OZONE Distributed Communication Library

Ora Lassila & Marcel Becker

CMU-RI-TR-96-11

The Robotics Institute *
Carnegie Mellon University
Pittsburgh, PA 15213

March 1996

* This research has been supported in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Material Command, USAF, under grant number F30602-95-1-0018 (as part of the ARPA / Rome Labs Planning Initiative), and the CMU Robotics Institute. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency and Rome Laboratory or the U.S. Government

Abstract

This report describes the distributed architecture and distributed programming primitives of the OZONE framework, a toolkit for building planning and scheduling applications. It also serves as a programmers' reference manual for those who want to build distributed system components either in the context of OZONE-based scheduling systems or in the context of some entirely new applications.

The class library documented in this report has two parts: The first part, implemented in CLOS, provides classes for such concepts as *server*, *client* and *command*. The second part is a function library written in TCL, allowing clients – including user-interface clients – to be built using TCL.

Acknowledgements

The development of the communication library described in this report was supported in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Material Command, USAF, under grant number F30602-95-1-0018 (as part of the ARPA/Rome Labs Planning Initiative), and the CMU Robotics Institute. The design and implementation of the system are based on an earlier OPIPE library as well as the distributed primitives included in the DITOPS scheduler (of the first phase of the ARPA/Rome Labs Planning Initiative), both written by Ora Lassila. Large parts of the new implementation, including all TCL code were written by Marcel Becker.

We would like to thank Ben Werle for his invaluable help with bug fixes and improvements to the system, as well as Stephen F. Smith for managerial and moral support during this project.

Pittsburgh, March 1996

Ora Lassila & Marcel Becker

Contents

Chapter 1

Introduction

This report will provide an overview of the communication architecture of the OZONE[1] Planning and Scheduling Toolkit [5]. The communication architecture (and its implementation as a class and function library) is suitable for building distributed applications and agent-based systems. The principal focus of the report is the use of the communication architecture to provide the separation of the user interface from the rest of the scheduling system (as advocated in [6]), thus describing the implementation of a TCL/TK-based [4] user interface for the various scheduling applications produced using the OZONE toolkit, including the DITOPS Transportation Scheduler and the DITOPS Aeromedical Evacuation Planner (see, for example, [3]).

For a quick start on how to use the communication library – how to adapt it to a new application environment – the reader is referred to Chapter 2.

1.1 Conceptual vs. Physical Architecture

The core components of the *conceptual* architecture are the *user interface* – through which the user communicates with the system – and the *executive* – which delegates requests for services from various components of the system. Other components of the system could include various kinds of service providers like a *scheduler*, a *model manager* and a *database* (see Figure 1.1).

[1] Please note that in order to avoid confusion (or to create some more, some people might say), we have decided to call the Planning and Scheduling Toolkit with the new name "OZONE" (= O_3 = "**O**bject-**O**riented **O**PIS"), and the name "DITOPS" used earlier will refer to the transportation-related applications of OZONE.

The conceptual architecture need not be *physically* implemented as shown, several modules could be combined into larger physical components (or processes). Indeed, the old (CLIM-based [8]) DITOPS architecture [2] has all of these component functionalities in a single physical "package." It must also be noted that the conceptual architecture reflects the *communication* organization of the system. This is not to be confused with lines of command and/or authority in the system (that is, the executive is *not* the so-called "Top-Level Manager" of the DITOPS scheduler). More specifically, the executive is akin to a telephone switchboard, it is responsible for finding the recipient of communication based on an "address" or very simple clues from the content of the communication (for example, certain commands could always be directed to the *scheduler*, etc.).

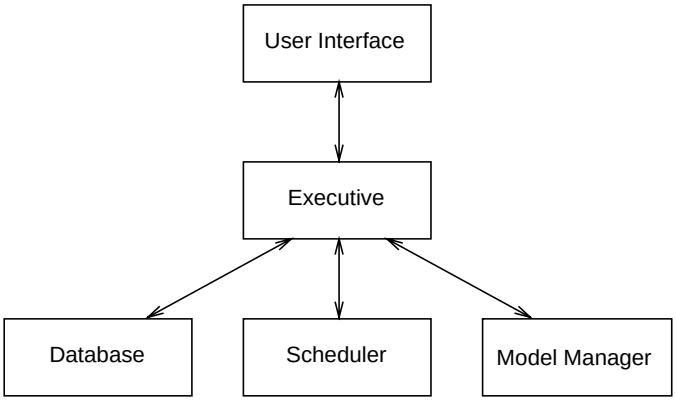

Figure 1.1: Conceptual view of the system architecture

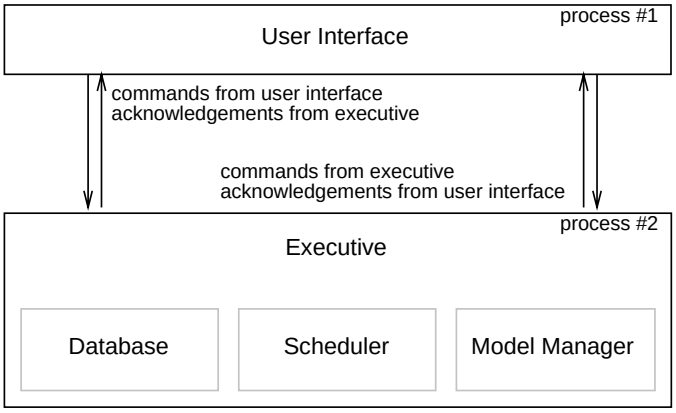

Figure 1.2: Physical view of the system architecture

The physical architecture – the actual implementation – can be realized in several ways. For example, every component of the system can be a separate process

(even reside on separate physical CPUs). The approach we have chosen for the first prototypes is to include the *scheduler*, the *model manager* and the *database* in the same process with the *executive*. This way, the executive directly manages the other components. The user interface is implemented as a separate process (see figure 1.2) of the underlying operating system.

Communication between the user interface and the executive requires some additional consideration. One should note that this is not a traditional client-server configuration, since both components need to be able to initiate actions and request services from the other component. From the user interface's standpoint, the executive / scheduler is a server, but from the executive / scheduler's standpoint the user interface is a server. Thus two independent lines of communication are needed. Both the user interface process and the executive process have to be multi-threaded (or at least event-driven) processes.

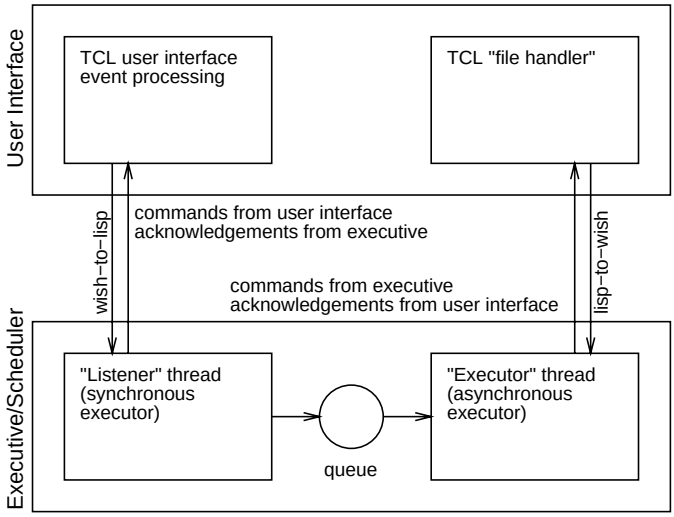

Figure 1.3: Internal Processes

The first prototype implementations (using BSD-style sockets) have the internal architecture described in Figure 1.3. The user interface is implemented using TCL / TK and runs on a `wish` interpreter. The server has the traditional CLOS implementation of OZONE. Command processing and communication is done in the following way:

1. On the TCL (`wish`) side, normal event processing causes commands to be issued and consequently sent over the "wish-to-lisp" socket. Acknowledgements are received over the same socket.

2. On the Lisp side, the “listener” thread receives the command and decides whether it is a synchronous command (in which case it is executed by that thread and results sent back with the acknowledgement) or an asynchronous one (in which case only a “command received” acknowledgement is sent back).

3. The asynchronous commands are placed in a (semaphore-guarded) queue.

4. The “executor” thread runs in a loop, always picking the next command from the queue and executing it. The execution of asynchronous commands may cause notifications to be sent to the user interface on the “lisp-to-wish” socket, and acknowledgements are received from TCL on the same socket.

5. The processing of Lisp-initiated communication (such as notifications sent by asynchronous commands) is handled by TCL’s “file handler” function.

There is no distinction between synchronous and asynchronous commands on the TCL side, since the user interface is assumed to only handle fast actions of relatively small granularity. Also, we are not using a multithreaded implementation of TCL.

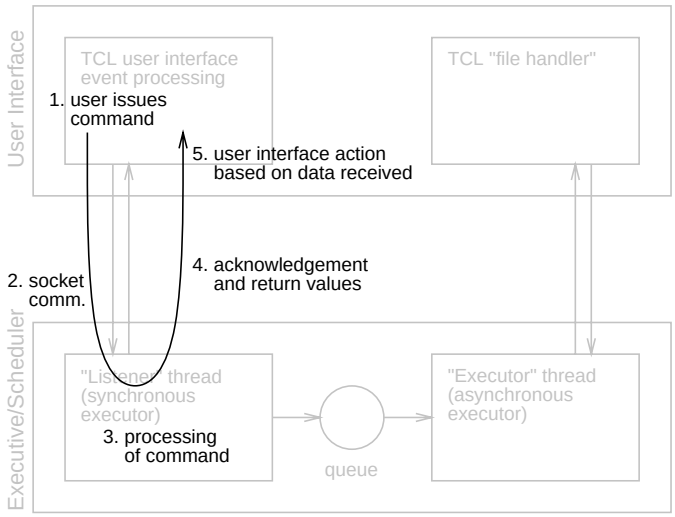

Figure 1.4: Example of synchronous command processing

Figure 1.4 illustrates the processing of a synchronous command. Similarly, Figure 1.5 illustrates the processing of an asynchronous command. Asynchronous

commands are those that may take a long time to process and are therefore executed without the user interface having to wait. The user interface receives an acknowledgement, however, as soon as the command is *received*, thus making synchronous and asynchronous command handling (from the user interface's standpoint) uniform.

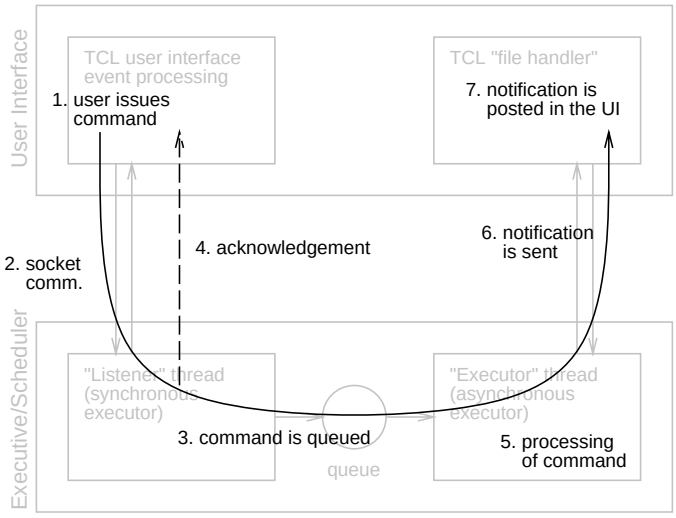

Figure 1.5: Example of asynchronous command processing

1.2 Communication Mechanisms

Several options are available for implementing the communication mechanism. Figure 1.6 illustrates the layered construction of this software. The "gray area", the *communication subsystem* can be implemented in a number of different ways:

- **Sockets:** Both "lines of communication" are bi-directional (BSD-style) sockets. As mentioned in the previous section, this option has been implemented in our first prototypes.
- **Apple Events:** This is the preferred mechanism on the Macintosh, since it allows us to make both the user interface and the executive/scheduler "scriptable" (and recordable for that matter), allowing other programs to use these modules for implementing services.

- **CORBA:** It would also be possible to use some type of object request broker (ORB) mechanism to implement the communication between clients and servers.

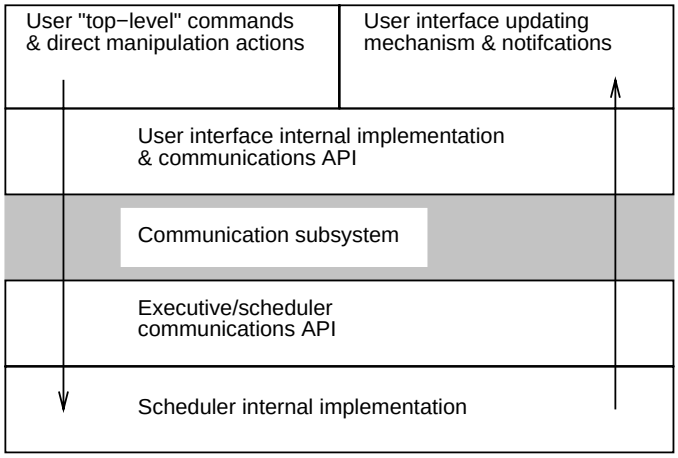

Figure 1.6: Communication architecture

Figure 1.6 shows the communication architecture from a standpoint of separating a user interface from an application. The interface-side “communications API” has been documented in Chapter 4 and the server-side API in Chapter 3. It should be noted that the library can be used for other types of communication as well. For example, two “servers” could communicate with each other using a symmetric configuration. This type of approach makes it possible to use the library to build agent-based systems.

Chapter 2

Using the Communication Library

There are two *required* steps to using the communication library: (1) all necessary software components have to be *installed*, and (2) the software needs to be *configured and customized* for use in a new application environment.

2.1 Installing Software

The current implementation of the library uses `dpwish`, a distributed version of the TCL interpreter. This software is available (at the time of writing) from

```
ftp://ftp.aud.alcatel.com/tcl/extensions/
```

as `tcl-dp3.3b1.tar.gz`.

The Common Lisp global variable `*wish-command*` must contain the correct pathname of the `dpwish` program to enable proper startup of the system. For example, your configuration file should contain a command like this:

```
(setf *wish-command* "/usr/local/bin/dpwish")
```

The system source code resides in a set of Common Lisp and TCL source files. The TCL source files are loaded into the interpreter per instructions from the Common Lisp process. By default, the lisp process expands the logical pathname

```
opis:ui;kernel-tcl;
```

to get the physical pathname of the directory where all TCL source files reside. Regardless of what other logical hosts and pathnames your system uses, you should establish the logical host "`opis`" and a corresponding translation for "`ui;kernel-tcl;`" (or alternatively you can always edit the source file).

2.2 Configuring Software

The linkage from a user interface – or some other "client" program – to an application is implemented as follows:

1. User's **actions** are translated into **API calls** in the user interface.
2. **API calls** cause **textual commands** to be communicated through the communication channels (in the current implementation).
3. **Textual commands** are parsed and translated into **command objects** using a command table.
4. **Command objects** are executed, resulting in **function calls** in the application's implementation.

In order for the library to be used to create the user interface for a new application, the application has to provide *commands* which are inserted into a *command table*, implementing the linkages from named commands to functions which provide command functionality. The global command table provided by the library is contained in the variable `*commands*`. It can be used with calls to `insert-command` to add commands. Here's an example of how to add a command called "SAMPLE" to correspond to the function `sample-command`:

```
(insert-command *commands* #'sample-command
                :name 'sample
                :asynchronousp t
                :status "Executing a sample command")
```

From the TCL side, this command can be called as follows:

```
opis_synch_command "SAMPLE;"
```

If one wanted to pass parameters to `sample-command`, the call would look like this:

```
opis_synch_command "SAMPLE foo 2;"
```

In this case the parameters would be the symbol `foo` (read in in the package specified to the synchronous executor) and the integer `2`.

To start the user interface, the server must call the function `init-interface`. To shut it down, the function `close-interface` must be called. In order to start the client from `dpwish`, one first needs to set the variable `tclSourceDir` to contain the pathname of the TCL source directory, then "`source`" the file "tcl_to_lisp.tcl", and then call the function `start_interface`.

Chapter 3

CL/CLOS Programming Interface

This chapter documents the interface functions available in the Common Lisp and CLOS client/server implementation of the toolkit. Same documentation style has been used as in the book "Common Lisp: the Language" [7].

Please note that unlike the rest of the OZONE Toolkit, the communication toolkit does not use PORK [1] as its object system, but is built using "plain vanilla" CLOS. This allows the communication library to be used even if one does not want to install the object infrastructure required by the OZONE framework.

3.1 Servers

Servers are entities capable of responding to outside requests. The toolkit has a base server class (called `server`) which is specialized to provide the different kinds of server entities required by the communication and command execution mechanism. *Executors* are specialized servers which read and execute commands.

3.1.1 Server Protocol

This section outlines all generic functions which server and executor classes have to define methods for.

`server-create-name` [Generic function]

server

This function is called when a server is created to generate a name for a server process (it is used for debugging purposes).

`server-stream` [Generic function]
server

This function accesses the stream assigned to the server, to be used for communication between the server and the client.

`server-process` [Generic function]
server

This function accesses the server process, i.e. the lightweight control thread on which the server is executing.

`server-run` [Generic function]
server

This function is the "main loop" of the server. It is called when the server is created and started.

`executor-queue` [Generic function]
executor

This function accesses the command queue associated with the executor. Queues are described later in this manual.

`terminatep` [Generic function]
executor

This function is called by an executor to determine if termination is requested. If this function returns true the execution of an executor ends.

`executor-read-next-command` [Generic function]
executor

This function reads and returns the next command to be executed by the executor. Typically this command is read either from an external I/O stream or the message queue of the executor.

`executor-process-command` [Generic function]
executor
command
args

This function is called to process (to execute) a command.

`executor-find-queue` [Generic function]

executor

other-executor

This function is used by the various executor classes to find the right message queue object during startup. It is called with the executor as the *executor* parameter and any other existing executor (or `nil`) as the *other-executor* parameter. See the documentation of actual methods to understand how this function works.

3.1.2 Server Classes

`server` [Class]

`:stream` [Initarg]

Subclasses of this class include `executor`. This is the base class for all servers. The initarg *stream* can be used to initialize the server's stream object.

`server-stream` [Method]

(*self* `server`)

This method accesses the slot `stream`. This slot holds the stream object that this server uses for communication.

`server-process` [Method]

(*self* `server`)

`(setf server-process)` [Method]

value

(*self* `server`)

These methods access the slot `process`. This slot holds the lisp process (lightweight thread) object on which this server is executing.

`server-create-name` [Method]

(*server* `server`)

This method implements the specified functionality of its generic function.

`server-run` [Method]

(*self* `server`)

The base server class has no associated run semantics (a warning is generated). Subclasses need to override this method.

`executor` [Class]

`:queue` [Initarg]

This class inherits directly from `server`. Subclasses of this class include `asynchronous-executor` and `synchronous-executor`. *Executors* are specialized servers which read and execute commands. This is the base class for various executor classes.

`executor-queue` [Method]
   (*self* `executor`)
`(setf executor-queue)` [Method]
   value
   (*self* `executor`)

These methods access the slot `queue`. This slot holds the message queue associated with the executor, if any.

`terminatep` [Method]
   (*self* `executor`)
`(setf terminatep)` [Method]
   value
   (*self* `executor`)

These methods access the slot `terminatep`. This slot is initially `nil`. Setting it to `t` terminates the execution of the executor.

`server-run` [Method]
   (*self* `executor`)

This function runs in a loop calling first `executor-read-next-command` and then `executor-process-command` until the executor is requested to terminate.

`executor-find-queue` [Method]
   (*self* `executor`)
   (*other* `null`)

This method gets called when the first executor is created (no other executors exist, so *other* is `nil`). It creates a message queue and assigns it to this executor.

`executor-find-queue` [Method]
   (*self* `executor`)
   (*other* `executor`)

This method takes the message queue of *other* (an executor) and assigns it to this executor.

`synchronous-executor` [Class]
`:package` [Initarg]

This class inherits directly from `executor`. This class implements the execution of synchronous commands. It reads commands from a socket stream and either executes them directly (synchronous commands) or passes them on to the asynchronous executor. The *package* initarg can be used to specify the package in which commands are read.

`executor-package` [Method]
(*self* `synchronous-executor`)

This method accesses the slot `package`. This slot holds the package (object) in which all new commands are read in. It defaults to the value of `(find-package :opis)`.

`server-create-name` [Method]
(*self* `synchronous-executor`)

This method implements the specified functionality of its generic function.

`executor-read-next-command` [Method]
(*self* `synchronous-executor`)

This method function reads commands from the executor's stream and translates the commands into command objects (using the function `find-command`).

`executor-process-command` [Method]
(*self* `synchronous-executor`)
(*command* `(eql -unknown-command-)`)
args

This method will acknowledge the command with the -1 status code.

`executor-process-command` [Method]
(*self* `synchronous-executor`)
(*command* `synchronous-command`)
args

This method will execute the command using `command-execute` and will send back the return values.

`executor-process-command` [Method]
(*self* `synchronous-executor`)
(*command* `asynchronous-command`)
args

This method will queue the command (using `add-to-queue`) for execution by an asynchronous executor.

`find-command` [Method]
(*self* `synchronous-executor`)
token

This method will call the generic function `find-command` with the global command table `*commands*` as a parameter (see below).

`asynchronous-executor` [Class]

This class inherits directly from `executor`. This class implements the execution of asynchronous commands. It reads its commands from its message queue.

`server-create-name` [Method]
(*self* `asynchronous-executor`)

This method implements the specified functionality of its generic function.

`executor-read-next-command` [Method]
(*self* `asynchronous-executor`)

This method will read the next command from the executor's message queue. If the queue is empty the method will wait for the synchronous executor to place something into the queue before returning (waiting is done using the proper process-wait primitive of the underlying Common Lisp system).

`empty-queue-p` [Method]
(*self* `asynchronous-executor`)

This method is used as the checking predicate for the process-wait primitive in `executor-read-next-command` (for example, in our Allegro CL implementation the process-wait function is `mp:process-wait`).

`executor-process-command` [Method]
(*self* `asynchronous-executor`)
(*command* `asynchronous-command`)
args

This method will execute the command (using `command-execute`) and will handle return values (from `accumulate-value`) by immediately sending them back to the client as asynchronous commands.

`executor-output` [Method]
(*self* `asynchronous-executor`)

string
&rest *args*

This method writes *string* to the executor's stream and waits for acknowledgement.

3.1.3 Server/Client Communication

This section describes the mechanisms for the server to communicate value back to the caller (client).

`accumulate-value` [Function]
value

During the processing of a server command, this function will send one value back to the client. Depending on whether the command processing is done synchronously or asynchronously, different things will happen: during synchronous processing, values are "accumulated" into a list, and at the end of processing these values become the return values of the command function; during asynchronous processing every call to `accumulate-value` sends an asynchronous notification back to the client.

`accumulated-value` [Condition Class]

This condition class is used for communicating accumulated values from the caller of `accumulate-value` to the currently running executor. The treatment of the condition is different depending on the type of executor (synchronous or asynchronous).

`accumulated-value` [Method]
(*self* `accumulated-value`)

This method accesses the slot `value`. This slot is used for communicating the value passed to `accumulate-value`.

3.2 Clients

A base client class is provided for implementing clients in CLOS. This class is not currently used by our implementation.

`client` [Class]
`:stream` [Initarg]
`:init-token` [Initarg]

This class implements client behavior. It is not used in the current implementation.

`client-stream` [Method]
(*self* `client`)
`(setf client-stream)` [Method]
value
(*self* `client`)

These methods access the slot `stream`. This slot holds the stream through which the client communicates with its server.

`client-init-token` [Method]
(*self* `client`)

This method accesses the slot `init-token`. This slot holds the "token" (a symbol or string) which gets written to the stream when a connection is first established (to identify the protocol the client will be using).

`connect-to-server` [Method]
(*self* `client`)
`&key` *host*
port

No documentation available for `connect-to-server`.

`disconnect-from-server` [Method]
(*self* `client`)

No documentation available for `disconnect-from-server`.

3.3 Commands

The command mechanism provides two base classes: one for commands and another for command tables.

3.3.1 Command Protocol

`command-name` [Generic function]
command

This function accesses the name of a command. Names are used when communicating commands over character streams.

`command-function` [Generic function]
command

This function accesses the function object which implements the command. This function is called when the command is executed.

`command-status` [Generic function]
command

This function accesses a string which may be displayed while the command is executing.

`command-execute` [Generic function]
command
args

This function will apply *args*, a list of parameters, to the implementing function of the command.

`find-command` [Generic function]
table
token

This function will map a command name (the *token*) to an actual command object contained in the command table.

`(setf find-command)` [Generic function]
command
table
token

This function will physically insert a command object into the command table.

`insert-command` [Generic function]
table
command
`&key` *name*

asynchronous
class
status

This function will (optionally) create a command object and will insert it into the command table. The parameter *command* is the function implementing the particular command, or a command object.

3.3.2 Command Classes

`command` [Class]
`:name` [Initarg]
`:function` [Initarg]
`:status` [Initarg]

Subclasses of this class include `asynchronous-command` and `synchronous--command`. This is the base class for all commands.

`command-name` [Method]
(*self* `command`)

This method accesses the slot `name`. This slot holds a symbol that names the command. The name is the one that gets transmitted over the communication link, and basically allows the command function to be called something else than the command itself.

`command-function` [Method]
(*self* `command`)

This method accesses the slot `function`. This slot holds the function object which implements the command.

`command-status` [Method]
(*self* `command`)

This method accesses the slot `status`. This slot holds a string which can be displayed while the command is executing (e.g., “loading resources”).

`command-execute` [Method]
(*command* `command`)
args

This method implements the specified functionality of its generic function.

`synchronous-command` [Class]

This class inherits directly from `command`. This is the class for all synchronously executed commands. The execution of a synchronous command by a server blocks the client.

`asynchronous-command` [Class]

This class inherits directly from `command`. This is the class for all asynchronously executed commands. The execution of an asynchronous command by a server does not block the client.

`command-table` [Class]

This class implements a repository for commands, with primitives for parsing.

`find-command` [Method]
(*self* `command-table`)
token

This method implements the specified functionality of its generic function.

`(setf find-command)` [Method]
command
(*self* `command-table`)
token

This method implements the specified functionality of its generic function.

`insert-command` [Method]
(*self* `command-table`)
(*command* `command`)
`&key` *name*
asynchronous
class
status

This method implements the specified functionality of its generic function.

`insert-command` [Method]
(*self* `command-table`)
(*function* `function`)
`&key` *name*
asynchronous
class
status

This method creates a command object as follows: *function* is the implementing function of the command. The parameter *name* must be specified and becomes the naming symbol of the command. If *asynchronous* is false, *class* will default to `synchronous-command`, otherwise *class* defaults to `asynchronous-command` (the command is generated using the class specified by *class*). The parameter *status* is the optional status string associated with the command.

`*commands*` [Variable]

This global variable holds the global command table.

3.4 Queues

`message-queue` [Class]

This class provides an implementation of a FIFO-queue with locking using a semaphore.

`queue-lock` [Method]
(*self* `message-queue`)

This method accesses the slot `lock`. This slot holds the semaphore object (e.g., in Allegro CL a process lock) used for locking the queue during modifications.

`add-to-queue` [Method]
(*self* `message-queue`)
thing

This method will add *thing* to the queue.

`remove-from-queue` [Method]
(*self* `message-queue`)

This method will remove and return the next item from the queue. Two values are actually returned: the value (or `nil` if queue is empty) and a boolean value indicating whether the queue is empty or not (true if queue is empty).

3.5 Client and Server Startup

`start-daemon` [Function]
`&key` *port*

process-name

This function starts the executor daemon, a process which listens to incoming requests to open a socket. Every socket opened is associated with a newly created server thread. This function is called automatically (if needed) by `init-interface`.

`*wish-command*` [Variable]

This variable holds the pathname of the unix command implementing the `wish` interpreter.

`init-interface` [Function]

This function creates the executor daemon (if necessary), launches the `wish` interpreter, loads interface code to `wish` and calls the `start-interface` TCL function.

`close-interface` [Function]

This function will shut down the user interface (TCL) side and will then kill the executor threads.

`kill-d` [Function]

This function will kill the executor daemon thread. It has been provided for debugging purposes.

`kill-p` [Function]

This function will kill the executor threads. It has been provided for debugging purposes.

3.6 Debugging

Certain variables and functions have been included to help with debugging a system built using this library. The main problem with debugging a multi-threaded implementation is that if an error occurs in one of the executor threads, this thread is effectively blocked and cannot execute.

`*server-trace*` [Variable]

If this variable is true (the default), the `server-trace` function will produce output into the `*trace-output*` stream.

`server-trace` [Function]

string
`&rest` *args*

This function works like `format`, all of its output is directed to the stream `*trace-output*` (unless `*server-trace*` is `nil` in which case no output is produced).

`*signal-server-errors*` [Variable]

If this variable is true (the default) the function `server-error` will signal errors, otherwise only a diagnostic message is printed.

`server-error` [Function]

error

This function will signal *error* (an error object) as if the function `error` had been called, unless the variable `*signal-server-errors*` is `nil` in which case only a simple diagnostic message is printed. The error is signaled in the thread executing the lisp listener (not one of the executor threads), so the stack for the error will not be correct.

Chapter 4

TCL Programming Interface

This chapter documents the interface functions available in the TCL client implementation of the toolkit. Same documentation style has been used as in the previous chapter, but all documented entities are TCL functions.

Setting up a TCL client process requires the `dpwish` interpreter (a distributed extension of the `wish` TCL interpreter). When a TCL client is launched (from the server side using the CL function `init-interface`, the TCL function `start-interface` is the first one that gets called. It will set up the TCL function `eval-asynch-commands` to be used for handling asynchronous notifications the server may send to the client (e.g., screen content modifications during processing).

`start_interface` [Function]

host
port

This function sets up all processing on the TCL client side. The parameter *host* is the name of the machine on which the server is running, and *port* is the socket port number through which the communication is established. It is assumed that the server knows these things when this function gets called. The current implementation will allocate a free port number and communicate that to TCL.

`eval_asynch_commands` [Function]

This function is not called by user's code. It is called repreatedly (during idle time) by the TCL interpreter to read and execute asynchronous notifications sent by the server. These notifications are (in the current implementation) in the form of TCL function calls.

`opis_synch_command` [Function]

args

This function is called by user's code to send commands to the server. Both synchronous and asynchronous commands can be sent (it is up to the server to decide the processing mode of the command). The parameter *args* should be a semicolon-terminated string containing the name of the command and some number of arguments. List objects can be expressed using the TCL list syntax (with braces), the server will translate these to an appropriate list structure.

This function expects the server to respond in a certain manner; the first value read back from the server is an error code: -2 indicates that an error occurred during the execution of the command by the server, -1 indicates that the server does not recognize this command, and 0 or a positive number indicates that command processing was successful and this is the number of return values. Each of the return values is assumed (in the current implementation) to be a TCL expression. These expressions are evaluated, and the results are returned as a list by this function.

Please note that asynchronous server commands always return 0 as their status code. This signifies the acknowledgement of the command. Any return values are returned as asynchronous notifications.

`notify_user` [Function]

message

This function can be called as a response to completed commands, for example. It puts up a small window with the string *message* in it, and the user has to explicitly dismiss the window (by clicking an "OK" button) for the function to return.

Bibliography

[1] Ora Lassila, 1995. “PORK Object System Programmers’ Guide”, Report CMU-RI-TR-95-12, Pittsburgh (PA), The Robotics Institute, Carnegie Mellon University.

[2] Ora Lassila and Stephen F. Smith, 1994. “Constructing Flexible Scheduling Systems for Decision Support”, in *Proceedings of the 1994 Finnish Artificial Intelligence Symposium (STeP-94)*, Turku (Finland), Finnish AI Society.

[3] Ora Lassila, Marcel Becker and Stephen F. Smith, 1996. *An Exploratory Prototype for Reactive Management of Aeromedical Evacuation Plans*, Report CMU-RI-TR-96-03, Pittsburgh (PA), The Robotics Institute, Carnegie Mellon University.

[4] John K. Ousterhout, 1994. *Tcl and the Tk Toolkit*, Reading (MA), Addison-Wesley.

[5] Stephen F. Smith and Ora Lassila, 1994. “Configurable Systems for Reactive Production Management”, in *Knowledge-Based Reactive Scheduling*, IFIP Transactions B-15, Amsterdam (The Netherlands), Elsevier Science Publishers.

[6] Stephen F. Smith and Ora Lassila, 1994. “Toward the Development of Mixed-Initiative Scheduling Systems”, in: Mark H. Burstein (ed.), *ARPA/Rome Laboratory Knowledge-Based Planning and Scheduling Initiative Workshop Proceedings*, San Francisco (CA), Morgan Kaufmann.

[7] Guy L. Steele, Jr., 1990. *Common Lisp – the Language* (second edition), Bedford (MA), Digital Press.

[8] –, 1992. *Common Lisp Interface Manager – User’s Guide*, Menlo Park (CA), Lucid.

Index