ptype	Set by `tt_ptype_declare`.
file	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 ttsession 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 ptype 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 ttsessions.
- 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 ptype represent?

Ptypes are programmer-defined strings that name tool kinds. (You can roughly translate ptype as *process type*.) Each ptype can be associated with a set of patterns that describe the messages in which that particular ptype is interested and a string for the ToolTalk service to invoke when an instance of that ptype needs to be started.

The main purpose of ptypes 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 ptype and ToolTalk will start the tool when necessary. Since the ptypes 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.

D

▼ 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 dbserver 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 dbserver 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 ttsession check for matches?

Table D-4 describes the various ways ttsession checks for matches.

Table D-4 How ttsession 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	• 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	=> Match
	• If sig prototype (number, type and mode of args) have different values	=> No Match
	• If the sig contexts are a subset of the contexts in the message	=> Match
	• 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, ttsession 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.	
TT_OBJECT & TT_OTYPE	Checks whether the otype argument is filled in	
	• If sig has a different otype	=> No Match
	• If sig has no otype & scope is different	=> No Match
	Otherwise, matches in the same manner as for `TT_PROCEDURE` matching.	

▼ 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 several100K 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 procids in the crashed session
- Outstanding requests from procids in the crashed session
- Messages that were passed the `tt_message_send_on_exit` call by procids 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 redispatched 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 ptype 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 dbserver 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.

D

▼ 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.

D

▼ 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*.

D