Designing and Writing a ToolTalk Procedural Protocol

Astrid Julienne and Brian Holtz

Introducing the ToolTalk Service

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

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

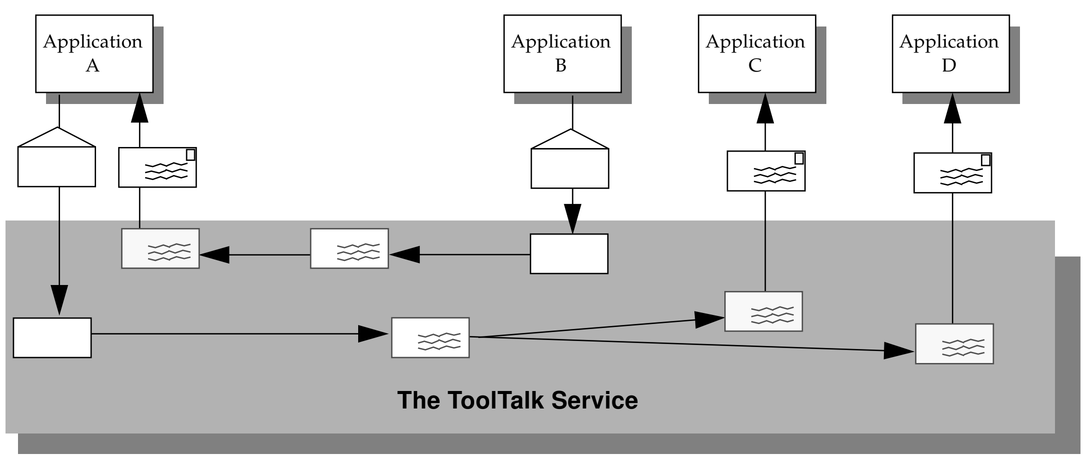

Figure 1 Applications Using the ToolTalk Service

Developers of inter-operating applications agree upon a set of messages and then use the ToolTalk service to deliver the messages. The set of messages exchanged by a group of inter-operating applications is called a *protocol*; a protocol forms a small well-defined interface that maximizes application autonomy.

The message protocol specification includes the set of messages and how applications should behave when they receive the messages. If protocols are observed, cooperating applications can be modified, even replaced, without affecting one another. An application written or modified today to observe a protocol is inherently compatible with future applications that observe the same protocol.

The ToolTalk service supports several messaging styles. A sender can address a ToolTalk message to a particular process, to any interested process, to an object, or to an object type. Message senders are not concerned with the locations of processes and objects in a local area network; the ToolTalk service finds receiving processes and objects.

Addressing Processes

To address messages to processes so that the process performs specific tasks, you would write a *procedural* protocol. This paper outlines a straight-forward method of how to create a good open procedural protocol.

Addressing Objects

To address messages to objects managed by applications, you would write an *object* protocol. Object-oriented messages are particularly useful for applications that currently use objects or that are to be designed around objects. If an existing application is not object-oriented, the ToolTalk service allows applications to identify portions of application data as objects so that applications can begin to communicate about these objects.

Object protocols are not covered in this paper. Detailed information on how to write a good object protocol is included in our forthcoming Prentice-Hall/SunPress book entitled “Open Protocols: Inter-Application Communication Now.” (The book will be available in bookstores, through SunSoft CDware™, and directly from Prentice-Hall.)

Deciding When the ToolTalk Service is the Appropriate Technology to Use

While we would like the ToolTalk service to be the answer to all your inter-operability needs, there are some problems for which the ToolTalk service may not be the appropriate technology to use. This section takes a look at some of the problems the ToolTalk service is designed to solve, and some of the problems that it is not designed to solve. When your application needs to solve both sorts of problems, you can use the ToolTalk service in combination with other technologies.

What Kinds of Problems Does the ToolTalk Service Solve?

The ToolTalk service is the appropriate technology to use if your application needs:

- Tool inter-changeability
- Control integration
- Network-transparent events that are not owned by any well-known server (for example, an X server) and that do not have any predictable set of listeners
- Automatic tool invocation
- Persistent objects

Tool Inter-changeability

Use the ToolTalk service when you want *plug-and-play* capability. The term *plug-and-play* means that any tool can be replaced by any other tool that follows the same protocol. That is, any tool that follows a given ToolTalk protocol can be placed (plugged) into your computing environment and perform (play) those functions indicated by the protocol. Tools can be mixed and matched, without modification and without having any specific built-in knowledge of each other.

Control Integration

Use the ToolTalk service when your application requires *control integration*. The term *control integration* indicates a group of tools working together toward a common end without direct user intervention. The ToolTalk service enables control integration through its easy and flexible facilities for issuing arbitrary requests, either to specific tool instances or to anonymous service providers.

Network-Transparent Events

Use the ToolTalk service when your application needs to generate or receive network-transparent events. To be useful, traditional event mechanisms (such as signals and window system events) require special circumstances; for example, you must know a process or window ID. The ToolTalk service allows events to be expressed naturally: in terms of the file to which the event refers, or the group of processes on the network to which the event is applicable. The ToolTalk service delivers events (called *notices*) to any interested process anywhere on the network. ToolTalk notices are a flexible and easy way to provide extensibility for your system.

Automatic Tool Invocation

Use the ToolTalk service when your application needs network-transparent automatic invocation. The ToolTalk service lets you describe the messages that, when sent from any location on the network, should cause your tool to be invoked. The ToolTalk auto-start facility is easier to use and less host-specific than the conventional `inetd(1)` facility.

Persistent Objects

Use the ToolTalk service when your application needs to place objects in the UNIX file system unobtrusively.

Note – ToolTalk objects are beyond the scope of this paper.

What Kinds of Problems May Other Technologies Solve More Appropriately?

Other technologies may be more appropriate to use when your application:

- Demands maximum speed and is willing to sacrifice flexibility
- Transfers massive amounts of data from one process to another
- Requires single-desktop clipboard services
- Requires control integration with direct user manipulation
- Needs a name service

Speed versus Flexibility

When your application requires speed more than it does flexibility, we recommend that you use the remote procedure call (RPC). RPC provides high performance, synchronous, point-to-point invocation of well-known services. For example, the Network File System (NFS) is a good application of RPC; NFS is a speed-critical service with a simple protocol that does not need extensibility or plug-and-play capability.

Transferring Large Amounts of Data

When your application needs to transfer large amounts of data from one process to another, we recommend that you use sockets and shared memory. The ToolTalk service is not the appropriate technology for these types of transfers because the current version copies all message data into the server in order to insulate senders from the flow-control burdens inherent in distributing large amounts of data.

Note – Future versions of the ToolTalk service may allow the ToolTalk server to be bypassed when transferring massive amounts of data.

Single-Desktop Clipboard Services

When your application needs a single-desktop clipboard service, we recommend that you use a window system mechanism, such as *X* selections, to retrieve named values on the desktop.

Control Integration with Direct User Manipulation

When your application requires control integration that involves direct user manipulation, we recommend that you use a workable technology already associated with the domain in which the manipulation occurs (if one exists). For example, we recommend that your application use a window system mechanism (such as the *drag-and-drop* mechanism) to make the initial connection between two window system entities if an operation requires that the user manually associate the two entities.

Name Services

When your application needs to look up resources or retrieve keyed values on the network, we recommend that you use a name service. Name services, such as the Network Information Service (NIS), are optimized for the requirements of scalability, availability, consistency, and security that are essential in this domain.

Developing a Procedural Protocol

Once you have decided that the ToolTalk service is the appropriate communications service to use, you need to determine the way that your tool will work with other tools. There are several basic questions you need to ask:

1. How will these tools work together?
2. What kinds of operations can these tools perform?
3. What kinds of operations can these tools ask other tools to perform?
4. What events will these tools generate which may interest other tools? (What type of messages will these tools want to send?)
5. What events generated by other tools will be of interest to these tools? (What type of messages will these tools want to receive?)

To best answer these questions, we develop a scenario that outlines the order of the events and operations that our tools will expect to perform and be performed. However, before we can develop a scenario, we need to understand the difference between events and operations, and how the ToolTalk service handles messages regarding each of these.

What is the Difference Between an Event and an Operation?

An *event* is an announcement that something has happened. An event is simply a news bulletin. The sending process has no formal expectations as to whether any other process will hear about the event, or whether an action is taken as a consequence of the event. When a process uses the ToolTalk service to inform interested processes that an event has occurred, it sends a *notice*. Since the sending process does not expect a reply, an event cannot fail.

An *operation* is an inquiry or an action. The requesting process makes an inquiry or requests that an operation be performed. The requesting process expects a result to be returned and needs to be informed of the status of the inquiry or action. When a process uses the ToolTalk service to ask another tool to perform an operation, it sends a *request*. The ToolTalk service delivers the request to interested processes and informs the sending process of the status of the request.

Sending Notices

When your application sends a ToolTalk notice, it will not receive a reply or be informed about whether or not any tool pays attention to the notice. It is important to make the notice an impartial report of the event as it happens.

For example, our tool sends a message `Look_At_This` and expects any listening tools to react in a given way, even though our tool does not care, and does not need to be informed, about whether any or no other tool reacts to the message. The message has the semantics of a request (that is, “do this action”) but the delivery characteristics of a notice (that is, it has no distinguished recipient, cannot fail, and does not require a reply). If the message is sent as a request, our tool will expect notification that the message was delivered and the action performed. However, our tool does not care whether the message is received or the action performed; it only wants to report the event:

```
User_Looked_At_This.
```

Therefore, it is clear that we would send this message as a notice.

Sending Requests

When your application sends a ToolTalk request, it expects one tool to perform the indicated operation, or answer the inquiry, and return a reply message. The ToolTalk service guarantees that either a reply will be returned by the receiving process or the sender will be informed of the request's failure.

You can identify the requests that should be part of your procedural protocol in three ways:

1. Identify the operations requested by the tools in your scenario that can fail.
2. Identify the operations each tool in your scenario can perform for other tools.
3. Identify the operations each tool in your scenario will want other tools to perform.

Developing a Scenario

Now we are ready to develop a scenario that outlines the order of the events and operations that our tool will expect to perform and be performed. For example, our tool is a document editor into which the user can place iconic annotations consisting of arbitrary files from other media. We have developed the following scenario:

1. User double-clicks on a document icon in the file manager.
   The file opens in the appropriate editor, which is started by the system if one is not already running.
2. User chooses an "Insert Annotation" command.
   A list of available media types for composing an annotation is displayed.
3. User selects a media type.
   The system starts the application appropriate for the media type.
4. The media application composes the annotation inter-actively.
5. The media application sends the annotation to the document editor.
6. User saves the document.
7. User exits the editor.

We can now answer our basic questions.

How Will These Tools Work Together?

- The file manager will request that the document editor open a document for editing.
- The document editor will request that a media application compose an annotation.
- The media application will send the composed annotation to the document editor.

What Kinds of Operations Can These Tools Perform?

- Each tool has the capability of performing operations such as:
  - iconify and de-iconify
  - raise to front and lower to back
  - change size
  - quit
- Each tool can answer questions about itself and its state such as:
  - What is your version?
  - What is your environment?
  - What is your status?
- The document editor and media tool are both a type of editor. These editors can perform such operations as:
  - compose a document
  - edit a document
  - display a document
  - print a document
  - translate a document among various formats

The ToolTalk service defines a set of standard messages that request a tool instance to perform these operations and answer these queries. We will use a subset of these generic messages in our example protocol.

What Kinds of Operations Can These Tools Ask Other Tools to Perform?

- The file manager must request that the document editor open a document for editing. The request needs to contain enough information so that the system will deliver the correct data to the correct program.

  We will call this request *Edit*.

- The document editor must request that a media tool compose an annotation.

  We will call this request *Compose*.

- The media tool may need to request the document editor's current screen position.

  We will call this request *Geometry_Get*.

What Events Will These Tools Generate Which May Interest Other Tools?

- The document has been saved.
- The document editor has been exited.

These are examples of generic events that any tool can generate which may interest other tools.

The ToolTalk service defines a set of standard messages that notify interested processes that a generic event (such as file operations, window operations, starting, and exiting) has taken place. We will use a subset of these generic messages in our example protocol.

What Events Generated by Other Tools Will Be of Interest to These Tools?

- During composition, the media tool may save the annotation as a checkpoint. The media tool will need to announce this event to the document editor.

  We will call this notice *Buffer_Saved*.

Designing an Open Procedural Protocol

Once you have determined the way your tool will interact with other tools and developed a scenario to determine the order of the messages in your protocol, you are now ready to design your protocol.

We encourage open protocols. A protocol is open largely to the extent that it consists of *anonymous messages* (that is, messages sent without knowledge of who is to receive them). The use of anonymous messages allows the user to easily inter-change applications that observe the same protocol.

We also encourage protocols to be extensible. An extensible protocol is one in which the existing messages can be adapted for new and unforeseen situations, thus allowing existing tools to become more useful as new tools and new messages are added to the system.

Expressing a ToolTalk Protocol

A ToolTalk protocol is expressed in terms of the *roles* that each tool plays (that is, the kinds of task each tool is assigned to perform); and in terms of the specific patterns that each tool registers in order to receive the messages in which it is interested.

Understanding the ToolTalk Type Mechanism

The ToolTalk types mechanism is designed to help the ToolTalk service route anonymous messages to the tools that have assumed a particular role. You define a process type (*ptype*) for each role in your protocol, and then compile the ptypes with the ToolTalk type compiler, `tt_type_comp`. A ptype provides the ToolTalk service with the following information:

- How your tool can be started
- The maximum number of instances of your tool that can run simultaneously
- What requests your tool is interested in handling
- What notices your tool is interested in observing
- What requests and notices can cause your tool to be started

When your tool declares a ptype, the message patterns listed in it are automatically registered; the ToolTalk service then matches messages it receives to these registered patterns. These static message patterns remain in effect until your tool closes communication with the ToolTalk service.

Understanding the Structure of a Ptype

A ptype contains the following information:

- A process type identifier
- A start string
- Maximum number of tool instances allowed per session
- Signatures

Process Type Identifier

A ptype begins with a process type identifier (*ptid*). This identifier must be unique within a given installation.

Note – We recommend that this identifier be unique across all possible installations by prefixing the ptid with, for example, the company or product name.

Start String

A ptype usually has a start string. The ToolTalk service invokes this string (through the shell) when an instance of the ptype needs to be started.

Per Session limit

A ptype usually includes a *per_session* number. This number tells the ToolTalk service the maximum number of instances of your tool that can run in the same session at the same time.

Signatures

Signatures describe the messages that your tool wants to receive when it is performing the role associated with the given ptype. A ptype contains two kinds of signatures: *handle* and *observe*. The handle signatures describe the requests that instances of the ptype can be expected to perform; the observe signatures describe the notices that instances of the ptype are interested in observing.

A signature is divided by an arrow (=>) into two parts. The first part of a signature specifies how the message is to be matched. The second part of a signature specifies what special actions (such as start a ptype instance or queue the message), if any, are to be performed when the signature is matched.

How to Design Your Protocol to be Open

To design your protocol to be open, you need to consider several points as you write it:

1. Make requests anonymous.
2. Let tools be started as needed.
3. Only reply to a request when the requested operation has been completed.
4. Avoid statefulness.
5. Declare one ptype for each role a tool can play.

Make Anonymous Requests

To design your protocol to be open, you want the requests to be anonymous. That is, the requesting process has no knowledge of which tool instance — or even which tool type — will perform the requested operation. If the requests are sent to a specific process, you control how users or potential message recipients can utilize their resources. If the requests are sent to a specific tool type, you restrict the new kinds of tools that can interact with your tool.

You want your message to describe the operation being requested or the event being reported. You do not want your message to describe the process that should receive the message. The less specific knowledge each tool encodes about the tools with which it will interact, the more flexible the overall system is for the user.

Let Tools Be Started as Needed

To design your protocol to be open, you want the system to start tools only as needed. When you let a new tool instance be started only as needed, you provide the user with more flexibility and more efficient use of resources such as cpu, screen real estate, and swap space. The ToolTalk service has several features that assume the responsibility of determining when to start a new tool instance:

- The ToolTalk service allows messages and type signatures to have "start" reliability; that is, if no eligible recipient of a start-reliability message is running (or willing to accept the request), the ToolTalk service will start an instance of the type of tool which is statically registered to handle or observe that message.
- The ToolTalk service allows each ptype to specify the maximum number of its instances that may be started in a given session.
- The ToolTalk service offers each request to all eligible running handlers before it starts a new tool instance. An eligible handler can accept or reject a request based on its own criteria (such as its ability to take on a new task; whether or not it has unsaved changes; idle time; iconic state; or whether or not the user has indicated that the tool is free to accept new work).

Reply When Operation has been Completed

To design your protocol to be open, you want to notify the sending process that its requested operation has been performed. However, the operation invoked by a request sometimes takes a relatively long time to complete compared to the time it takes to send the message. Since the sending process is expecting a reply, your tool can respond in two ways:

1. It can reply immediately that it has received the request and then convey the actual results of the completed operation in a later message.
2. It can withhold the reply until the operation has been completed.

We recommend the second policy because ToolTalk messaging is entirely asynchronous: neither a tool (nor the session it is in) is blocked because it has one or more requests outstanding.

However, if an operation requires a long time to complete, you might want a *control channel* for status notices or cancellation requests. When a control channel requirement is included in the request, the handler will send a series of status notices point-to-point back to the sending process if it knows the operation will be lengthy. These notices identify the handler to the sending process.

Note – The current version of the ToolTalk service does not deliver control channel messages to observing applications. Observers of the request will receive only the actual reply when the operation has been performed.

Avoid Statefulness

To design your protocol to be open, you want each message to make sense by itself whenever possible. When a protocol is stateless, the messages in it avoid dependency on any previous messages or on some state in the assumed recipient. However, we recognize that it is not always practical to avoid statefulness.

Declare One Ptype per Operation

A ToolTalk ptype essentially instructs the ToolTalk service how to handle any messages in which a tool is interested that are sent when that tool is not running. To design your protocol to be open, you want to declare one ptype for each role in your protocol. When you declare only one ptype per role in your protocol, you provide users with the flexibility to inter-change tools as their needs require. For example, a user may want a sophisticated sound-authoring tool for recording but also prefer a simple audio tool to perform the playback.

Thus, you will sometimes want to include only one message signature per ptype. When you include more than one message signature in the same ptype, you are requiring that any program that can handle one message can handle the other messages. For example, a ptype “UWriteIt” can include the three message signatures “Compose,” “Display,“ and “Edit” because it is expected that any tool that understands the UWriteIt document format can perform all three of these operations.

How to Design Your Protocol to be Extensible

To design your protocol to be extensible, you need to consider several points as you write it:

1. Use (and expect) optional arguments.
2. Report events accurately.
3. Use generic messages to cause well-known behaviors.
4. Use parameterized messages.

Use Optional Arguments

To design your protocol to be extensible, you will want to use — and expect — optional arguments. That is, specify only the minimum number of arguments required in order for the message to make sense. If a particular argument is not explicitly required, omit it from the message pattern and document it as an optional argument that the tool will use if the argument is included in the message. When a message contains optional arguments, the receiving tool can process these arguments according to their value type (vtype) name and ignore any optional argument whose vtype name is unfamiliar in the context of that message.

Note – Always place optional arguments *after* the mandatory arguments.

Reporting Events Accurately

To design your protocol to be extensible, you will want to make sure that your notices accurately and objectively report events. For example, a given event may have two different interpretations, such as `File_Saved` and `File_CheckedIn`. If a tool only reports the `File_Saved` interpretation because it expects observers not to differentiate between the two operations, the tool is reporting the event neither accurately nor objectively.

Using Generic Messages

To design your protocol to be extensible, use *generic messages* when you want a tool instance to behave in a well-known or specific manner; for example, if you want to cause a tool instance to move, iconify, or quit.

Generic messages are not anonymous messages — for example, you know exactly which process you want to iconify — and the specified process does not need to register interest in the iconify message. Generic messages are sent point-to-point and are delivered regardless of whether the recipient has registered for them in its types file.

Using Parameterized Messages

To design your protocol to be extensible, use *parameterized messages*. A parameterized message is a message with a single operation name that is still routed to different types of recipient tools based on the arguments in the message. Parameterized messages allow you to use message variants that you cannot anticipate when you construct your application.

For example, you could set conventions that allow an editing tool to compose a message appropriate to any media format that it encounters. The editing tool could issue requests about generic formats – such as "video" – and have the request handled by whatever video tool the user has installed. The request could return a "newMediaType" string argument. The handling video tool would return the data in the new format named by the "newMediaType" argument if it is unable to return the data in the generic format. The editing tool would then associate the new format name with the returned data and construct future messages appropriately.

Practical Considerations for Designing Your Protocol

There are several final points you need to consider as you write your protocol:

1. Note whether your tool is being started by the ToolTalk service.

2. How you use out-of-band data transfers.

3. Use file-scoping when available.

4. Know when to reject messages.

Starting Your Tools

When the ToolTalk service starts a tool, it marks the message that started the tool. However, tools typically do not retrieve ToolTalk messages until they have entered their event loop; that is, *after* they have made such decisions as screen location and locale.

If your tool supports a `–tooltalk` option, it can use that option in the start string of the receiving tool's ptype. When the receiving tool is started, it can immediately retrieve the message that started it and then send requests point-to-point directly to the tool that caused it to start to obtain any information it needs.

Using Out-of-Band Data Transfers

The preferred way to transfer data associated with a ToolTalk message is to put the data in the message itself as an argument. For example, a display request can have a required "contents" argument that contains the data to be displayed.

However, if the data is too voluminous to efficiently transfer as a message argument, use an *out-of-band* avenue. An out-of-band avenue in your message indicates that the data is stored elsewhere. For example, the "contents" argument of the display request could have no value and an optional argument could indicate the out-of-band avenue. Alternatively, you could scope the message to the file that contains the data.

Using File-Scoping

ToolTalk file-scoping allows messages and patterns to reference files in a convenient and network-transparent way. However, the ToolTalk database server must be installed on the relevant file servers; you need to be aware that it may not be installed if file-scoping is not an essential part of an installed product's protocol.

When you reference a file, for example, as an out-of-band data transfer method, use file-scoping as your preferred referencing method; if you anticipate an installation problem for the file's file server, also name the file in an optional argument.

Avoid using file scoping as an artificial rendezvous mechanism. If a set of tools on the network need to talk to each other and only have a string in common, do not treat that string as an NFS pathname to which to scope all their messages; instead, treat the string as a name service key by which a ToolTalk session identifier is retrieved. To create a dedicated ToolTalk session that is not connected to any X display, use the `-p` or `-c` option when you run ttsession. Other tools can join this session by either:

- calling `tt_default_session_set()`
- setting the `_SUN_TT_SESSION` environment variable before calling `tt_open()`

Reject Messages

A running tool usually accepts a request if the new request will not interfere with any ongoing operation. In general, a tool usually accepts or rejects requests based on whether:

- It is busy with another request.
- It has unsaved edits.
- The user has indicated it must maintain its state (from its previous operation).

For example, our hypothetical document editor can open a new window for each compose request it receives; therefore, it can accept a second request before the first request is satisfied. By contrast, our hypothetical media tool can only service one compose request at a time; therefore, it must reject subsequent compose requests until the current request is serviced.

Writing an Open Procedural Protocol

Now we are ready to actually begin to write our protocol. First, we will translate the information from our scenario into ToolTalk ptypes. There are three process types that we want to declare:

- the sound playback role: Acme_AudioPlayer
- the sound authoring role: Acme_AudioComposer
- the document editing role: CasaDePublishing_UWriteIt

In addition, we want:

- the ToolTalk service to start each of these processes if no process is running
- a maximum of 5 audio tool and 1 uwrite tool instances to be running at the same time
- to receive status notices of the compose operation

The following ptype sample is an example of how we translated the information from our scenario into our protocol.

Annotation	Code
Declares ptype	`ptype Acme_AudioPlayer {`
Starts tool as needed	`start "audiotool -tooltalk";`
Maximum tool instances	`per_session 5;`
Receiving process	`handle:`
	`/*`
Optional arguments	`* Optional extra arguments for these requests:`
	`*                     in    requestIDrequestID`
	`*/`
Message signature	`session Display(      in    audiocontents) => start;`
	`}`

An Open Procedural Protocol

The following protocol is based on the scenario we developed in this paper.

```
ptype Acme_AudioPlayer {
    start "audiotool -tooltalk";
    per_session 5;
    handle:
    /*
     * Optional extra arguments for these requests:
     *                    in    requestIDrequestID
     */
    session Display(      in    audiocontents) => start;
}
ptype Acme_AudioComposer {
    start "audiotool -tooltalk";
    per_session 5;
    handle:
    /*
     * Optional extra arguments for these requests:
     *                    in    requestIDrequestID
     */
    session Compose(      out   audiocontents,
                          out   stringnewMediaType) => start;
    observe:
    /*
     * These are notices which this tool *sends* (as opposed to
     * receives), so we document them here but hide them from the
```

```
     * types compiler.
     *
     * Optional extra arguments for this notice:
     *            in    <mediatype>contents
     *
    session Buffer_Saved( in    stringbufferID);
     */
};
ptype CasaDePublishing_UWriteIt {
    start "uwrite -tooltalk";
    per_session 1;
    handle:
    /*
     * Optional extra arguments for these requests:
     *                    in    requestIDrequestID
     */
    session Display(      in    udoccontents) => start;
    session Edit(         inout udoccontents,
                          out   stringnewMediaType) => start;
    session Compose(      out   udoccontents,
                          out   stringnewMediaType) => start;
    /*
     * Optional extra arguments for this request:
     *                    in    <mediatype>contents
     *
```

```
    session Buffer_Saved( in    stringbufferID);
     */
};
```

Summary

As the demand for independently-developed applications that work together increases, the inter-operability of your application is an important factor *today* and an open protocol is the key to successful inter-operation. The ToolTalk service is designed to facilitate your development of applications that meet the demands of the future. When you apply the general principles described in this paper to the writing of your protocol, your application will have the flexibility required for users to easily inter-change tools, as illustrated by our simple ToolTalk example.

We are very excited about this technology of the future. Our forthcoming book "Open Protocols: Inter-Application Communication Now" provides guidelines for writing more sophisticated protocols, including object protocols, and also include topics such as how to implement your protocol; how to use inheritance to simplify the administration of protocols; and when and how to use multiple procids. It also contains design and practical tips, and sample message protocols. The book, published by Prentice-Hall/SunPress, will be available in bookstores, through SunSoft CDWare, and directly from Prentice-Hall.

Designing and Writing a ToolTalk Procedural Protocol

A White Paper

SunSoft
A Sun Microsystems Company

Table of Contents

SunSoft, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

For more information, call 1 800 227-9227.

Printed in USA 6/92 92199-001 1.5K