ToolTalk

The SunSoft Developer Engineering team is dedicated to helping developers migrate their software applications to SunSoft products, and will provide the answers to your questions. Please submit your queries to us via E-Mail at flash@doctor.eng@Sun.COM. We will attempt to answer all questions posted to us, but we may not have room to print all responses.

WHAT

What is ToolTalk?

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 ToolTalk Sun's implementation of ORB?

No, ToolTalk is not Sun's CORBA compliant ORB; that is DOMF, which is not yet shipping. ToolTalk was designed and shipped in 1991, before the OMG's CORBA spec was even defined. SunSoft is publicly committed to support the ToolTalk API running on the Distributed Object Management Facility (DOMF) (which is SunSoft's CORBA compliant version of the OMG CORBA specification) when DOMF becomes generally available as part of Solaris. Customers who integrate their applications using ToolTalk messaging will have their investments protected in the transition to DOE.

What files are part of ToolTalk?

The files are normally in /usr/openwin/ bin, lib, share/include/desktop, and man/man1. and consist of:

- 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 that inform the ToolTalk service when files that contain ToolTalk objects or files that are the subject of ToolTalk messages are copied, moved, or removed.
- ttdbck — A database check and recovery tool for the ToolTalk databases.
- tt_type_comp — A compiler for ptypes and otypes. It 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 — These files are the application programming interface (API) libraries and header file that contain the ToolTalk functions used by applications to send and receive messages.

Is Solaris 2.3 configured to execute on both UniProcessors (UP) and MultiProcessors (MP) without recompilation (i.e., does a single binary version run on both types of machines)?

We support five varieties of SPARC machines:

- Sun-4 (the 4/260, 4/110, 4/390, 4/490 series, etc.)
- Sun-4e (the SPARCengine® 1/e VME 6u boards)
- Sun-4c (SPARCstation™ 1, 1+, 2, IPC, IPX, ELC, etc.)
- Sun-4m (SPARCclassic™, LX, SPARCstation 10, SPARCserver™ 690, etc.)
- Sun-4d (SPARCcenter™ 2000, etc.)

Only the Sun-4m and Sun-4d are multiprocessor architectures. All Sun-4m and Sun-4d machines, UP and MP, run the same binary.

What extensions are provided in Solaris 2.3 for priority control commands to support multiprocessing? For example, does priocntl(1) affect threads in addition to processes?

To answer this question requires some knowledge about the model of multithreading in Solaris 2.3. Threads in Solaris 2.3 are user-level entities that are implemented by a user-level library. It is this user-level library that schedules threads onto kernel level entities, called Light-Weight Processes (LWPs), that really represent execution resources. The priocntl() interface changes the scheduling behavior of LWPs.

Applications use the thread interfaces. Most threads in an application do not require special scheduling behavior. They can execute on a symmetric pool of LWPs. If the application needs to create a special thread that has unique scheduling behavior like real-time, then the application can create a thread that is permanently bound to an LWP. A priocntl() with the appropriate parameters can be done from this bound thread to change the bound thread's scheduling behavior to real-time.

In addition, priocntl() can affect the entire process. This means all the LWPs in the process will be changed to have the selected scheduling behavior.

Does a light-weight process executing in kernel mode determine which thread to run, or does the user library decide?

LWPs represent execution resources which are scheduled by the kernel onto CPUs. Threads are also like execution resources; however, they are scheduled by a user library onto LWPs. This is usually referred to as a two-level model of multithreading. These threads have full access to system services. They are truly independent threads of control within a process. The process does not block when one of its threads block.

What are the differences between Sun's thread implementation and USL's?

First, Unix International adopted the Solaris model of multithreading. From the API point of view we are identical. USL's implementation of user-level threads is based on an early version of SunSoft's implementation.

WHERE

Where is the initial X-based ttsession started (on Sun machines)?

$OPENWINHOME/lib/openwin-sys contains an entry like:

```
# Start ttsession here.
ttsession -s
```

The first call to tt_open() would autostart ttsession if none were running, so this entry is not strictly necessary.

Where is rpc.ttdbserverd started (on Sun machines)?

/etc/inet/inetd.conf contains an entry like:

```
# Sun ToolTalk Database Server
100083/1 stream rpc/tcp wait root /usr/openwin/bin/
rpc.ttdbserverd rpc.ttdbserverd
```

Where are the ToolTalk type databases stored?

TTPATH tells the ToolTalk service where the ToolTalk Types databases reside. The format of this variable is: `userDB[:systemDB[:networkDB]]`

NOTE: Type files are read in reverse order of TTPATH so thus entries to the left shadow those to the right. This also tells ToolTalk where to search for database server redirection files. The default locations are: user ~/.tt, system /etc/tt, and network $OPENWINHOME/etc/tt

X WINDOWS / OPENWINDOWS

Do you need X Windows to use ToolTalk?

ToolTalk does not use X messages or protocols to deliver messages. X servers per se are not important to ToolTalk's operation. The X server is used as a convenient common location to park the session name. ToolTalk is only associated with X when there is an X session. What distinguishes an X session from a process tree session is the way the name of the session is advertised to its members. In a process tree session, the session name is advertised in the environment variable _SUN_TT_SESSION. Therefore, everybody in the tree of processes descending from the process that set the environment variable gets that session as the default session — which is why it is called a process tree session. (Of course, you can explicitly set _SUN_TT_SESSION to session id also.) In an X session, the session name is advertised as a property (also named _SUN_TT_SESSION) on the root window of the X server. Therefore, every process which names that X server as its display gets that X session as default session.

You need an X server to have a X session; but since the X session is defined to be the group of processes displaying on a particular X display, this is true by definition, not because ToolTalk depends on X in any way.

If there is no X server running at all — for example, a session consisting entirely of character-mode applications running on a dumb terminal — then a “process tree session” is what to use. To start a process tree session, use “ttsession -c” which starts the ttsession server running and fork a shell with the environment variable _SUN_TT_SESSION set to identify that session. Processes started under that shell locates the ttsession by consulting the environment variable.

Since ToolTalk does use Xlib when run in “X session” mode, libtt does have external references to Xlib. If you run ttsession in the other, non-X modes then ToolTalk never calls these Xlib routines; however, at link time, the reference is there, so the library must be there to satisfy the linker, whether or not it is ever called.

Can I use ToolTalk with MIT X?

Yes, just make sure LD_LIBRARY_PATH points to $OPENWINHOME/lib so libtt.so.Customer doesn't want to load all of OpenWindows.

Do I have to load all of OpenWindows to use ToolTalk?

No, but you do have to load at least the following binaries and libraries from the OpenWindows distribution:

```
$OPENWINHOME/lib
   libtt.so*
   libI18N.a [on 4.x systems only]
   libce.so*
   libX11.so.*
   libX11.sa.* [on 4.x systems only]
$OPENWINHOME/bin
   ttsession
   tt_type_comp
$OPENWINHOME/include
   desktop/tt_c.h
```

It's possible that there are one or two other shared libraries that need to be loaded.

NOTE: Since libX11.so.4.3 is required for ToolTalk clients even if no X session is involved!

Since ToolTalk doesn't know at link time whether libX11 is needed, the library must always be there, even though it may never be called. [This is fixed in ToolTalk 1.0.1 for Solaris 2.0, by dlopen'ing libX11 only when it's really needed.] 1.0 can be found at runtime.

Is it necessary to link ToolTalk programs with the xview library?

No. ToolTalk uses some internationalization (NOTE TO REVIEWER FROM DEBRA: SHOULD THIS BE INTERNATIONALIZATION OR INTERNALIZATION???) library routines which are shipped with OpenWindows V3 (because the internationalization routines in SunOS 4.1 are basically broken). These routines happen to be in libxview, so linking with libxview makes libtt happy; but if you don't want to link with libxview, link with “libI18N” which is also in /usr/openwin/lib (but is much smaller since it contains just the internationalization routines.) You still have to link with libX11, though — since the determination of whether an X session or a process tree session is not made until run-time, we have to link libtt with Xlib just in case we need to make the calls. (In Solaris 2, we paper over this by doing a dlopen() of libX11, so the application programmer won't see that libX11 is being used.)

The OW3.0.0 (SunOS 4.x) version of ToolTalk requires that some of the OW run-time libraries be present, but it does not require an X server to be running. This restriction (that the libraries be present) goes away in OW3.0.1.

Where's the session id of the “X session” I keep reading about?

To get this identifier, do xprop -root | grep _SUN_TT_SESSION. Note that the *only* thing that makes a session an X session is the way a session identifier is advertised — on the _SUN_TT_SESSION property of root window. Such a ttsession is usually started when the window system is started.

SESSIONS

How does tt_open connect to a ttsession?

After some internal initialization, tt_open tries to find a ttsession. To do this, it first checks to see if the environment variable _SUN_TT_SESSION is set. If so, it assumes that its value is the id of the ttsession to use. If _SUN_TT_SESSION isn't set, it will check to see if the DISPLAY environment variable is set. If so, it sees if the _SUN_TT_SESSION property on root X window (of the machine running the display) is set. If it is, it assumes that its value is the id of the ttsession to use. Failing either of these, it starts a ttsession itself.

Next, tt_open 'pings' the ttsession to make sure it's actually alive and well. Then it will creates a procid. This involves checking the environment variable _SUN_TT_TOKEN to determine whether the client was started from a ‘start' command for the ptype (and if so the client is ‘committed' to the default session). Next, it creates a tcp/ip socket on the client side to which ttsession connects. Activity on the socket is noticed via the socket's associated fd [returned from tt_fd()]. This ‘backchannel' is used *only* by ttsession to notify the client of incoming messages.

NOTE: Don't do close() on the fd returned by tt_fd - use tt_close(). Failure to do this causes your file descriptor count to rise upon successive tt_open() and close() calls.

Finally, tt_open refreshes the DB hostname redirection map.

When you do a tt_open, does a session actually begin?

If the default session is a X session, and there is no ttsession running, libtt will start one. Any other kind of session requires that the ttsession be started first to get the session name.

If you attach to another session, does the first session get killed?

No. The session stays around.

How can processes on different machines communicate via ToolTalk?

There are two ways. First, they can do so by connecting to the same session — figuring out how to get the session name to all the processes is the hard part. (Coming up with a particular scheme generally requires deciding what common interest the processes have, and associating a session name with that common interest somehow.) There is no mechanism built into ToolTalk for distributing this session address, other than the possible advertisement of a session id on the _SUN_TT_SESSION property of the root windows of X servers (the 'X-session'). You have to arrange somehow for the session address to get to the client.

NOTE: Since there can be multiple ttsessions per machine, we can't just have a specific RPC number for ttsession on a machine — part of a session identifier is a 'transient' RPC number.)

You get a session name via something like “ttsession -p”, which forks off a new session and prints its name to stdout, or with “ttsession -c” which sets the environment variable $_SUN_TT_SESSION to the session id. Use some mechanism to put that session name “somewhere,” where the other processes can find it. (This mechanism is perhaps a shared file, putting it in a .plan file and using finger to retrieve it, a mail message, a separate RPC call of your own design, NIS, etc. You might also advertise newly created ttsessions by sending messages in the ttsession usually associated with a particular X server.)

An approach to do this with a well-known file in a NFS-exported file system would be to start ttsession with:

```
ttsession -p >/home/foo/sessionaddress
```

and then ensure the clients use the session address from the file. An easy way to do this would be to wrap the clients in a shell script which reads the session address and sets SUN_TT_SESSION:

```
#!/bin/csh
setenv _SUN_TT_SESSION `cat /home/foo/sessionaddress`
exec client-program
```

or have the processes use that session name in tt_default_session_set before tt_open, so they connect to that session. **Warning!** There is a bug in some versions of ToolTalk that causes tt_open to fail if the default session is different from the _SUN_TT_SESSION environment variable. The workaround is to setenv(), the environment variable, to the same value before calling tt_open. Note that when using session identifiers, you can also have processes on ONE machine using different ttsessions.

NOTE: Some versions contain bugs related to this. They have all been fixed in a subsequent release (ToolTalk 1.1), which allows a given client to open multiple sessions. In 1.0.1, you can only have one open session at a time. In addition, if you try and pass the value of _SUN_TT_SESSION to tt_default_session_set(), you get an error returned. This is because of a different bug, which expects that process tree session IDs begin with “P“ and X session IDs begin with “X“. Unfortunately what's stored in _SUN_TT_SESSION lacks the “P“ prefix. Prepending that P-space string to the value of _SUN_TT_SESSION should allow your client to pass the string to tt_default_session_set() and then to successfully call tt_open. Note that the correct string **IS** returned by tt_default_session(); it's just not displayed correctly in the environment variable. Unfortunately this cannot be changed in the environment variable because of backward compatibility issues. However, the 1.1 release prepends the "P " if it's not there, and if the passed string doesn't begin with a "X“.

A second way is by scoping to a file that is NFS mounted on the machines involved. The drawback is that it requires a file system to be NFS mounted on all the systems, and rpc.ttdbserverd running on the NFS server. Fundamentally, what happens with file scoping is that when a process registers a file scope pattern, the name of that processes' session is placed on a list in rpc.ttdbserverd associated with the file. When a file scope message is sent, ToolTalk retrieves the list of sessions for the file, and forwards the message to each of the sessions. So file scopes are built by “globbing together” session scopes. Note that what's going on here is very like the “custom session” idea above. What we did is to identify the common interest (the file); park the session name in a place associated with the file (the rpc.ttdbserverd on the file's server); and use that session name (but in a slightly different way).

What is the purpose of tt_default_session_set?

tt_default_session_set() only determines which ttsession a call to tt_open() connects to. You can think of this as a 'global' variable.

How can a process connect to more than one ToolTalk session?

There are several 'default' variables used when communicating to ToolTalk:

- procid: Set tt_open() — it's how ttsession knows who the client is
- ptype: Set by tt_ptype_declare()
- file: Set when you join a file. If file is not set in the message, the file attribute gets set to the default.

By using 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 somewhere, connect to session 2, and store the default procid somewhere. Then before interacting with a particular session, restore its associated default procid.

NOTE: Changing the default procid also changes the default file and ptype to whatever was/is associated with that procid — the default file and ptype are part of the current default procid.

Can you start a ttsession with a known session id?

No, you have to get the session id from ToolTalk. The “ttsession -p” option starts a new ttsession and prints out its session id, so the problem that remains is to get that session id published to all the possible clients. (Alternative ways to get session ids are ttsession -c and then check the value of the environment variable _SUN_TT_SESSION, and for the X session, check the _SUN_TT_SESSION property of the root X window.)

What is all that stuff in a session id?

This is *CERTAINLY SUBJECT TO CHANGE*, and *NOTHING BUT 'OH-I-SEE'S* should be derived from this, but... the session ID consists of a number of fields: Version of address format; Unix pid of process; RPC Transient Program Number; Unused version (compatibility holdover); Authorization level; User id; Host IP address; and RPC version

Is there any standard way to announce new programs joining a session?

When new processes join up, they should broadcast a notice message announcing themselves. Other processes that want to know should register patterns to observe these notices. The Desktop Message Alliance set has the “Started” message for this purpose.

MESSAGES

Where is my message going?

Sometimes you'll send a ToolTalk message and nothing seems to happen. You can watch how ttsession processes each message by putting it in trace mode. This can be done at start-up by providing the -t switch. Trace can also be toggled on and off by sending ttsession a USR1 signal (e.g., kill -USR1 <ttsession_pid>). You can also use ttsnoop (or ttmon) with very general patterns to get ideas about what's happening to your message.

What is the basic flow of a message?

<u>For Session Scoped Message Flow</u>: Client builds request message and calls tt_message_send(); TTsession finds a handler (let's assume it had to start it based on a static pattern); The handlers start up and does a tt_open(), tt_fd(). This establishes communication to ttsession.

NOTE: The environment variable _SUN_TT_TOKEN was set by ttsession when it fired up the handler, so this tt_open also lets ttsession know that this is the handler; the handlers declare it's ptype to ttsession. This causes ttsession to basically change all the static patterns for the ptype into dynamic patterns. (Note that at this point the patterns are not active yet because the handler hasn't joined the session yet.) Handler next joins session, activating patterns; TTsession sends a byte down the back channel to let the handler know that there's a message waiting; Handler notices activity on the fd, and calls tt_message_receive to get the message; If the message returned by tt_message_receive has a status of TT_WRN_START_MESSAGE, the ToolTalk service started the process to deliver a message and the process must either reply/reject/fail this message (even if it is a notice) or use tt_message_accept(). Messages for the ptype are blocked until this occurs; Handler does its processing (or could reject or fail the message); Handler returns reply to request; TTsession sends a byte down the backchannel to the client to let him know that there's a (reply) message waiting for it. This activates the tt_fd() of the client; Client does a tt_message_receive to get the result.

NOTE: All the while ttsession is honoring observer patterns, the client actually receives a message every time its request message changes state.

For File Scoped Message Flow: When a file scoped pattern is registered, libtt tells the dbserver about the file and the session its registering the pattern in; For notices, libtt checks with the dbserver to find all the sessions that have clients who have registered interest in the file, and then communicates with all these sessions directly; For requests, libtt checks with the dbserver to find all the sessions that have clients who have registered interest in the file, and then tells its session the message and the list of other sessions involved. The sessions communicate among each other to find a handler (looking first for a handler in the clients session).

What goes on when a message arrives to my application?

1. A message arrives for your program.

2. The file descriptor (tt_fd()) becomes active.

3. The Xt main loop breaks out of its select and calls the function registered by XtAppAddInput (call that the input callback).

4. That function calls tt_message_receive().

5. As part of tt_message_receive(), the message is read in and any callbacks associated with the message are run. This is where the message callback (if any) is executed.

6. The message callback returns. If the message callback returned TT_CALLBACK_PROCESSED, tt_message_receive returns to the input callback with a value of NULL. If the message callback returns TT_CALLBACK_CONTINUE, a TT_message handle for the message is returned. The input callback then continues with any other processing put in after the tt_message_receive.

So if the input callback looks like:

```
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 message callback looks like:

```
message_callback(...)
{
  printf("message callback entered\n");
  return TT_CALLBACK_PROCESSED;
}
```

you'll see the output

```
input callback entered
message callback entered
input callback exiting, message handle is 0
```

How can I tell messages apart?

First, messages have an identifier (obtainable from tt_message_id(my_msg)) that uniquely identifies the message across all running ttsessions.

Second, you can put some information on a user cell (tt_message_user... calls) to tie the message to the application's internal state.

Also, message handles remain the same. So for example, if you want to see if the message you received is the same one you sent out:

```
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. The only problem we've seen is that usually the handler's pattern callback does something like:

```
{  ...
   tt_message_arg_val_set(m, 1, "answer");
   tt_message_reply(m);
   tt_message_destroy(m);
   return TT_CALLBACK_PROCESSED;
}
```

Which is fine *except* in the case where the handler and the sender are the same process. Then, when the reply comes back (to the same process) the message has already been destroyed, so any callbacks, user data, etc. that the sender attached to the message are gone.

The easiest way to avoid this is to *not* destroy the message if you sent it:

```
{
   ...
   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()?

Use the user data cells on the message (see tt_message_user_set() and tt_message_user()).

NOTE: User data can only be seen in the client where the data is sent.

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);
...
}
```

How can I send arbitrary data in a ToolTalk message?

ToolTalk has no built-in way of sending structs, only strings, ints, and byte arrays. The recommended way to send structs is to use XDR to turn the struct into a byte array, put the bytes in the message, and use the same XDR routine on the other side to deserialize.

Are single-threaded drivers supported in Solaris 2.3? Will existing drivers work? What is the implementation (i.e., how is single-threaded driver support provided)?

There is some support for what we call “MT-unsafe” drivers. Whenever such a driver is active, it holds a global mutex, thus preventing any other thread from entering that driver or any other MT-unsafe driver. There are so many restrictions on MT-unsafe drivers that we've decided not to document this support. Driver writers must provide their own locking.

Is there any way that files can be transferred using ToolTalk?

Not directly. You could either put the file data in a message argument, put the file name in a message argument or put the file name in the tt_message_file attribute. Each of these has advantages and disadvantages:

File data in message argument: ToolTalk has to copy 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. This is a lot of copying and can be slow and chew up memory if the data is large.

File name in message argument: assumes that every receiver mounts the file, and in the same spot.

File name in tt_message_file attribute: assumes that every receiver mounts the file, but at least ToolTalk will take care of mount point differences

How are memory (byte) ordering issues handled by ToolTalk?

ToolTalk lets you put ints, strings, and byte vectors into messages. We XDR them so that they will come out correctly for each client. So integers come out correctly due to the magic of XDR, and reading/writing strings or byte vectors from/into ToolTalk messages is just like reading/writing from/into a file.

But if you have data that isn't one of these three data types, then before you put it into a message you should serialize it into a byte vector using XDR or something similar.

Can I re-use messages?

Messages cannot be sent multiple times with different args; they must be iteratively created, sent, then destroyed.

When I destroy a message, what's really happening?

When you do the tt_message_destroy you destroy that handle but not the underlying message. The underlying message only goes away when ToolTalk is finished with it and all the external handles are destroyed. (There's an extra level of indirection and some reference counting involved.) Let's say you destroyed a handle to a message right after sending it. Since you'll get a new handle when the reply comes back, destroying the message right after sending it makes it hard to match requests with replies, but ToolTalk won't seg fault, lose messages, or anything like that.

But note that destroying a message means we it will never be shown again, under any circumstances. So if you register a pattern to observe a request you send, and then destroy the message when your pattern matches it, you will never see the message when it is in state “handled” (i.e., when it is a reply).

Can I have more than one handler per message?

Currently no. If you want multiple processes to do the action, your only choices are to: use notices; or perform some trickery using message rejection to force ToolTalk to deliver the request to all the possible handlers, but have each of these handlers actually 'do' something.

Can I have more than one handler of a given ptype running?

It's certainly possible to have more than one handler pattern out, but ToolTalk has no concept of load balancing; ToolTalk will choose *one* of the handlers and will keep delivering additional matching messages to that one, unless something is done to stop it.

There are ways to accomplish this:

1. Use tt_message_reject(). If a message comes in and a process doesn't want to handle because the process is busy, the process can reject the message; ToolTalk will then try the next possible handler (and apply the disposition options when it runs out of registered handlers).

   This works fine, but it does require the process to still be in an event loop, calling tt_message_receive() when the tt_`fd` is active. If the process is in heavy computational loop, not looking for messages, this approach fails. [ToolTalk does have an internal time-out; if the process fails to pick up the message within five minutes, ToolTalk figures the process has hung up somehow and goes to another handler. But this time-out is too long for load-balancing use.]

2. Unregister the pattern when busy.

   The idea would be to do something like

   ```
   m = tt_message_receive();
   if (m is the message that causes us to go busy){
       tt_pattern_unregister(p);
   }
   ```

   and then register the pattern again when we want messages again. Since the pattern is unregistered, ToolTalk won't route matching messages to the process.

   This works all right *except* there's a race condition. It's possible that a second message could be sent and routed to the process in the time between the tt_message_receive() and the tt_pattern_unregister(). Depending on the exact application, this may not be fatal — after all, the message is still there queued up for the process, and the process will eventually finish the first request and start working on the second one; the only problem is the second request is unnecessarily delayed.

3. The combination.

   To be really safe, use a combination of the two techniques above. The idea would be:

   ```
   get the message
   unregister the pattern
   loop, calling tt_message_receive until it returns 0;
   reject all the returned messages [See NOTE below]
   handle the message
   re-register the pattern
   repeat.
   ```

NOTE: This assumes that the process only registers one pattern; if there are other kinds of messages, the process needs to handle those messages in the “receive/reject” loop, since there's no way in ToolTalk to block out some messages while receiving others.

What value is disposition in a message?

At first glance it seems that message disposition makes no sense since it seems the disposition specified in the static type definition is the disposition that is followed. However, if the message specifies the handler ptype (implying there is a static type definition for this ptype or the message will fail), and the message does not match any of the static signatures, the disposition set in the message will be adhered to (e.g., if disposition in the message was TT_START and the ptype specifies a start-string, it WILL be started). This effectively allows you to ‘ram a message down somebody's throat' if you know their ptype, and the message you’re sending doesn't match their listed signatures.

What are the message status elements?

Message_status_string is never used by ToolTalk. This message component is for use by the applications. Message status is set by ToolTalk only when there is a problem with message delivery. At other times it may be set/read in an application dependent manner.

What should I be tt_freeing?

libtt maintains an internal storage stack. It is from this stack that you receive data buffers (e.g., from tt_message_arg_val). Any time you call a ToolTalk API routine and get back a char * or void *, it points to a copy that you are responsible for tt_free()ing. This includes routines which have char/void * out arguments like tt_message_arg_val and tt_message_arg_bval. Mark and release are a convenient way of freeing everything allocated during some sequence of operations. Release frees *everything* allocated since the corresponding mark. This means that if you want to keep something around that you got back from ToolTalk, you should be very careful to make a copy of it before you do any operations (i.e., tt_release()) that may free it.

TYPES

What is the ptype supposed to 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 describing the messages that ptype is interested in, and a string for ToolTalk 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 tool is actually running in the scope in which the message is sent. If a tool is able to carry out a message's requested operation, or wants to be told when a particular message is sent, this is indicated in its ptype, and ToolTalk starts 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 restart ttsession, but a much better way is to make your running ttsession reread the types database by sending it the USR2 signal (e.g., kill -USR2 <ttsession pid>).

Can I declare my ptype to be something not in the types database?

Currently, no.

Is ptype information used if a process of that ptype already exists?

ToolTalk always looks for one handler and any number of observers for every message. In this case, even though ToolTalk finds a handler running, ToolTalk still looks through the ptypes for any observe patterns that match the message. If there is a ptype with an observe pattern that matches, and there is no process of that ptype currently running, ToolTalk proceeds to start a new process or queues the message as specified in the ptype pattern (or in the message).

Can the ptype definition be modified to always start (whether or not one is already running)?

No, but the implementation of the ptype is perfectly free to say “I am coded such that I can only handle one request per instance, and so in my code I will always tt_message_reject() any request I get that isn't a TT_WRN_START_MESSAGE.’

Note that this means that requests will be delivered to (and rejected by) all running instances of the ptype before a new one gets started. If you envision lots of these ptypes running at the same time, or if the message can contain lots of data, then marching each request past all these rejectors will be slow. You can also take a look at what tt_message_accept() may do for you — it basically says to 'unblock' messages to the ptype (messages to a ptype are blocked during start-up until the ptype either replies to the message, or issues tt_message_accept()).

What does tt_ptype_declare actually do?

By declaring the ptype, your static patterns (process and object types compiled with tt_type_comp and residing in the types database) now exist in ttsession memory.

NOTE: Multiple declarations by an application of the same ptype are ignored.

When a ptype is registered by an app, ToolTalk also checks for otypes that mention the ptype and registers the patterns found in these otypes. This does not mean that the static patterns are active yet though. For that, a 'join' must be performed.

How do I specify that the program should start on a specific machine? Can I specify which machine at runtime?

The start string always gets exec'd on the host on which ttsession is running. But the exec'd process is perfectly free to start another process on another host.

A simplistic way to do this is to make your start string be “rsh farhost myprog". But in order to get myprog to be in the right session and receive its initial message, you need to propagate the important ToolTalk environment variables. The appended ttrsh shell script does just that.

```
#! /bin/sh
# This command runs a command remotely in the background,
# by pointing stdout and stderr at /dev/null. (Should we point
# stdin at /dev/null too?)
# By running this through the Bourne shell at the other end,
# and we get rid of the rsh and rshd which otherwise hand
# around at either end.
# Hacked from a posting by clyde@emx.utexas.edu to list.xpert
# Hacked by holtz@eng.sun.com to export choice environment
# variables.
#
#set -x
user=
debug=
HOST=${HOST-`hostname`}
if [ "$1" = "-debug" ]; then
```

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

What is the deal on _sun_tt_token ?

When processing messages that require an application to be ‘started,' ToolTalk sets this environment variable in the child process. When the application starts and performs tt_open, this information is passed back to ToolTalk so it knows that the application coming up is the one ‘started' (or delegated) to handle the message.

PATTERNS

When are my patterns active?

If the session / file / context is already set in a pattern when it is registered, it becomes active upon registration. 'Joining', adds the session / file / context to ALL patterns that have been registered by your program.

NOTE: Contexts are not scopes. A pattern that is joined to some contexts but not joined to any file or session can not match any message.

A pattern must be registered with the session in which it wants to be active. (You can't reuse patterns — e.g., create a pattern, register it with session1, change it, register it with session2, etc.) Your patterns can be active for more than one file (for a given procid). This says the file part of the pattern will match any of the listed files (an ‘OR' relationship). Your patterns can restrict their matching to particular context. Every context in the pattern must have a match in the message (an ‘AND' relationship).

NOTE: Contexts are new for 4-93.

Must I register patterns to get replies?

No, never. If you do register a pattern that matches the reply, then the reply will come through your event loop twice; once because it matched a pattern, and once because it's a reply.

How can I observe requests?

Observers *can* observe requests if the pattern matches (and the message is not point to point (i.e. address=TT_HANDLER). If you have problems, you may want to run with ttsession -t on to see why your observer pattern isn't matching.

How do I match on attribute values in static patterns?

The ToolTalk static pattern (i.e., types database) mechanism does not allow you to specify attribute values to match by. That is, you can match by file scope or argument vtype, but not by the particular filename or by argument value. The same applies for contexts in static patterns.

Why doesn’t my pattern for tt_handler 't wildcard?

Because TT_HANDLER messages basically aren't pattern matched at all. (Slightly changed for 10/93 to match against the handler's patterns so callbacks can get run, but still doesn't allow other processes to snoop on TT_HANDLER messages.)

Can I set a pattern to watch for any file scoped message?

No. Using file scope without a file name is like saying I want to match file scoped messages with every file in the universe. We just don't provide that sort of functionality. Note also, that a session attribute may be set on a FILE scoped pattern to provide a file-in-session-like scoping, but a tt_session_join will not update the session attribute of a pattern that is TT_FILE scoped.

In static patterns isn't file scope the same as file_in_session scope?

Let’s say we have a file foo.bar, and assume no session has any clients who have registered interest in foo.bar. Assume that all sessions have the same static patterns (for now), and that there's at least one pattern P that will match a message M that we will be sending.

I'm connected to session A, and I issue a file-scoped message M for file foo.bar. Now, since no client of any session has ever expressed any interest in foo.bar, session A is the only one that will get the message. The message will match against static pattern P in session A (the reason I think of file==file_in_session at this point). Once the ptype gets fired up, the pattern really does become scoped to file (within that session). Now, since all the sessions have the same static pattern set, session A will actually honor all the promises, taking care of the semantics we expect.

Now we eliminate the assumption that all sessions have the same static patterns. (*NOTE:* It was an assumption in the original design of ToolTalk that all sessions in a scope have identical type definitions.) If that assumption is violated, strange things are allowed to happen. Hopefully not too strange, of course. Let’s say session B has an extra pattern P', scoped to file, that should match message M. In session A, we now send M. But since no client of session B has ever expressed interest in foo.bar (hence the dbserver won't send the message to session B), we're really breaking out promise for pattern P'. If however, some client of session B *had* previously expressed interest in foo.bar, then we would have luckily kept our promise for P' since the dbserver would have known that at least one client there had expressed interest in foo.bar. The key point here is that for situations like P' in B, it seems file==file_in_session until one of two things happen: some client of B registers interest in foo.bar; or some client of B sends a message scoped to foo.bar.

What's the difference between arg_add, barg_add, and iarg_add?

barg_add and iarg_add are basically a quick way of doing an arg_add followed by a set of the value. Note that for patterns vtype == "ALL" matches any vtype and Value == NULL, matches any value.

What exactly is the 'type' or 'vtype' in an message argument?

It's whatever your application decides — this is not C typing! This is for application type space use. Setting vtype to ALL matches any vtype.

Vtype is short for "value type," and indicates the semantic domain in which the argument's value has meaning. Every vtype should by convention correspond to one and only one of the three possible data types for argument values.

Vtypes are analogous to typedefs in C. In C, there are a few fundamental types (int, unsigned long, void *, etc.), which tell the compiler what it means, say, to apply the plus operator to two objects of a given type. The typedef mechanism allows the C programmer to say "the custom type Tt_message looks just like the fundamental type void *, but I attach special semantics to Tt_message that I don't attach to void *, and I want you to slap my hand when I try to use a non-Tt_message as a Tt_message, even if that non-Tt_message is a void * (which is in your eyes the same as a Tt_message)". The vtype mechanism lets the ToolTalk protocol designer say the same sort of thing: "I'm proclaiming that the vtypes 'messageID' and 'bufferID' are both just C strings, but they have different semantics. Some operations are valid on one, some on the other, and some on both, and I want to use the pattern-matching mechanism to make sure that a request with a bufferID string doesn't get matched to a pattern for an operation that is only valid on messageID strings."

What are contexts for?

(new for 1.1) Contexts can be used to restrict matching. They 'and' together: i.e., 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 $ character, for example $ISV, and the message causes an application to be started, the application will have the environment variable ISV set to whatever value the context slot had.

Basically, how does ttsession check for matches?

TT_HANDLER Point to point requires no matching — it is passed directly to the receiver. Note that this means that you can't monitor point to point messages since any patterns you register to do so are never checked. TT_OBJECT & TT_OTYPE — otype must be filled in for object messages, then do PROCEDURE matching. TT_PROCEDURE — Scan list of static sigs (signatures) that have same op, collecting lists of observers and potential handlers; If sig has otype, & it doesn't match => Mismatch (picks up scope from otype signature); else If sig has no otype & scope doesn't match => Mismatch; If sig has no args and no contexts => Match; If sig prototype (number, type and mode of args) don't match => Mismatch; Sig.contexts a subset of those in the message => Match; Save info for any static observers that require queuing; Scan through dynamic patterns, adding to lists of observers and potential handlers; Use ones with op then ones without to form list; Check reliability, states, class, address, handler, handler ptype, scope, object, otype, sender, sender_ptype, args, contexts (wildcard or value(s)); Delivery - Send to Observers(observers first since handler could change state), then Handler with best match; If more than one Handler at 'best match', one will be chosen arbitrarily;

SCOPE

What kinds of scope are there?

Currently there are only two kinds of scope in ToolTalk: session and file. (Sometimes X session is referred to, but this is really just the same as session scope, except its session id is advertised on the _SUN_TT_SESSION property of the root X Window.)

How does file scoping work?

NOTE: The bug mentioned in the following was fixed in release 1.1. This describes the way ToolTalk handles file-based scoping mechanisms and when ToolTalk does and does not use the ToolTalk database server for handling file-based scoping mechanisms. Specifically, because of a bug in the current release, ToolTalk does not use the DB server for any type of scoping (other then TT_OBJECT scoping) unless an explicit tt_file_join call is made.

The following is an explanation of when the DB server is and is not used and how the DB server is used for scoping and how the various file-based scoping methods work.

<u>Brief Overview of the ToolTalk DB Server Duties</u>:

In the current release of ToolTalk, the DB server is used to store three types of information:

1. Storage of ToolTalk objects specs (see the tt_spec_write and the other tt_spec_* API calls).

2. Storage of ToolTalk session IDs of sessions with clients that have joined a file via the tt_file_join call.

3. Storage of a file-scoped (NOT file-in-session scoped) message that is queued because the message disposition is TT_QUEUED and a handler has not yet been started that can handle the message.

<u>File-based Scoping in Patterns</u>:

With patterns, there three types of scopes that use files:

1. TT_FILE, TT_BOTH, TT_FILE_IN_SESSION, TT_SESSION+tt_pattern_-file_add()

2. TT_FILE scopes to just a file. A session attribute may be set on this type of pattern to provide a file-in-session-like scoping, but a tt_session_join will NOT update the session attribute of a pattern that is TT_FILE scoped.

3. TT_BOTH scopes to the union of interest in the file and the session. A pattern with just this scope will match messages that are scoped to either the file and/or the session.

TT_FILE_IN_SESSION scopes to the intersection of interest in the file and the session. A pattern with just this scope will only match messages that are scoped to both the file and session.

Using the TT_SESSION scope and calling tt_pattern_file_add provides exactly the same type of pattern scoping as TT_FILE_IN_SESSION, except, if a tt_file_join call is made, it will add files to TT_FILE, TT_BOTH, and TT_FILE_IN_SESSION scoped patterns, but NOT to TT_SESSION scoped patterns. This is important because patterns that were registered via a tt_ptype_declare call can only have files added to them via a tt_file_join call.

In the case of the TT_FILE_IN_SESSION scope, all you need to do is create a pattern, add the TT_FILE_IN_SESSION scope to the pattern, add a session to the pattern (or call tt_session_join after registering the pattern), add a file to the pattern, and then register the pattern. The following two examples register an equivalent pattern (sans error checking):

Example 1:

```
tt_open();

Tt_pattern pat = tt_create_pattern();
tt_pattern_scope_add(pat, TT_FILE_IN_SESSION);
tt_pattern_file_add(pat, file);
tt_pattern_session_add(pat,tt_default_session());
tt_pattern_register(pat);
```

Example 2:

```
tt_open();

Tt_pattern pat = tt_create_pattern();
tt_pattern_scope_add(pat, TT_FILE_IN_SESSION);
tt_pattern_file_add(pat, file);
tt_pattern_register(pat);
tt_session_join(tt_default_session());
```

Because TT_FILE_IN_SESSION scope is the intersection of file interest and session interest, all the information required to properly match similarly scoped messages can be stored in the ToolTalk session and the ToolTalk DB server is never contacted. If TT_FILE scoping had been used, the second example would NOT have updated the session attribute in the pattern.

If ToolTalk had no bugs, then setting the scope of a pattern (via tt_pattern_scope_add) and setting the session and/or file of a pattern (via tt_pattern_session_add and/or tt_pattern_file_add) would be all you needed to do and the pattern scopes would match the message scopes as described above. However, because of a bug in ToolTalk, if you do not explicitly join a file, the above scopes will only match messages that are scoped to both the file and the session of tt_open the pattern was registered under. This means that if you do not make an explicit call to tt_file_join, TT_FILE and TT_BOTH effectively act like TT_FILE_IN_SESSION.

The reason this occurs is because of how the ToolTalk DB server is used. When a message is scoped to TT_FILE or TT_BOTH, the ToolTalk client library that is sending the message (this is where the tt_message_send call was made) contacts the ToolTalk DB server and asks for a list of ToolTalk sessions that have registered interest in the file the message is scoped to. The client library can then send the message to each ToolTalk session in the list. When a ToolTalk session receives the message from the sending client, it can then match the message to patterns that have been registered with the ToolTalk session and send the messages to any clients that have patterns that match. The problem is that the ToolTalk session that a pattern is registered on is not stored on the ToolTalk DB server until a tt_file_join call is made, not when an actual pattern is registered.

The reason this is a bug is because when a tt_file_join call is made, this implicitly does a tt_pattern_file_add to every pattern that has already been registered and effectively re-registers the patterns. This means that there is no way to create several different patterns in one ToolTalk proc that all scope to different files. For example, if I make the following calls:

```
tt_open();

Tt_pattern pat1 = tt_create_pattern();
tt_pattern_scope_add(pat1, TT_FILE);
tt_pattern_register(pat1);
tt_file_join(file1);

Tt_pattern pat2 = tt_create_pattern();
tt_pattern_scope_add(pat2, TT_FILE);
tt_pattern_register(pat2);
tt_file_join(file2);

Tt_pattern pat3 = tt_create_pattern();
tt_pattern_scope_add(pat3, TT_FILE);
tt_pattern_register(pat3);
tt_file_join(file3);
```

Pattern pat1 will actually be scoped to 3 different files (file1, file2, and file3). Pattern pat2 will be scoped to 2 files (file2 and file3), and pattern pat3 will be scoped to just file3. I can't use just tt_pattern_file_add to set individual files on individual patterns, because a tt_pattern_file_add doesn't cause the pattern's ToolTalk session to be stored in the database. Only a tt_file_join call will do that. This bug will be fixed in a future release of ToolTalk.

Some may ask, if you fix that bug, how will I scope to the union of TT_FILE_IN_SESSION scope and TT_SESSION scope with one pattern? Currently, if I use TT_BOTH on a pattern, register the pattern, and do NOT call tt_file_join, it effectively provides that ability.

The answer is, when the bug is fixed you can still scope to the union of TT_FILE_IN_SESSION and TT_SESSION by adding both scopes to the same pattern:

```
tt_open();

Tt_pattern pat = tt_create_pattern();
tt_pattern_scope_add(pat, TT_FILE_IN_SESSION);
tt_pattern_scope_add(pat, TT_SESSION);
tt_pattern_file_add(pat, file);
tt_pattern_session_add(pat, tt_default_session());
tt_pattern_register(pat);
```

This is the way you should implement the union of the scopes. When the file scoping bug is fixed, just adding the scope TT_BOTH will have a different behavior than it does right now. However, adding TT_BOTH and then calling tt_file_join after registering the pattern will have exactly the same behavior as it does now.

File-based Scoping in Messages:

Messages have the same types of file-based scoping mechanisms as patterns:

- TT_FILE - The message is scoped to all clients that have registered interest in a file.
- TT_BOTH - The message is scoped to all clients that have registered interest in the message's session and/or file.
- TT_FILE_IN_SESSION - The message is scoped to all clients that have registered in interest in both the message's file and session.
- TT_SESSION + tt_message_file_set() - I've listed this as file-based scoping mechanism because of the call to tt_message_file_set. This will scope 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 get the file name by calling tt_message_file.

The bugs that applied to file-scoped patterns do NOT apply to messages. When a message is TT_FILE or TT_BOTH scoped, the ToolTalk client library will check the database server for all sessions that have clients that joined the file and send the message to all of the interested ToolTalk sessions. The ToolTalk sessions will then match the messages to the appropriate clients. NO explicit call to tt_file_join is required.

In the case of a TT_FILE_IN_SESSION scoped message or a TT_SESSION scoped message that has a file set in it (via tt_message_file_set), the database server is not contacted and the message is only sent to clients that are scoped to the message's session.

What is the difference between the types database and the tt_db directories? What is the tt_db directory I see at the root of my partitions?

The types database (databases actually, since you can have user, host and network components) store the static ptype and otype definitions. These declare what messages applications and objects will respond to. tt_type_comp modifies this database when you add/delete/etc. static type definitions. Ttsession reads in these type files when it starts. These static type databases are stored in by default: for user ~/.tt, for system /etc/tt, and for network $OPENWINHOME/etc/tt (see TTPATH).

TT_DB is a database created by rpc.ttdbserverd. It contains the associations between files in this partition, and sessions with patterns interested in these files, as well as all the object spec information for file in this partition.

What does a healthy tt_db database look like?

For versions 1.1 or later, healthy databases should 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, and property_table.var, with permissions set to -rw-r--r-- For versions before 1.1: the following ten files are present: docoid_path.ind, docoid_path.rec, docoid_path.var, oid_access.ind, oid_access.rec, oid_container.ind, oid_container.rec, oid_prop.ind, oid_prop.rec, and oid_prop.var with permissions set to -rw--------. Note that ttdbck performs maintenance on these TT_DBs.

What is rpc.ttdbserverd for?

It performs three major functional duties: Storage of ToolTalk session IDs of sessions with clients that have joined a file via the tt_file_join call; Storage of a file-scoped (NOT file-in-session scoped) message that is queued because the message disposition is TT_QUEUED and a handler has not yet been started that can handle the message; and Storage of ToolTalk objects specs (see tt_spec_write and the other tt_spec_* API calls).

Do ttsession and rpc.ttdbserverd ever communicate?

Currently, no. Only libtt communicates with rpc.ttdbserverd (not ttsession). libtt passes information back and forth when needed.

What can happen if I set /usr/tmp to a tempfs (O/S version: 4.1)?

When there's no TT_DB and you send a file scoped message (say on an existing file like /etc/fstab), the db code wants to create a file called "/usr/tmp/NETISAM.LOCKTABLE." It then tries to do a mmap on it, and after that things go downhill and the server exits. The end result is that the database only gets partially started toward being built. The next time the server fires up, he thinks the db's there, but since it's messed up, subsequent uses of the TT_DB end up failing. Check to see what and what is not a tempfs what is read only, etc. before doing file actions. Note that the 4.1 documentation highly recommends against making /usr/tmp a tempfs.

PERFORMANCE

What message bandwidth can be supported?

About 100 small messages per second in the Solaris 2.0 release, on local systems (SparcStation 2). Performance mainly depends on how many recipients each message has — i.e., notices that don't match any pattern are cheapest, messages that match many observers are the most expensive.

Say you were to go into a loop that did nothing but send messages. You are basically going to saturate the ToolTalk server (ttsession). Since ttsession is single-threaded, what ttsession is doing is:

```
do forever {
   get message from sender
   find receivers
   for each receiver {
      deliver message
   }
}
```

So the more receivers there are, the longer it takes ttsession to get back to the "get message from sender" part. So your sender basically blocks down in tt_message_send() waiting for ttsession to service the RPC call.

We normally say that tt_message_send() is non-blocking, but what we mean is that ttsession replies to the RPC call immediately, before delivering the message. So if the sending application were doing any real work, it could continue while ttsession delivers the message. But you can't send another ToolTalk message until the server finishes dealing with the previous one.

Is there a limit to message size or number of arguments?

There's no designed-in limitation to the size of a ToolTalk message or the number of arguments. However, ToolTalk does copy the data several times, both from one spot in the client's address space to another, and by sending it across the RPC connection to and from the server. So if you put, say, a megabyte of data in a ToolTalk message, that megabyte of data is going to be copied at least four times:

1. From your storage to the ToolTalk library's storage.
2. From the library to the ToolTalk server.
3. From the server to the receiver's library.
4. From the receiver's library to the final resting place.

If there are processes observing the message, even more copying will take place. (If you get really excessive, you'll run out of swap space when ToolTalk copies the argument values in.)

That's a lot more copying than would take place compared to more direct methods (like writing the data out to a file and sending the file name in a message, or sending a socket address in a message and opening a connection between the two processes). Further, since the ttsession process is single-threaded, no other messages for this session are going to be delivered while this copying is going on. So, there's a possibility of bogging down the session if you send a lot of big messages.

Since the increased load of big messages is just the time spent to copy the data, it really doesn't have a “cliff”; you can't say that “messages under 1K are fast and messages over 1K are slow.” Basically, the time required to send drops off linearly with the size of the message.

So, the principle is not “If message size is beyond X bytes, don't use ToolTalk.” It's the (unfortunately) more nebulous “if you plan to send really big chunks of data really often, you might want to think about using some non-ToolTalk way to pass the data.”

What is the most time efficient method to send a ToolTalk message?

Directly to process (TT_HANDLER) is faster than procedural messages that match only one receiver. Procedural and Otype messages are about the same. Object messages are the slowest (there's an extra I/O to look up the object type). File scope is slower than session scope. File-scoped messages are our slowest kind, but we always get at least 5/sec when we measure them, and we have a demo that runs fine blasting out several file-scoped deliveries per keystroke in an editor at touch-typing speeds.

What's the network overhead involved?

The message is sent directly to the ttsession process for the session. (This could be 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. So if a process off on another machine isn't interested in a message, nothing on that machine may have to wake up and look at it.

ToolTalk does *not* use hardware broadcast or multicast, so there's no overhead of sending out the message and having everybody look at it and decide if they want it. [On the other hand, if hundreds of processes register interest in the same message, ToolTalk will have to send the message out hundreds of times. ToolTalk is designed on the premise that each message is only interesting to relatively few processes.]

Does ToolTalk do load balancing for handling requests?

ToolTalk is not a load-distribution mechanism. If there are two processes with identical patterns registered, ToolTalk ends up just picking one of them and deliver all the messages to it. If you want to do load distribution, you have to do more work, like unregistering the pattern while the process is busy, and rejecting any messages that got in before the pattern was unregistered.

What resources are required by a ToolTalk application?

Coarse numbers say it takes several 100K of working set for a sending client, ttsession, and a receiving client to process messages. ToolTalk's memory requirements don't grow over time, so performance concerns should be addressed by faking the traffic you anticipate and seeing if you like the performance.

RELIABILITY

What if ttsession gets killed?

For versions 1.1 and beyond (also 1.0 patch level 4): When ttsession is killed, the tt_fd becomes active, and most ToolTalk API calls return TT_ERR_NOMP (that's “No Message Passer”, Message Passer being another name for the ttsession process). Most applications just assume that this means something bad has happened to ttsession, and stop trying to send/receive ToolTalk messages. If an application wants to recover from this situation, it may be possible. Some of the things it would do include:

- Recognize the TT_ERR_NOMP situation
- Do a tt_close() to clean up the connection from its end

- Redo the ToolTalk initialization sequence:
    - tt_open, tt_default_session_join, tt_fd

      (Note that what tt_open does varies with the setting of the environment variable _SUN_TT_SESSION, the value of the _SUN_TT_SESSION property of the root X window (if it exists), some clever manipulation of these is likely necessary when restarting a crashed ttsession to take over where the last one left off. Also, other participants of the crashed session must somehow be made aware of the restarting and the new session id so that they can recover.)
    - Re-register all patterns and re-declare ptypes.
- Continue on

The following are lost when ttsession crashes:

- Patterns registered by procids in the crashed session
- Outstanding requests from procids in the crashed session
- Messages bequeathed via tt_message_send_on_exit() by procids in the crashed session.
- Session props
- Session-queued messages

What if rpc.ttdbserverd gets killed?

If rpc.ttdbserverd exits unexpectedly, there may be no effect at all, since inetd (Suns) will start a new one to replace it. The worst that is supposed to happen is that one or more API calls dealing with specs or scoped-to files on that fileserver 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. If the dbserver crashes, data may be temporarily unavailable as zero or more API calls will fail with TT_ERR_DBAVAIL, but no data will be lost.

What if a host is down or a link is down?

When TCP notices this, the TCP connection breaks. When a process' connection to ttsession breaks, ttsession acts as if the process exited. All the patterns are cleaned up. The process will get TT_ERR_NOMP next time it tries to send or receive.

What's tt_close do and what happens if I don't call it?

On tt_close(), ttsession only closes the current procid, and if that was the last procid to close, all other baggage (structures, etc... all the tt structures created since tt_open()) are cleaned up/out/destroyed. If your program doesn't call tt_close? - if tt_message_ send_on_exit() was called previously, then the queued message is sent out before cleanup. Note: tt_message_send_on_exit() first appears in ToolTalk 1.1.

NOTE: Don't do close() on the fd returned by tt_fd - use tt_close(). Failure to do this will cause your file descriptor count to rise upon successive tt_open() and close() calls.

Is message delivery guaranteed on a network?

Messages are sent via RPC on TCP/IP, so delivery is reliable.

Is there a temporal sequence of message delivery?

Between a given sender and receiver, message sequence is preserved. That is, if process A sends messages M1 and then later M2, and process B receives both M1 and M2, process B will get M1 before M2. There are two exceptions, both fairly specialized:

1. If a hander receives a message and then rejects it (with tt_message_receive) the request gets redispatched to a second process. In the meantime, (while the first handler is mulling over the request and deciding whether to reply or reject) ToolTalk continues to deliver messages. Those messages can appear to “pass” the first request.

2. If message queueing is used, a process will receive queued messages when it declares a ptype containing the pattern that caused the queuing. That might happen after the process has already received other messages from the sending process...

SECURITY

Can ToolTalk use secure rpc?

Yes, if you use the “-a des” option to ttsession is started. To put options on ttsession you either have to start ttsession “by hand” (for example, put “ttsession -a des” in .xinitrc) or set the environment variable “SUN_TTSESSION_CMD” to the ttsession command to use if libtt has to start ttsession automatically.

What's unix, xauth, and des about?

For starters, we should point you to the rpc_clnt_auth(2), xauth(1), and secure_rpc(3)/des(1) man pages, respectively, for the three kinds of authentication. In a nutshell, unix authentication is when an entity making an rpc call on you tells you what uid it is, and you believe it. xauth authentication is a way of controlling access to your X display (and thus to your ttsession) by means of a read-protected file in your home directory. des authentication uses the Data Encryption Standard to make sure processes talking to ttsession really are who they say they are.

The dbserver enforces security on each RPC call (such as the ones made by tt_spec_write() or tt_file_join()). The dbserver uses AUTH_UNIX security by default, or no security with the -n option.

Can my applications hide messages from each other?

No. Security in Unix focuses on protecting you (the user) from other users, and not on protecting you from yourself — and the applications that you choose to run. ToolTalk intentionally provides no mechanism for letting one application lock out others from seeing its messages. We want to enable third parties to provide applications that listen in on the message stream and respond with functionality that the authors of the sending application hadn't envisioned. It is up to the user (not the authors of the sending application) to decide whether this third party functionality is desirable or not; the user is free to install or deinstall it based on judgement.

We provide no hooks for easily encrypting data you give us, but nothing can stop you from encrypting data before you give it to us, and decrypting it when we give it back. Public-key techniques can of course be used to verify authenticity over channels considered to be insecure.

Is there protection against interception / imitation?

No - that's the kind of thing that we want to happen. If it's decided that app B responds to protocol X better than app A does, it should be allowed to deinstall app A and install app B.

Where are queued messages stored and how secure is it?

File-scoped queued messages are stored in a database on the same filesystem as the file they are scoped to. The database is only readable to the super-user, and the ToolTalk database server (running as root) will only give 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 managing the session. ttsession will only give the messages to a process that has satisfied the authentication mode that the ttsession is running in.

Is ToolTalk C2 qualified?

No.

How do I get a trace of what's happening to my message?

Turn on the trace output of the ttsession involved. To do this, perform kill -USR1 <unix_-pid_of_the_ttsession_process>.

NOTE: Note rpc.ttdbserverd also has a trace option available on start-up.

DEBUGGING

How can I isolate my debugging from all the other tools using ToolTalk?

The normal Sun deskset tools, and the SPARCworks debugger actually use ToolTalk for communication. This can make for voluminous trace output, and much confusion since so much is going on in one session. Also, it's really not too desirable to have the debugger using the same ttsession as the programs you're attempting to debug. Now's the time to use the "process tree session" mode that we document. This is the mode that puts the session name in a environment variable instead of using the X DISPLAY environment variable to find the ttsession process. The way to use it here:

Step 1:

% ttsession -t -c $SHELL * * ttsession (version 1.0, library 1.0) * ttsession: starting %

This starts a new session, with trace mode on (-t). The -c option makes this a process tree session; ttsession starts, sets the environment variable, and forks the given command ($SHELL) (defaults to csh)

Now, you're running in a subshell. All the commands run from this subshell will use the ttsession started from the command line, not the usual “X session” ttsession (usually auto-started by a DeskSet tool.) You can check what the session id is for this new ttsession by seeing the value of the _SUN_TT_SESSION environment variable.

Running this way isolates the test from the “normal” ttsession. This is a big advantage here as all the patterns and messages flowing for “normal” session operations won't show up.

Step 2: inside the subshell, run the test programs:

```
% ./my_receiver & [1] 4532 % ./my_sender &
```

.. and look at the output of the ttsession trace.

Step 3: Remember to exit the subshell after testing — if you start any ToolTalk-using tool in the subshell, it will use the process tree ttsession, not the X-session ttsession, and things can get pretty confusing.

C++

Can I use ToolTalk with C++?

Certainly. In fact, ToolTalk is written in C++ and most internal test drivers are in C++ too. Of course the API is still C. The main reason is that C++ (at least, all current implementations of C++) requires that all clients be recompiled whenever any part of the class changes — even the “private” parts. This would make it impossible to maintain binary compatibility between releases.

FILENAMES

Should I qualify my filenames?

We don't allow explicit hostname qualification of pathnames. If you give us a filename with a ':' in it, we will look for filename with a ':' in it.

Whenever you do a tt_message_file() or a tt_default_file(), what we give you is the realpath of that file as it appears on the machine on which you invoked tt_message_file() or tt_default_file(). (See the realpath(3) man page for details on what a real path is.)

What it comes down to is that ToolTalk goes to VERY elaborate lengths to make sure that

1. If two clients file-scope to the same file on different machines, then they can talk to each other, no matter how differently those two files are actually mounted on those two machines; and

2. We always give you a locally-valid, canonical pathname back to you, no matter what tortuous, varied pathnames you give us.

In most of the media messages, there are the contents parameter and an optional docName parameter. How are they related to the file attribute of a standard ToolTalk message? If docName and file attribute do not match, what does it imply?

It implies that the requester is of the opinion that the user-visible name of the document is different from the user-visible name of the file in which the document is stored.

For instance, mailtool keeps track of the user-visible name of attachments. If mailtool wants to transmit an attachment's contents via the filesystem, this allows it not to have to give the temp file the same name as the attachment.

VERSIONS

What versions have what?

First, all the ToolTalk programs support a -v switch that will tell you their version.

ToolTalk 1.0, 1.0.1, and 1.0.2 all have virtually the same function.

All the current ports are essentially 1.0.2

ToolTalk 1.1:added contexts; was made MT safe; added ptype_undeclare() and tt_message_send_on_exit(); removed dependence on classing engine

ToolTalk 1.1.1 added XDR support, point-to-point and static pattern call- backs, and a way for an application to dynamically load PTYPEs.

DOCUMENTATION

Is there a ToolTalk news group?

Yes, alt.soft-sys.tooltalk

What documentation is available?

PRODUCT AND TECH SPECs

- *TOOLTALK: APPLICATION INTEGRATION SOLUTIONS* (Tech Spec; 2 pages; 10/92)
  Provides a high level overview of the ToolTalk features and benefits that are bundled with Solaris 1.0 and 2.X.
  Printed Tech Spec also available: Part # 92265-001.
  (SunWIN token #21666)(SOS token #10633)

- *MULTIPLATFORM TOOLTALK* (Product Spec; 2 pages; 10/92)
  Explains to customers how to get ToolTalk binaries, SDK, licensing, and support for DEC, HP, and IBM RISC platforms.
  Printed Product Spec also available: Part # 92252-001.
  (SunWIN token #21668)(SOS token #10634)

OVERVIEW WHITE PAPERS

- *INTRODUCTION TO THE TOOLTALK SERVICE* (18 pages; 10/92)
  Overviews ToolTalk application integration, benefits, usage, standard message sets, and migration to DOE.
  (Printed version also available: Part # 91031-002)
  (SunWIN token #21658)(SOS token #10635)

- *TOOLTALK SERVICE: A TECHNICAL REPORT* (39 pages, 10/92)
  Detailed technical overview including ToolTalk capabilities, architecture, API, and standard message sets.
  (Printed version also available: Part # 91022-002)
  (SunWIN token #21776)(SOS token #10649)

- *DESIGNING AND WRITING A TOOLTALK PROCEDURAL PROTOCOL*
  (23 pages; 6/92)
  Straightforward discussion of when ToolTalk messaging is appropriate and how easy it is to design and write an open procedural protocol using ToolTalk.
  (Copy Center Part # 801-3592-01)
  (SunWIN token #21704)(SOS token #10636)

- *TOOL INTEROPERABILITY: A HANDS-ON DEMONSTATION* (23 pages; 8/92)
  A simple demonstration of how the ToolTalk service works including code examples of how to add ToolTalk messages to a real application.
  (Copy Center Part # 801-3593-01)
  (SunWIN token #21724)(SOS token #10637)

- *TOOLTALK IN ELECTRONIC DESIGN AUTOMATION* (11 pages; 9/92)
  Summary of ToolTalk's usefulness in the EDA market; Overview of ToolTalk messaging in the CAD Framework Initiative's Integration Project.
  (SunWIN token #21670)
  (SOS token #10638)

STANDARD MESSAGE SETS

- *TOOLTALK DESKTOP SERVICES MESSAGE SET: Release 1.0* (42 pages; 8/92)
  Provides complete information on the basic message set required to achieve desktop integration and to enable inter-application control.
  (SunWIN token #21710)
  (SOS token #10639)

- *TOOLTALK DOCUMENTATION AND MEDIA EXCHANGE MESSAGE SET: Release 1.0* (24 pages, 7/92)
  Facilitates the transfer of multimedia and conventional data and documents between applications for any type of data — from ASCII to full-motion video.
  (SunWIN token #21685)
  (SOS token #10640)

  The latest release for these last 2 is 1.1.1, but that version probably isn't yet available through SunWIN or SOS[?].

- *CASE INTEROPERABILITY MESSAGE SETS: Release 1.0* (180 pages; 10/92)
  SunSoft collaborated with Silicon Graphics, Digital Equipment, and the CASE Interoperability Alliance (Cadre, Centerline, Frame, IDE, Interleaf, Softool, and SunPro to develop this message document. It provides user scenarios for message integration and 145 pages of detailed man pages related to the standard message sets. (Also, see the CASE World press release below.)
  (SunWIN token #21653)
  (SOS token #10641)

PRESS RELEASES AND FACT SHEETS

- *"SUNSOFT'S TOOLTALK AVAILABLE FOR INDUSTRY'S LEADING COMPUTER PLATFORMS: Application Interoperability Demonstrated Across Multiple Systems"* (2 pages; Interop, 10/28/92)
  Announces that ToolTalk is available for 75% of world's UNIX workstations including Cray, DEC, HP, IBM, Intergraph, SGI, SMCC, and SPARC- and x86-compatibles.
  (SunWIN token #21662)
  (SOS token #10642)

- *"DIGITAL, SILICON GRAPHICS AND SUNSOFT LEAD INDUSTRY TOWARD OPEN CASE TOOL INTEGRATION STANDARD"* (4 pages; CASE World 10/1/92)
  SunSoft, DEC, and SGI announce CASE messaging proposal and submit it to the ANSI X3H6 CASE Tool Integration Models Committee for standardization. 12 leading tool suppliers endorse the joint proposal. (See above under "Standard Message Sets" for the "CASE Interoperability Message Sets" document.)
  (SunWIN token #21620)
  (SOS token #10643)

- *"MEDIA EXCHANGE: Enables Applications to Share Multimedia Functionality* (press fact sheet; 2 pages; Siggraph 7/92)
  SunSoft and SMCC announce the open Media Exchange messaging specification for ToolTalk, which allows applications to share multimedia information. (See above under "Standard Message Sets" for the "ToolTalk Documentation And Media Exchange Message Set" document itself.)
  (SunWIN token #21577)
  (SOS token #10644)

- *"MOMENTUM CONTINUES FOR SUNSOFT'S TOOLTALK: Leading Developers Make ToolTalk Foundation for Next-Generation Apps"* (3 pages; CASE World; 2/19/92)
  Announces ToolTalk endorsements by Frame, Interleaf, Applix, Island Graphics, Harris Scientific, Visix, Softool, CaseWare, QualTrak, SunPro, Digital Tools, TeamOne, Procase, and Silvaco. Also announces formation of CASE Interoperability Alliance and CASEWorld demonstration of 9 applications integrated through ToolTalk messaging.
  (SunWIN token #21563)
  (SOS token #10645)

- *"SILICON GRAPHICS TO SHIP SUNSOFT'S TOOLTALK: Object-Oriented Software Allows Apps to Communicate Across Multiple Platforms"* (3 pages; CASE World; 2/19/92)
  Silicon Graphics endorses ToolTalk and states intention to ship ToolTalk on IRIS 4D systems by the end of 1992.
  (SunWIN token #21565)
  (SOS token #10646)

BOOKS

- *TOOLTALK PROGRAMMER'S GUIDE* (Part #801-2875-10) and *TOOLTALK SETUP AND ADMINISTRATION GUIDE* (Part #801-2876-10)

  These two documents from the product are available in several forms:

  1. Part of the OpenWindows Developer documentation set for SunOS 4.1.x/ Solaris 1.0 and Solaris 2.0. Part of Solaris 2.1 documentation set.

  2. Both manuals will be combined into one volume as: *THE TOOLTALK SERVICE: AN INTER-OPERABILITY SOLUTION*. Published by SunSoft Press/Prentice Hall; available December 1992; 354-page paperback; $30.00 list price; ISBN 013-088717-x. Will be available in retail bookstores worldwide and through SunExpress. Internal discounts for SMI departments and employees available via SunSoft Press.

  This book describes the ToolTalk service and how you modify your applications to send and receive ToolTalk messages. This book is a technical presentation of ToolTalk programming, setup, and administration that is not tied to platform or OS specifics (although the presentation assumes UNIX is the base environment).

- *TOOLTALK AND OPEN PROTOCOLS: INTER-APPLICATION COMMUNICATION*
  by Astrid Julienne and Brian Holtz (~350 pages)
  Published by SunSoft Press/Prentice Hall; available April 1993; $36.00 list price; ISBN 013-031055-7.
  Will be available in retail bookstores worldwide and through SunExpress. Internal discounts for SMI departments and employees available via SunSoft Press.

  This book will discuss how to design, write, and implement good open protocols with examples using ToolTalk as the messaging service. The book's five parts will include:

  Part 1: How Applications Talk with Each Other
  Part 2: Using ToolTalk Functionality to Design Your Protocol
  Part 3: Creating ToolTalk Procedural Message Protocols
  Part 4: Creating ToolTalk Object Message Protocols
  Part 5: Examples of Good ToolTalk Protocols