LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
- LIGO -
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Document Type	**LIGO-T990086-02 -**	**E**	02/09/2000

The MPI API's baseline requirements

James Kent Blackburn

Distribution of this document:

LIGO LDAS Group

This is an internal working document
of the LIGO Project.

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (818) 395-2129
Fax (818) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS 20B-145
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

The MPI (Message Passing Interface) API's *baseline requirements*

James Kent Blackburn

California Institute of Technology
LIGO Data Analysis Group
February 09, 2000

I. Introduction

A. General Description:

1. The mpiAPI is responsible for managing the advanced analysis processes which are based on MPI and executing in the LDAS distributed computing parallel cluster of nodes using an interpreted command language.
   a) The interpreted command language to be used is TCL/TK, which provides a command line, scripting and a graphical interface.
   b) The TCL/TK commands are extended to support low level system interfaces to the algorithms used to "communicate" the data, as well as provide greater computational performance using C++ code that utilizes the standard TCL C code API library in the form of a TCL/TK package.
2. The mpiAPI's TCL/TK script accesses the mpiAPI.rsc file containing necessary information and configuration resources to extend the command set of the TCL/TK language using the mpiAPI package, which exists as a shared object.
3. The mpiAPI will receive its commands from the managerAPI, reporting back to the managerAPI upon completion of each command. This command completion message will include the incoming identification used by the manager to track completion of sequenced commands being handled by the assistant manager levels of the managerAPI.
4. Because of the way MPI parallel jobs execute, the mpiAPI will only launch MPI jobs on the LDAS distributed computing parallel cluster of nodes and manage the job allocations based on queue configurations which can be dynamically adjusted by algorithms and / or LDAS operators.
5. The mpiAPI will also monitor the state of all MPI parallel jobs and report the status to the controlMonitorAPI using the "*Internal Lightweight Data Format*" (ILWD).

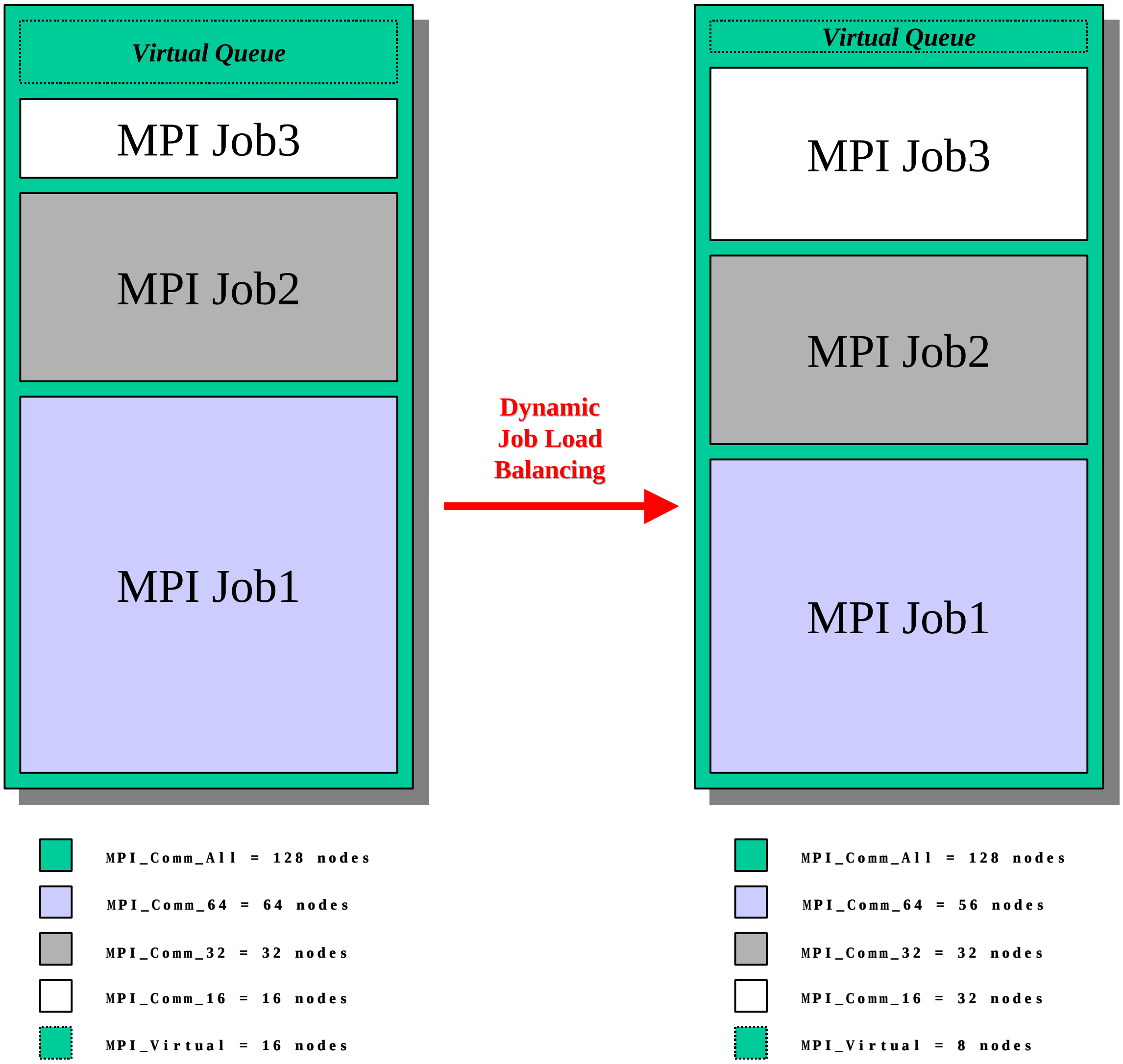
B. The mpiAPI.tcl Script's Requirements:

1. The mpiAPI.tcl script will provide all the functionality inherited by the genericAPI.tcl script (*i.e. help, logging, operator, jobstate & emergency sockets, etc.*).
2. The mpiAPI.tcl script will report to the managerAPI's receive socket upon completion of each command issued by the managerAPI's assistant manager

levels. This involves transmission of a message identifying the specific command completed as coded by the managerAPI (*see LIGO-T980115-0x-E for details*).

3. The mpiAPI.tcl script will validate each command received on the operator, jobstate or emergency socket as appropriate for the mpiAPI to evaluate. This includes validation of commands, command options, encryption keys and managerAPI identification indices.
4. In the event that an exception occurs while processing a command, the mpiAPI.tcl layer will report the exception to the ManagerAPI's *receive socket* along with the necessary command identification issued by the managerAPI with the specific mpiAPI command.
   Note: Once reported to the managerAPI, the appropriate *assistant manager* will terminate the high level command and the userAPI that issued this high level command will be notified of the exception.
5. The mpiAPI.tcl script will be responsible for spawning MPI parallel processes using the unix "mpirun" command and all of its appropriate "options".
6. Spawned jobs will have three levels of priority associated with there processing, ***high***, ***normal***, ***low***. These priorities will be used to borrow nodes in the event that a job can not be balanced with the available number of nodes. Jobs run in the ***high*** priority can dynamically borrow nodes first from ***low*** priority jobs (and if needed from ***normal*** priority jobs). Jobs of ***normal*** priority can borrow nodes only from ***low*** priority jobs. Borrowing only occurs when the virtual queue (*see 7 & 8 below*) is empty.
7. The mpiAPI.tcl script will manage lists of node names and job queues which are used to identify the various allocations of compute nodes that can be used by each MPI parallel process. **Note:** This is intended to allow various job sizes and guarentee uniqueness in the queues assignments for MPI jobs.
8. The mpiAPI.tcl script will keep track of the node allocation needs of MPI jobs. As MPI jobs report a partial release of nodes needed to carry out an active parallel task most efficiently, the mpiAPI.tcl script will manage virtual job queues. **For example**, a MPI parallel process is started using a job queue that has 64 nodes allocated to it. The MPI parallel process "learns" that it can accomplish the task in the necessary time with minimal ideal cpu cycles after a few iterations of its main loop using 56 nodes. The MPI process continues to run in the 56 node queue but reports to the mpiAPI that 8 of the nodes in the queue are no longer needed. This increases the size of the virtual queue by 8 nodes which could be used by a different MPI parallel job to assist in bringing its process time into balance with the real time arrival of data from the interferometer, even after all of the "real" job queues may have been started up already. In the event that a MPI process was started in a job queue that was inadequate for its demands on the system, and all nodes were already allocated to existing job queues, the MPI process could request extra

nodes as they become available in the virtual queue. This in effect allows for dynamic load balancing. (*See figure below for example*). The communication used to communicate node allocation changes to the mpiAPI will be via the jobstate socket of the mpiAPI. Each running MPI process will send commands directly to the jobstate socket containing within the body of the command, the jobid, the nodes currently being used the nodes being released and the nodes being requested. The mpiAPI will respond over the socket with a verification signal, and in the event of a request for new nodes, with a new queue for the job to grow into.



Start-up Job Load **Balanced Job Load**

C. The mpiAPI.so Package Requirements:

1. The mpiAPI.so package will not have any functionality not already provided

in the genericAPI.so package, i.e., it will only need to send and receive "Internal LDAS Lightweight Data Formats" on command from the TCL/TK layer. That is to say the mpiAPI.so package is simply the genericAPI.so package.

D. The mpirun command Requirements:

1. The basic format of the mpirun command as it will be used by ldas is the following:

   mpirun {*mpirun options*} wrapperAPI {*wrapperAPI options*}

   where mpirun is a command script distributed with MPICH and wrapperAPI is the name of the MPI executable developed by LDAS for parallel computation of template based algorithms.

2. The mpirun options requirements are:

   a) **-np N** which is used to specify the number of processors **N** to used in the parallel computation. The value of **N** is always an integer less than or equal to the total number of processors in the LDAS Beowulf Cluster and is set by mpiAPI and its queue management facilities.

   b) **-machinefile /path/file** which is used to identify the list of machine host names used to select our the first N processors required in the previous **-np N** option. The full path and filename for this option must be specified. It is also possible that different instances of the mpirun could use different machinefiles (at the discression of the mpiAPI). The format of the machinefile is simple:

   ```
   hostname1[:n]
   hostname2[:n]
   ...
   ```

   where the hostname most be of the form return by the unix "hostname" command. This hostname may be followed by an optional ":" and an integer number representing the number of CPUs on that particular host-for SMP nodes.

   c) **-nolocal** is an option which specifies to mpirun that the local host is not to be used in the configuration of the parallel processing job. This option may be necessary when the mpiAPI starts a parallel processing job from a host that is not in the core of the Beowulf Cluster (as will be the case in general).

   d) other mpirun options used to test and debug MPI processing will likely be used during commissioning of the mpiAPI and the wrapperAPI. However, they will not be used in general. Thier use must not conflict with the operation of the wrapperAPI and its own set of command line arguments. For more detail on these testing and debugging mpirun command line arguments see the MPICH Users' Guide and Installation Guide.

3. The wrapperAPI options requirements are:

   a) **-nodelist=[i-j,k,l,m-n,...]** which is used to specify the subset of nodes to be used by the MPI slave processes in actual calculation of the templated filters. This list of nodes contains comma delimited node numbers and/or ranges of nodes. All node numbers appearing in this list must be from 0 to N-1, where N is the number of nodes in the commworld specified in the mpirun option **-np** described above.

   b) **-dynlib=/path/libname.a** is used to specify the full path and file name of the dynamically loaded library containing the templated filter algorithms. Note: This library is must be a shared object library.

   c) **-mpiAPIport=[hostname, socketport]** is used to specify the port on the mpiAPI to connect with in order to communicate state information, job progress, and make requests to balance the load by increasing or decreasing the number of processes associated with the nodelist. The **hostname** parameter specifies the name of the host the mpiAPI is running on and the **socketport** parameter specifies the port the mpiAPI is listening at for the purpose of communications with the wrapperAPI.

   d) **-dataAPI=[hostname, socketport]** is used to specify the LDAS API used to provide (serve) data in the ILWD format to the wrapperAPI. Typically this will be the dataConditioningAPI, but others are possible through this argument. Again, the **hostname** specifies the name of the host at with the LDAS API to serve data is running on and the **socketport** parameter specifies the port the data serving LDAS API will be listening at for the purpose of transmitting ILWD formated data.

   e) **-resultAPI=[hostname, socketport]** is used to specify the LDAS API which will receive data products that result from the parallel computation. Again, this data will be shared using the ILWD format. Typically the **resultAPI** will be the eventManagerAPI, however other LDAS APIs may be specified to receive the data products using this argument. The **hostname** parameter specifies the name of the host the receiving API is running on and the **socketport** parameter specifies the port the receiving API is listening at for the purpose of receiving data products from the wrapperAPI.

   f) **-templatesAtNode=N** is the number of templates to be evaluated at each node (in each slave process) per call to the filter algorithm. **N** must be an integer lager than or equal to 1. The wrapperAPI will not allow this number to exceed the total number of templates divided by the number of processors in the comm world. Smaller values of this number allow for more accurate measurements of progress and shorter time intervals for command exchanges between the wrapperAPI and the mpiAPI. Larger values can marginally increase the parallel computation performance by reducing the number of messages passed between master and slave processes.

g) **-filterparams=[a,b,c,d,...]** is used to specify the list of parameters used to control (customize) the parallel filter algorithm. When the designated dynamically loaded library is recognized by the mpiAPI, the values in this list will be validated as being consistant with the expected type, range, and total number for that particular filter library. This will always be the case for LDAS developed dynamically loaded filter libraries. Other which wish to use this mechanism must provide the parameter checks internal to the dynamically loaded library.

4. The master process of the wrapperAPI will be responsible for communicating all state information, warnings, errors, job progress, and make requests to balance the load by increasing or decreasing the number of processes associated with the nodelist. This informations will be communicated using simple textural strings sent to the mpiAPI's listening socket designated by the **-mpiAPIport** command line option using just a simple unix socket connection. Supported command syntax which the wrapperAPI sends to the mpiAPI is as follows:

   a) **"#:request add N"** where # is the request ID (*an incremental counter starting at 1*) and **N** is the number of nodes the wrapperAPI would like to add to the process space associated with the current comm world. The mpiAPI will respond to this request with one of the following three forms of syntax (*NOTE - a request to add may be answered with an order to subtract nodes or even to kill the parallel job*):

      (1) **"#:add N [i-j,k,l,m-n,...]"** where # is the original request ID and **N** may or may not agree with the requested number of nodes and is zero or larger, but can not exceed the comm world. The list in square brackets consists of the actual **N** nodes involved in the **add**.

      (2) **"#:sub N [i-j,k,l,m-n,...]"** where # is the original request ID and **N** may or may not agree with the requested number of nodes and is zero or larger, but can not exceed the comm world. The list in square brackets consists of the actual **N** nodes involved in the **sub**.

      (3) **"#:kill"** where # is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

      (4) **"#:cont"** where # is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

   b) **"#:request sub N"** where # is the request ID (*an incremental counter starting at 1*) and **N** is the number of nodes the wrapperAPI would like to subtract from the process space associated with the current comm world.

The mpiAPI will respond to this request with one of the following three forms of syntax (*NOTE - a request to subtract may be answered with an order to add nodes or even to kill the parallel job*):

(1) **"#:sub N [i-j,k,l,m-n,...]"** where **#** is the original request ID and **N** may or may not agree with the requested number of nodes and is zero or larger, but can not exceed the comm world. The list in square brackets consists of the actual **N** nodes involved in the **sub**.

(2) **"#:add N [i-j,k,l,m-n,...]"** where **#** is the original request ID and **N** may or may not agree with the requested number of nodes and is zero or larger, but can not exceed the comm world. The list in square brackets consists of the actual **N** nodes involved in the **add**.

(3) **"#:kill"** where **#** is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

(4) **"#:cont"** where **#** is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

c) **"#:warning {list of warning messages}"** where **#** is the request ID (*an incremental counter starting at 1*) and **warning** reports that a warning level exception has occured at some level of the wrapperAPI which is described by the messages contained in the **list**. Typically warnings will be used to indicate that a non-fatal condition exists in the wrapperAPI's execution. The mpiAPI log this warning message using the standard LDAS log file system and then will respond to this request with one of the following forms of syntax:

(1) **"#:cont"** where **#** is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

(2) **"#:kill"** where **#** is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

d) **"#:error {list of error messages}"** where **#** is the request ID (*an incremental counter starting at 1*) and **error** reports that a error level exception has occured at some level of the wrapperAPI which is described by the messages contained in the **list**. Typically error will be used to indicate that a fatal condition exists in the wrapperAPI's execution. The mpiAPI log this error message using the standard LDAS log file system and then will respond to this request with one of the following forms of syntax:

(1) **"#:kill"** where **#** is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

(2) **"#:cont"** where **#** is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

e) **"#:progress nnn.mm%"** where **#** is the request ID (*an incremental counter starting at 1*) and **nnn.mm%** is the percent complete for the wrapperAPI's parallel process job. The mpiAPI log this error message using the standard LDAS log file system and then will respond to this request with one of the following forms of syntax:

(1) **"#:cont"** where # is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

(2) **"#:kill"** where # is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

f) **"#:using N [i-j,k,l,m-n,...] nodes out of the M available in comm world"** where # is the request ID (*an incremental counter starting at 1*) and N is the number of nodes being actively used (more specifically the **N** found in the list **[i-j,k,l,m-n,...]**) from the **M** available in the comm world. The mpiAPI log this warning message using the standard LDAS log file system and then will respond to this request with one of the following forms of syntax:

(1) **"#:cont"** where # is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

(2) **"#:kill"** where # is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

g) **"#:projected ratio n.mmmmm"** where # is the request ID (*an incremental counter starting at 1*) and **n.mmmmm** is the ratio of the projected time to completion to the amount of data being analyzed. The mpiAPI log this error message using the standard LDAS log file system and then will respond to this request with one of the following forms of syntax:

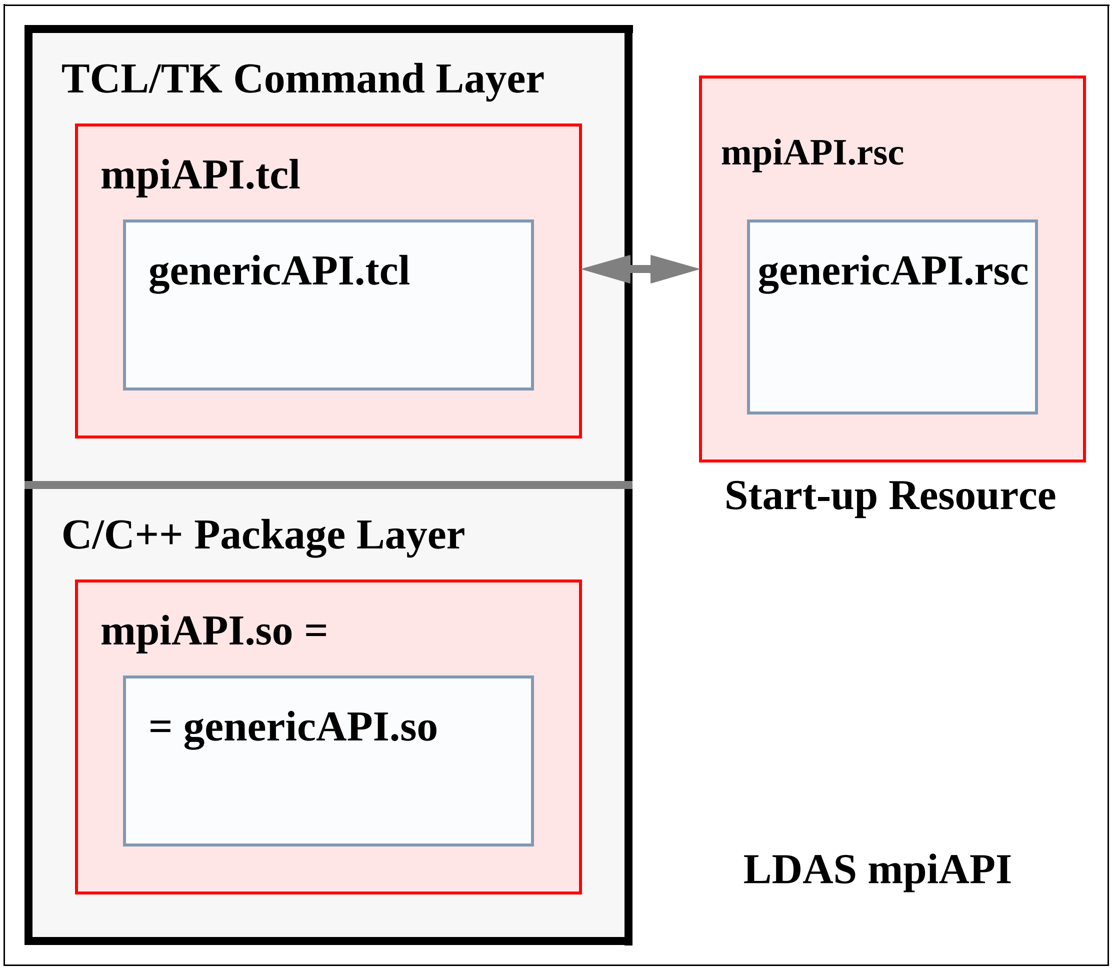
(1) **"#:cont"** where # is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

(2) **"#:kill"** where # is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

The wrapperAPI will typically send a subset of these commands to the mpiAPI upon completion of each cycle of data through the slave processes.

5. The wrapperAPI will provide a method to estimate the number of nodes needed to run the parallel process in real time and calculate the load balancing request as integer nodes, such that the projected ratio is less than or equal to 1.00000, while remaining as close to 1.00000 as possible. This in itself requires that the wrapperAPI be able to extract the length of the data sequence in terms of collection time, while also measuring progress on analysing the data in wall clock time.

II. Component Layers of the LDAS mpiAPI



A. LDAS Distributed mpiAPI:

1. The LDAS distributed mpiAPI is made up of two major layers.
   a) TCL/TK Layer - this layer is the command layer and deals primarily with commands and/or messages and their attributes and/or parameters, as well as communicate with the underlying Package Layer through TCL/TK extensions.
   b) C/C++ Package Layer - this layer is the data engine layer and deals primarily with the binary data and the algorithms and methods needed to manipulate LIGO's data
2. The TCL/TK layer consists of two internal and two external components, designed to optimize code reuse at the level of the command language used in all LDAS API's.
   a) The mpiAPI.tcl - this TCL/TK script contains specialized TCL/TK pro-

cedures and specialized command language extensions which are particular to the mpiAPI in the LDAS architecture.

b) The genericAPI.tcl - this TCL/TK script contains the common TCL/TK procedures and command language extensions found in all LDAS API's. the genericAPI.tcl code will be sourced in the mpiAPI.tcl script.

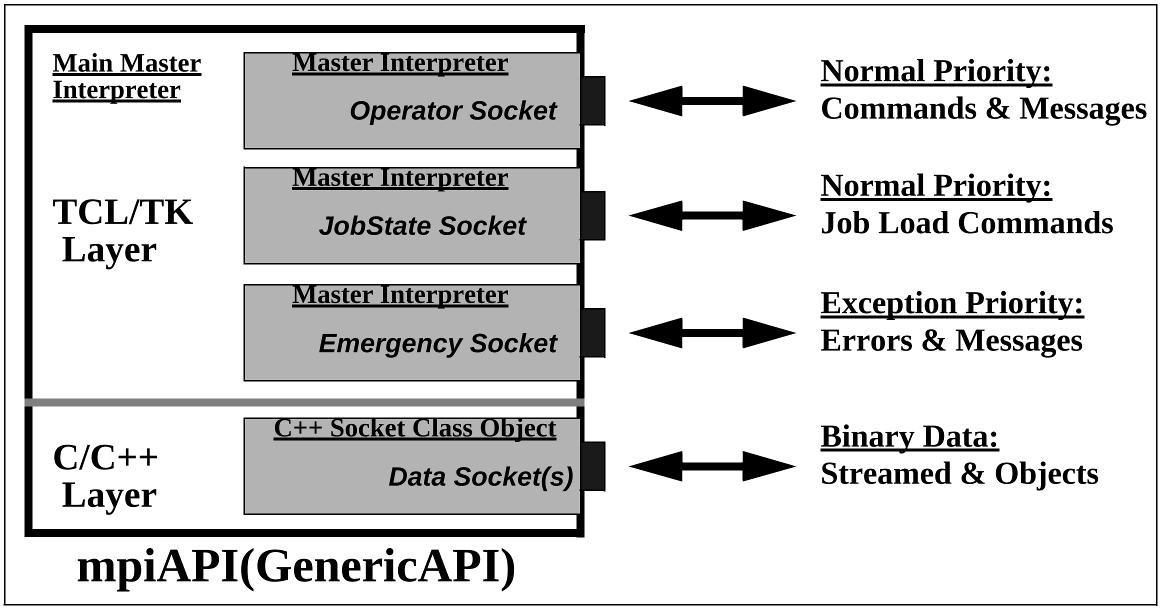
c) The mpiAPI.rsc - this TCL/TK script contains the start-up and configuration defaults which are unique to the mpiAPI.

d) The genericAPI.rsc - this TCL/TK script contains the start-up and configuration defaults which are common to each LDAS API. The genericAPI.rsc will be embedded in the mpiAPI.rsc file.

3. The C/C++ package layer consists of one internal components, developed in C++ and C to take advantage of the higher performance associated with compiled languages which is needed for the types of activities that are being carried out in this layer and loaded as shared objects.

a) The genericAPI.so - this shared object contains the C++ classes and C interface functions needed to extend the command language set of all API's in LDAS, allowing efficiency and optimal code reuse. It will be linked into the mpiAPI.so shared object directly.

III. Communications in mpiAPI from GenericAPI



A. Socket Based Communications in mpiAPI:

1. The genericAPI will provide the mpiAPI with an internet socket within the TCL/TK layer that is the primary communication port for commands and messages of a normal priority. This port is commonly referred to as the *Oper-*

ator Socket to reflect its association with normal operations. Requirements on this socket are that defined by the genericAPI. The genericAPI will also provide the mpiAPI with an internel socket within the TCL/TK layer for exception priority messages. This port is commonly referred to as the *Emergency Socket* to reflect its association with exception handling. The mpiAPI will also have a unique internet server socket within the TCL/TK layer that will be used to receive requests to load balance the queues being managed by the mpiAPI. This JobState Socket will be notified when an MPI job has excess nodes which it does not need. It will also be used by MPI jobs to request additional nodes (if available) for increasing MPI job performance.

2. The genericAPI will provide the mpiAPI with dynamic TCP/IP sockets within the C/C++ layer that is used to communicate all data (*typically binary data*) in the form of streamed binary data or distributed C++ class objects using the ObjectSpace C++ Component Series Socket Library. This port is commonly referred to as the *Data Socket* to reflect its primary duty in communicating data sets. Requirements on these sockets are defined 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. MPI:

1. The parallel software components of the LDAS will use the public domain version of MPI from MPICH, release 1.2 or greater.
2. The use of MPI code within LDAS will be restricted to the C++ interface bindings and the use of object oriented design technologies whenever possible. The templated analysis filters and associated functions are not required to be developed using C++ and object oriented design techniques. However, they must support bindings to the core C++ slave processes.

D. 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.

E. 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. The Make facility is being extended using AutoConfig, AutoMake and LibTools, all from the public domain.

F. 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.

G. Documentation:

1. DOC++ is a documentation system for C/C++ and Java. It generates LaTeX or HTML documents, providing for sophisticated online 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 online browsing system to the LDAS help files. Documents include a hypertext linked table of contents and a hierarchical structured format.