Document Type **LIGO-T980115-00 -** **E** 12/4/1998

The Manager API's baseline requirements

James Kent Blackburn

Distribution of this document:

LIGO LDAS Group

The Manager API’s
baseline requirements

James Kent Blackburn

California Institute of Technology
LIGO Data Analysis Group

December 4, 1998

I. Introduction

A. General Description:

1. The managerAPI is responsible for centralized administration of all LIGO Data Analysis System (LDAS) distributed computing API components using an interpreted command language.
   a) The interpreted command language to be used is TCL/TK, which provides a command line, scripting and graphical interface.
   b) The first generation(s) of the managerAPI will be developed entirely in TCL/TK. If a performance requirement is not being satisfied by the purely TCL/TK based managerAPI, then C++ code can easily be used to extend the language utilizing the standard TCL/TK C code API library in the form of TCL/TK packages.
2. The managerAPI TCL/TK script will be responsible for configuration and initialization of the LDAS system. This involves testing communications with all API’s registered in the managerAPI.rsc resource file, initializing the state of all API’s and start-up of missing API’s as outlined in the resource file.
3. All default behavior (*e.g.*, *conduct binary inspiral searches at sites*) will be established after initialization from its resource file managerAPI.rsc.
4. The managerAPI will act as the broker for user requests to the LDAS. All request from User API’s will be in the form of high level language commands to the managerAPI which are queued in a FIFO queue.

B. The managerAPI.tcl Script’s Requirements:

1. The managerAPI.tcl script will provide two additional communication sockets beyond the Operator and Emergency sockets provided by the underlying genericAPI, allowing interactions between LDAS API’s and the manager. These sockets are referred to as the Send and the Receive Sockets in the managerAPI. As with the Emergency and Operator Sockets from the genericAPI, the Send and Receive Sockets will each have their own interpreter. The Send Socket will only connect to listening Operator Sockets on the LDAS API’s and the Receive Socket will always listen for connections from the Operator Sockets on the LDAS API’s.
2. The managerAPI will maintain three command queues used to store com-

mands and messages between users and the LDAS API's.

a) The Command Queue is used to hold incoming high level language commands from User API's. Each command in this queue is passed off to the next available assistant manager interpreter for processing. This queue will be capable of holding 25 high level commands. An attempt to communicate with the Operator Socket by a User API when this queue is full results in a *currently unavailable* message being sent via the Emergency Sockets to the User API.

b) The Send Queue is used to send commands from individual assistant managers to the LDAS API's. This queue will be capable of holding 100 API specific commands before being full. If the queue is full the assistant manager requesting to use the queue is told to wait. All commands placed in this queue have an index associating them with the specific high-level command from the Command Queue and the ID of the assistant manager that pushed the command onto the Send Queue.

c) Each Assistant Manager requires a completion of requested command response from the LDAS API before proceeding to the next LDAS API command instruction associated with processing high level commands. The LDAS API's report completion of commands to the Receive Socket on the managerAPI. The completion message includes the index to the high level command from the Command Queue and the ID of the assistant Manager that requested the LDAS API to perform the command. This queue will be capable of holding 100 API command completion messages. If this queue ever becomes full, the Receive Socket will not accept a new message from the API, forcing it to retry the connection when the queue is available.

3. The managerAPI will use Assistant Manager interpreters to supervise the execution of high level commands. There will be a minimum of three assistant managers running in the managerAPI, with the possibility of more being started if needed. Each Assistant Manager shares equally the CPU time with the managerAPI while carrying out commands. Assistant Managers will be comprised of three sub-processing kernels.

a) Command Parser - This sub-process is responsible for parsing the high level command taken from the Command Queue and parsing it into a recognized procedure (TCL script) and the associated parameters that customize the procedures behavior.

b) Command Scheduler - This sub-processor is responsible for integrating the managerAPI's configuration knowledge base with the output of the Command Parser to produce a script of control commands that can successfully be run based on the current LDAS configuration map. This includes the integrations of parameters from the Command Parser into the script.

   c) API Command Sequencer - This sub-process is responsible for carrying out each command statement in the resulting script produced by the Command Scheduler. Each individual command is sent to the Send Queue where the ManagerAPI forward the command to the appropriate API. The API Command Sequencer then polls the Receive Queue waiting to be notified that the involved API has completed has completed the command before repeating the procedure with the next command in the script.

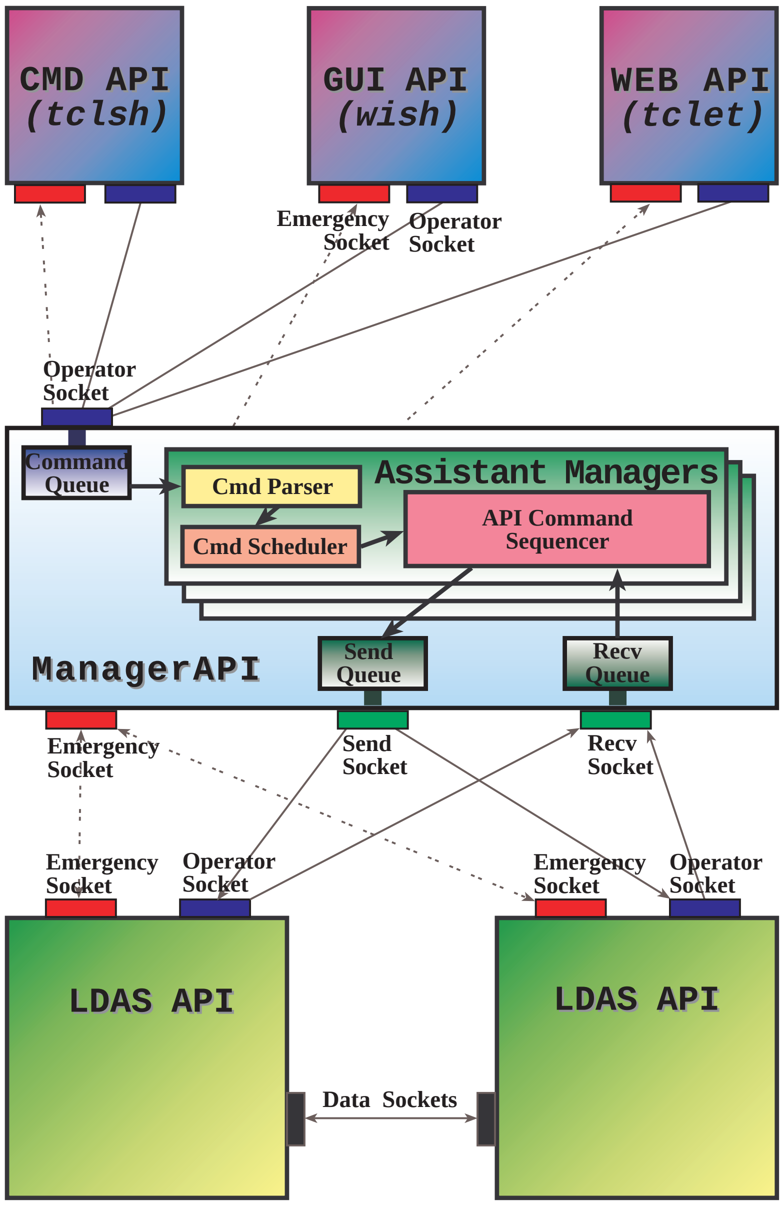
4. The managerAPI will inherit functionality from the genericAPI.tcl script for the purpose of providing logging, help, socket communications for the Operator and Emergency ports, resource management and other features found in the genericAPI.tcl script.
5. The managerAPI will provide a GUI which shows the current status of all sockets, queues, and system resources. This GUI will also provide a command interface which allows an operator at the ManagerAPI to issue commands to the Command Queue. It will also allow for viewing of log files and help documents using the standard set of tools provided by the genericAPI's TCL/TK script.
6. The managerAPI.tcl script will continue to be expanding with newer more detailed functionality as the LDAS API are developed. This means that a complete specification of the functionality will evolve as the command sets of the LDAS API's are implemented.
7. The managerAPI will not buffer data sets as a go between path, in the Frame Format or the LIGO Light Weight Format, as it is transferred between LDAS API's. The LDAS API's will make direct socket connections between themselves as instructed to do so by the managerAPI and report back to the manager upon completion of data transfers.
8. The managerAPI will maintain a list containing the performance statistics for each high level command that appears on the command queue. This list will act as a mini-database containing minimum, average, maximum execution times for each high level command, along with the number of assistant managers interpreters running for these execution times and the number of times each particular command has been executed. The times will be measured using wall clock times starting when an assistant manager first takes a command off the command queue and finishing when the assistant manager completes the command sequence for that high level command. This time will be reported to the manager which uses it to update the list information

C. The managerAPI.so Package Requirements:

1. A managerAPI.so package will not be implemented in the initial version of the managerAPI.
2. If a future requirement for performance in the managerAPI requires a manag-

erAPI.so then it will be developed in C++ using the C language interface to TCL/TK to communicate with the TCL/TK command layer. The wrappers between C++ functionality and TCL/TK command language extensions will be machine generated using the SWIG API code writer.

II. LDAS Command Flow Using the ManagerAPI



A. Command Flow using the ManagerAPI:

1. **Operator Socket**: This socket receives incoming commands from the User-API's and places them on the Command Queue. Prior to placing the com-

mand in the Command Queue, the managerAPI will collect the execution statistics from the Command Statistics List and send these statistics back to the UserAPI, along with the current load as measured by the number of active Assistant Manager Interpreters. The UserAPI will then be given the opportunity to continue or cancel the command based on this summary. If the UserAPI decided to continue the high level command, then the command will be placed on the Command Queue. If a UserAPI has opened a communications socket with the Operator Socket at a time when the Command Queue is full, the Emergency Socket in the ManagerAPI reports to the Emergency Socket on the connected API that the Queue is full, interrupting commands from the UserAPI until such time that the Command Queue is again available for new commands. Commands received at this socket will have return address information attached to the command along with an optional encrypted key needed by the ManagerAPI in order to carry out the requested command. Assuming that the key matches, the command is pushed onto the Command Queue.

Command Queue: This queue receives commands from three sources, the Operator Socket discussed above, the initialization script for the ManagerAPI (*managerAPI.rsc*) and the managerAPI's Command GUI interface. Each command in the queue includes information about the source of the command (*e.g., the UserAPI, or managerAPI's Command GUI, etc.*). This information includes return IP address, port number, etc. needed to identify and report back if necessary to the originator of the command. This queue FIFO (*first in, first out*) queue and is supervised by the ManagerAPI with one exception - all commands from the managerAPI's Command GUI interface are pushed to the top of the queue for immediate execution.

2. **Assistant Manager**: The Assistant Manager processes are responsible for seeing that the high level commands that are in the Command Queue are carried out using the distributed LDAS API components (*servlets*). The ManagerAPI starts up with three such Assistant Managers (interpreters), but can add more if commands are found to reside on the Command Queue for periods longer than 2 seconds. However, the system will never support more than 10 Assistant Managers in order to minimize overhead associated with interpreter context switching. Each Assistant Manager begins by taking the next available command off the Command Queue, starting a timer to measure execution time, and sending it to the Command Parser.

    a) **Command Parser**: This parser separates out all return address information form the command, extracts the parameters which are used to specialize the commands behavior and identifies the ManagerAPI procedure that the command maps into. If the command is found to be unrecognizable by the parser, then an illegal command message is returned to the source of the command via the Emergency Socket (*via a dialog box in the case of commands from issued from the resource file or the GUI*). The accepted command and its parameters are passed from the Command Parser to the Command Scheduler.

b) **Command Scheduler**: This scheduler takes commands (*scripted procedures*), command parameters, and LDAS configuration information (*IP numbers, port numbers, etc.*) and produces a complete, ready for execution script which has had all variable entries assigned from the parameters and configuration data. This script along with an index to the source of the high level command which was used to produce the completed script are then sent to the API Command Sequencer.

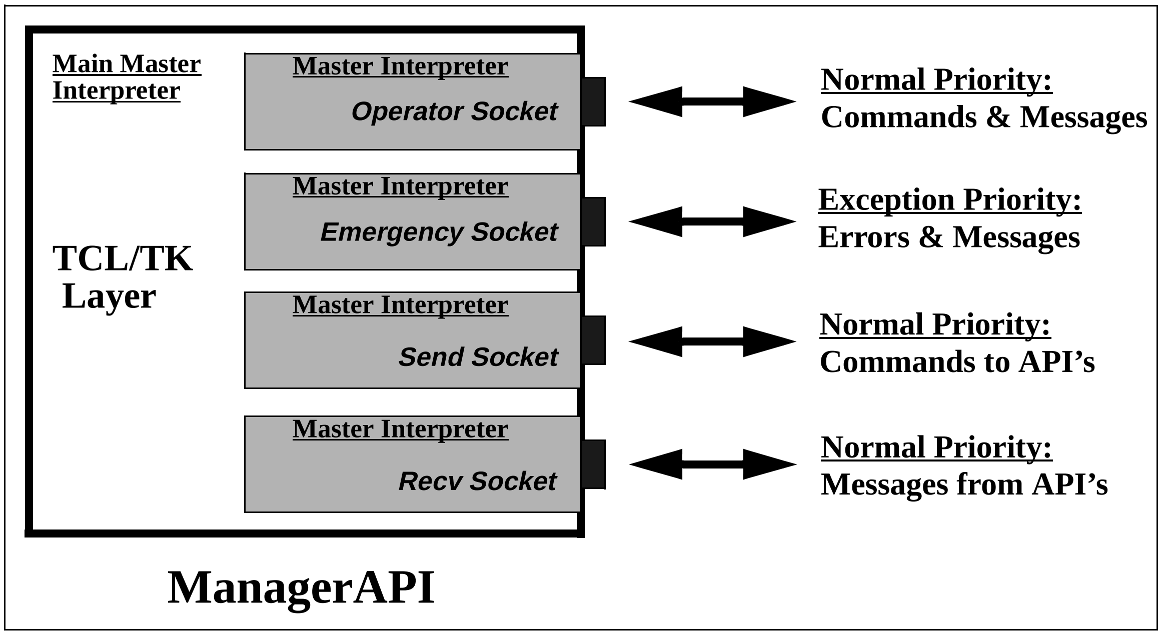
c) **API Command Sequencer**: This sequencer executes each command found in the output script from the scheduler, one at a time, in order to guarantee synchronization of the LDAS system in executing the high level command which triggered the procedure. The sequencer send each command from the scheduler script one at a time to the Send Queue. The next command from the scheduler script will not be sent until the API Command Sequencer finds a command completed message associated with that scheduler command in the Receive Queue. Association is determined via an ID that determines the particular Assistant Manager, the particular high level command and the line number in the scheduler script. This ID information is sent out with the command to the Send Queue and then to the Send Socket to the LDAS API which will perform the command and is then returned with the command completion command to the Receive Socket and the Receive Queue for matching by the corresponding Assistant Manager's API Command Sequencer when reading from the Receive Queue. When the last command in the sequence has been completed, the high level command execution timer is stopped and the wall time needed to complete the high level command is reported to the managerAPI for inclusion in the statistics list.

3. **Send Queue**: This queue takes commands from each Assistant Managers' API Command Sequencer along with the unique ID used to match up command completion messages in the Receive Queue. The ManagerAPI uses this queue as a FIFO to send out LDAS API command using the Send Socket. Each command in the Send Queue is delivered to the Send Socket as soon as the Send Socket is ready to send again.
4. **Send Socket**: Commands (including the scheduler ID information discussed above) along with the optional encryption key are sent out this socket to the appropriate LDAS API for execution.
5. **Receive Socket**: Once an LDAS API has completed the requested command delivered to the API form the managerAPI's Send Socket, it will send back a message stating such to the Receive Socket on the managerAPI with also contains the unique ID associated with the Assistant Manager which placed the requested command in the Send Queue. As these messages are received

at the Receive Socket, they are placed on the Received Queue.

6. **Receive Queue**: This queue holds all messages coming back from LDAS API’s upon completion of their requested commands from the Assistant Managers. All of the Assistant Managers monitor this queue waiting for the unique message reporting that the last sent command has been completed, thereby allowing the Assistant Managers to issue the next command from their API Command Sequencers.
7. **Emergency Sockets**: These sockets are used to report exceptions in the execution of commands by the LDAS API’s. The Assistant Managers will monitor the Emergency Socket Queue looking for messages that one of their issued commands was unsuccessful, causing the scheduled script to abort and an exception message to be sent to the Emergency Socket on the UserAPI which issued the original high level command that was used to generate the command script that failed.

III. Communications Provided by ManagerAPI



A. Socket Based Communications in ManagerAPI:

1. The ManagerAPI will provide an internet socket within the TCL/TK layer that is the primary port for command communications from UserAPI’s. This port, the Operator Socket will be inherited from the genericAPI.tcl script contained within the ManagerAPI. The requirements are a super-set to those defined for the genericAPI, namely that

   a) it run in either a Standard or Safe Master Interpreter depending on the requirements of each specific LDAS API,

b) it have the option of requiring an encrypted key attached to each incoming command for authentication prior to execution of the command as a security safeguard (*depending on the nature of each specific API*),

c) it will provide a queue for incoming commands, allowing for at least 25 commands to be staged in a FIFO during peak levels of communications. If the FIFO ever fills up the UserAPI will be notified using the *Emergency Socket* discussed below.

d) each command is placed in the Command Queue along with communications connection information with the UserAPI issuing the command.

e) The Command Socket and the Command Queue are supervised by the ManagerAPI itself. The Assistant Managers will request the next available command from the Command Queue.

2. The ManagerAPI will provide an internet socket within the TCL/TK layer that is an exception communication port for commands and messages of a highest priority. This port is commonly referred to as the *Emergency Socket* to reflect its association with critical operations. The Emergency Socket will be inherited from the genericAPI.tcl script contained within the ManagerAPI. The requirements are a super-set to those defined for the genericAPI, namely that

   a) it runs in a Safe Master Interpreter supporting only a few exception handing commands for each specific API through the TCL/TK aliasing mechanism,

   b) it have the option of requiring an encrypted key attached to each incoming command for authentication prior to execution of the command as a security safeguard (*depending on the nature of each specific API*),

   c) it will provide a queue for incoming commands, allowing for no more than 10 commands to be staged in a FIFO during peak levels of communications. If the FIFO ever fills up the UserAPI's will be notified.

3. The ManagerAPI will provide a pair of sockets for communicating with the LDAS API's needed to carry out high level command procedures. These are known as the Send Socket and the Receive Socket pair. These sockets are buffered through two queues, the Send Queue and the Receive Queue, each having an initial capacity for 100 commands.

   a) The Send Socket reads each command off the Send Queue as it appears and sends the LDAS API command to the API address/port specified by the Assistant Manager posting the command to the Send Queue. The Assistant Managers only post new commands to the Send Queue, the ManagerAPI is responsible for taking commands off the Send Queue and pushing them out the Send Socket.

b) The Receive Socket takes tagged (index to the high level command and Assistant Manager which originally sent the command) messages from the LDAS API's and places them on the Receive Queue. The Manager-API is responsible for supervising this Receive Socket and placing the incoming messages in the Receive Queue.

4. The ManagerAPI does not use the Data Sockets provided by the GenericAPI.

IV. Software Development Tools

A. TCL/TK:

1. TCL is a string based command language. The language has only a few fundamental constructs and relatively little syntax making it easy to learn. TCL is designed to be the glue that assembles software building blocks into applications. It is an interpreted language, but provides run-time tokenization of commands to achieve near to compiled performance in some cases. TK is an TCL integrated (as of release 8.x) tool-kit for building graphical user interfaces. Using the TCL command interface to TK, it is quick and easy to build powerful user interfaces which are portable between Unix, Windows and Macintosh computers. As of release 8.x of TCL/TK, the language has native support for binary data.

B. C and C++:

1. The C and C++ languages are ANSI standard compiled languages. C has been in use since 1972 and has become one of the most popular and powerful compiled languages in use today. C++ is an object oriented super-set of C which only just became an ANSI/ISO standard in November of 1997. It provided facilities for greater code reuse, software reliability and maintainability than is possible in traditional procedural languages like C and FORTRAN. LIGO's data analysis software development will be dominated by C++ source code.

C. SWIG:

1. SWIG is a utility to automate the process of building wrappers to C and C++ declarations found in C and C++ source files or a special *interface file* for API's to such languages as TCL, PERL, PYTHON and GUIDE. LDAS will use the TCL interface wrappers to the TCL extension API's.

D. Make:

1. Make is a standard Unix utility for customizing the build process for executables, objects, shared objects, libraries, etc. in an efficient manor which detects the files that have changed and only rebuilds components that depend on the changed files.If/when LDAS software becomes architecturally dependent, it will be necessary to supplement make with auto-configuration scripts.

E. CVS:

1. CVS is the Concurrent Version System. It is based on the public domain (and is public domain itself) software version management utility RSC. CVS is based on the concept of a software source code repository from which multiple software developers can check in and out components of a software from any point in the development history.

F. Documentation:

1. DOC++ is a documentation system for C/C++ and Java. It generates LaTeX or HTML documents, providing for sophisticated on-line browsing. The documents are extracted directly from the source code files. Documents are hierarchical and structured with formatting and references.
2. TclDOC is a documentation system for TCL/TK. It generates structured HTML documents directly from the source code, providing for a similar on-line browsing system to the LDAS help files. Documents include a hyper-text linked table of contents and a hierarchical structured format.