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

Document Type **LIGO-T990097-01-** **E** 03/23/2000

The wrapper 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 wrapper API's
baseline requirements

James Kent Blackburn

California Institute of Technology
LIGO Data Analysis Group
March 23, 2000

I. Introduction

A. General Description:

1. The wrapperAPI is responsible for executing 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.
2. The wrapperAPI will be written entirely in C and C++. No TCL/TK will be used in this LDAS API.
3. The wrapperAPI will be initiated using the mpirun command. This command will be started solely by the mpiAPI. The mpiAPI will determine the appropriate values for the mpirun command, as well as the appropriate commands for the wrapperAPI and pass these as command line arguments to mpirun. The wrapperAPI will interpret its own command line arguments which are automatically passed to the wrapperAPI by the mpirun command.
4. The wrapperAPI will process a "chunk" of data of finite length. For each new "chunk" of data a new wrapperAPI will be started up by the mpiAPI.

B. The wrapperAPI TCL/TK Script's Requirements:

1. None applicable to the wrapperAPI since it will not have TCL/TK code.

C. The wrapperAPI Shared Object Package Requirements:

1. The wrapperAPI will be written as an MPI executable, not as a shared object library. However, the wrapperAPI will make extensive use of other LDAS shared object libraries.
2. The wrapperAPI will dynamically load a shared object containing the templated filter algorithms used for LIGO data analysis.

D. The mpirun command Requirements:

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.

1. 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. If N exceeds the number of processors in the Beowulf Cluster, it will automatically be set to the corresponding value.

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 discretion 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. Their 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.

2. 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. Any integer values in the list greater than N will be ignored.

b) **-dynlib=/path/libname.a** is used to specify the full (*absolute*) path and file name of the dynamically loaded shared object library containing the templated filter algorithms. Note: This library 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, warnings, errors, 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 formatted 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) **-nodeDutyCycle=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 consistent 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 libraries which wish to use this mechanism must provide the parameter checks internal to the dynamically loaded library. Numeric parameters **a,b,c,d,...** without decimal places will represent integers. All other numeric parameters will be interpreted as doubles. Everything else will be C strings.

h) **-realTimeRatio=n.mmmmm** is used to specify the desired ratio of the time required to process the data to the time contained within the data segment. As an example, a value of 0.90 would request that 54 second be used to analyze 60 seconds worth of data.

E. The wrapperAPI executable requirements:

1. The wrapperAPI will be a parallel program based on MPICH version 1.2 or later designed to properly operate strictly within the LDAS system.
2. The target platform for running the wrapperAPI will be any LDAS Beowulf Cluster. *Current design plans for this cluster are based on Intel Pentium PCs using the Redhat 6.x operating system. However, this choice for cluster technology may evolve as the commodity PC market changes. No choices for the wrapperAPI should strictly assume this cluster technology*.
3. The wrapperAPI will divide parallel processing into two general categories:
   a) A single master process responsible for communicating commands with mpiAPI and communicating with other LDAS APIs which send and receive data in the form of ILWD objects through LDAS API data sockets. The master process will also act as the central parallel node used by all slave processes in the MPI environment. The master process is also responsible for translating the ILWD data objects into MPI data types necessary for parallel communications.
   b) A collection of slave processes responsible for carrying out the templated filtering in a trivially parallel manor. Each slave process must dynamically load the template analysis algorithm library as part of its initialization. The slaves will communicate analysis results back to the master using MPI data types.
4. The format of ILWD (Internal Light Weight Data) being received by the wrapperAPI (typically from the dataConditioningAPI) will be of the form of a collection of adc channel data sequences in either the time domain or the frequency domain. A short ASCII ILWD example is given below:

```
<ilwd name='dataConditioningAPI:container' size='2'>
     <ilwd name='XYZ:channel:sequence' size='9'>
         <lstring name='real:domain' dims='4'>TIME</lstring>
         <int_8u name='gps_sec:start_time' units='sec'> 62348734 </int_4u>
         <int_8u name='gps_nan:start_time' units='nanosec'> 0 </int_4u>
         <int_8u name='gps_sec:stop_time' units='sec'> 62348735 </int_4u>
         <int_8u name='gps_nan:stop_time' units='nanosec'> 0 </int_4u>
         <real_8 name='time:step_size' units='sec'> 0.0625 </real_8>
         <real_4 name='filterX:decimation' dims='1'>1024.000</real_4>
         <real_4 name='methodII:line_removal' dims='3' units='HZ'> 60.0,
         180.0, 360.0 </real_4>
         ...(other filter history)...
         <real_4 name='real:data' dims='16' units='volts'> -0.01, -0.05,
```

```
        -0.02, -0.00, 0.01, 0.04, 0.08, 0.11, 0.03, -0.04, -0.07, -0.03,
        0.01, 0.04, 0.06, 0.06 </real_4>
    </ilwd>
    <ilwd name='XYZ:calibration:sequence' size='7'>
        <lstring name='complex:domain' dims='4'>FREQ</lstring>
        <int_8u name='gps_sec:start_time' units='sec'> 62348734 </int_4u>
        <int_8u name='gps_nan:start_time' units='nanosec'> 0 </int_4u>
        <real_8 name='start_freq' units='hz'> -2048.0 </real_8>
        <real_8 name='stop_freq' units='hz'> 2048.0 </real_8>
        <real_8 name='freq:step_size' units='hz'> 256.0 </real_8>
        <lstring name='hann:window'>overlap=15%</lstring>
        ...(other filter history)...
        <real_4 name='real:data' dims='17' units='strain/volt'> -0.31,
        -0.55, -0.12, -0.40, 0.61, 0.24, 0.58, 0.11, 0.13, -0.64, -0.87,
        -0.53, 0.71, 0.84, 0.26, 0.56, 0.91 </real_4>
        <real_4 name='imag:data' dims='17' units='strain/volt'> -0.01,
        -0.05, -0.02, -0.00, 0.01, 0.04, 0.08, 0.11, 0.03, -0.04, -0.07,
        -0.03, 0.01, 0.04, 0.06, 0.06, 0.00 </real_4>
    </ilwd>
    <ilwd name='Wavelet:time-frequency:sequence' size='7'>
        <lstring name='complex:domain' dims='4'>BOTH</lstring>
        <int_8u name='gps_sec:start_time' units='sec'> 62348734 </int_4u>
        <int_8u name='gps_nan:start_time' units='nanosec'> 0 </int_4u>
        <int_8u name='gps_sec:stop_time' units='sec'> 62348735 </int_4u>
        <int_8u name='gps_nan:stop_time' units='nanosec'> 0 </int_4u>
        <real_8 name='time:step_size' units='sec'> 0.0625 </real_8>
        <real_8 name='start_freq' units='hz'> -2048.0 </real_8>
        <real_8 name='stop_freq' units='hz'> 2048.0 </real_8>
        <real_8 name='freq:step_size' units='hz'> 256.0 </real_8>
        <lstring name='uwm-method:wavelet' dims='2' size='14'>
        a=0.30\, b=1.50</lstring>
        ...(other filter history)...
        <real_4 name='real:data' dims='17' units='strain/volt'> -0.31,
        -0.55, -0.12, -0.40, 0.61, 0.24, 0.58, 0.11, 0.13, -0.64, -0.87,
        -0.53, 0.71, 0.84, 0.26, 0.56, 0.91 </real_4>
        <real_4 name='imag:data' dims='17' units='strain/volt'> -0.01,
        -0.05, -0.02, -0.00, 0.01, 0.04, 0.08, 0.11, 0.03, -0.04, -0.07,
        -0.03, 0.01, 0.04, 0.06, 0.06, 0.00 </real_4>
    </ilwd>
    ...(other channels of data)...
</ilwd>
```

In this example two "sequences" of data are sent from the dataConditioningAPI to the wrapperAPI. The first represents channel XYZ in the single precision time domain for a given start time and stop time in GPS seconds and nanoseconds (along with the time interval between time stamps). The channel was decimated by a factor of 1024 using the filter named *filterX*. The next filter applied according to this sequence container is that of line removal using filter *methodII*. This filter requires an array of real_4 frequencies corresponding to the lines which were removed (*here it is the 60hz, 180hz, and 360hz lines*). Other data conditioning filters may have been applied and they would come here in the container. Next comes the actual sequence of data. It is identified by the name="xxxx:data", where xxxx could be real or imag depending on the domain.

The second sequence container is for a calibration of the adc XYZ. It is data in the frequency domain and is represented by complex data. The frequency range is specified, as well as any filters (*here a hann window function with an overlap of 15%*), followed by the actual data in two distinct ilwd arrays, one for the real component and the other for the imaginary component.

5. It will be the responsibility of the master process to open a ilwd class object socket connection to the LDAS API (*typically the dataConditioningAPI*) server socket specified by the **-dataAPI** command line option and receive the ilwd data object. The master process will interpret (parse) this ilwd data object and reconstruct it as a structured MPI::Datatype for use in MPI communications with the slave processors.
6. It will be the responsibility of the master process to establish ilwd class object socket connections to the LDAS API (typically the eventManagerAPI) server socket specified by the **-resultAPI** command line option and send ilwd table data objects (*along with any optional ilwd sequence data in containers of the type outlined in item **4** above*) when data is ready to be transferred. The master will construct these ilwd objects out of the results received from the slaves as MPI data types.
7. The master process will also 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 text strings sent to the mpiAPI's listening socket designated by the **-mpiAPIport** command line option using just a simple unix socket connection. In general, multiple commands may be sent at a time in a set to the mpiAPI, each separated by a newline `'\n'` character and each set using the same request ID #. 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 four 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 four 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 occurred 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 logs 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 occurred 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 logs 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 logs 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"** is the default "nominal" command 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 logs 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 logs 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.

h) The set of commands will consist of either of the following sets:

(1) During regular processing:

one command from **a)** or **b)** or **f)**

plus

command **c)** if warnings occurred

plus

command **d)** if errors occurred

plus

command **e)** and command **g)**.

(2) At the completion all analysis:

command **e)** with progress at 100.00%

plus

command **c)** if warnings occurred.

8. 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 **realTimeRatio**, while remaining as close to **realTimeRatio** 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 analyzing the data in wall clock time.

F. WrapperAPI Dynamically Loaded Library Requirements

1. The wrapperAPI will load all template analysis algorithms from dynamically loaded libraries from the local Beowulf file space. Each type of search will have its own dynamically loaded library. This will allow each mpirun command to be associated with a particular type of search by the particular dynamically loaded library assigned by the command line arguments (see **-dynlib** command line option above). The wrapperAPI will load the dynamically loaded library using the Unix dopen and related functions:

```
SYNOPSYS
   #include <dlnfcn.h>

   void *dlopen (const char *filename, int flag);
   const char *dlerror( void );
   void *dlsym( void *handle, char *symbol );
   int dclose (void *handle);

   Special symbols _init, _fini.
```

   The flag variable used in the dlopen call must not include the RTLD_GLOBAL flag value. See the Unix (Linux) manpages for more details on the use of dlopen.

2. Each slave process will dlopen the dynamically loaded library specified by the command line option. Each dynamically loaded library must contain four C functions used to interface the wrapperAPI with the dynamically loaded library. All four of these required functions will be called in order by each slave. These functions will be defined within an `extern "C" {}` from the

wrapperAPI's C++ code. Each function will use an error or warning message string variable. This message string will be nulled out before each call to one of the four interface functions. Any non-null value returned by one of the interface messages will be cached by the wrapperAPI for command message communications with the mpiAPI at a convenient time. These messages will be logged into the LDAS logging system by the mpiAPI:

a) The first required C function is used to initialize the dynamically loaded library and to configure the parameterization as defined by the command line option **-filterparams**. It is called once by the master and each slave in the MPI_COMM_WORLD:

```
SYNOPSIS

extern "C" {

#include "lalDataTypes.h"
#include "wrapperInterface.h"

CHAR initMessage[1024] = "";
INT4 filterInit( INT4 argc, CHAR* argv[], CHAR* initMessage );
}
```

where **`argc`** is a count of the number of parameter arguments being passed into the dynamically loaded library from the -filterparams command line option and **`argv`** is an array of pointers to the -filterparams command line options. Note that the argv[0] pointer will store the string "-filterparams". This will allow filterInit() to verify that the options are associated with the appropriate command line option. As an example, if the command line contained -filterparams=[1.0, 3.0, 10.0], the value of argc would be 4 and argv[0] would point to "-filterparams", argv[1] would point to "1.0", argv[2] would point to "3.0", and argv[3] would point to "10.0". The filterInit() function will be responsible for parsing the character values pointed to by argv into numerical values. As noted earlier all floating point values MUST have a decimal point. Non numerical values are allowed by this mechanism, however they should be discouraged.

The **filterInit()** function returns a 0 (zero) value if it is successful in parsing and interpreting the values from argv (including any and all range checking on these values). In the event of an error, the filterInit() function must return a 1 (one) and assign a unique error message of no more than 1024 characters to the initMessage variable. In the event that a warning condition is established, the value returned by filterInit shall be -1 and the corresponding warning message will be stored in the **`initMessage`** variable.

The values parsed from the **`argv`** pointers must be stored internally in the global variable space of the dynamically loaded library if they are to be used by other functions within the dynamically loaded library.

b) The second required function is used to calculate the total number of template filters that will be used in the search algorithms contained within the dynamically loaded library. It is called once by the master and each slave in the MPI_COMM_WORLD. Filters will be called using an index counting scheme that ranges from 1 to the maximum specified by the value returned by this function. It may also be used to internally construct a one-to-one identification map between the sequential index values and a specific set of parameter values used by the templated filter algorithms:

```
SYNOPSIS

extern "C" {

#include "lalDataTypes.h"
#include "wrapperInterface.h"

CHAR indexMessage[1024] = "";
INT4 indexFilters( char* indexMessage );
}
```

If the indexFilter() function has an error then it must return with a value of 0 (zero) and assign a unique error message to **`initMessage`** of no more than 1024 characters. This function will be guaranteed to be called by each node or processes of the wrapperAPI before the first templated search filter is called.

c) The third required function is used to carry out any further pre-conditioning of data which may not be possible for the LDAS dataConditionAPI. It is called by each slave process. It will have write privileges on the input data structure:

```
SYNOPSIS

extern "C" {

#include <mpi.h>
#include "lalDataTypes.h"
#include "wrapperInterface.h"

CHAR filterMessage[1024] = "";
inPut data[];
MPI_Comm* comm;

INT4 conditionData( inPut* data, CHAR* conditionMessage,
MPI_Comm* comm );
}
```

The conditionData() function is called with the **`inPut`** data structure which contains the data to be further conditioned. **NOTE:** It is highly recommended that use of this function be held to a minimum. Ideally any functionality present in this function should be migrated upstream into the dataConditionAPI. Hence, under most circumstances it simply returns successfully as described below without modifying the **`inPut`** data structure. The justification for this being that it is less expensive to condition the data upstream once than to condition it a total of N times on each of the slave processes. The **conditionData()** function should never overwrite data found in the **inPut** data structure, only extend it with new data local to the node it occupies.

The **conditionData()** function will return an integer value of 0 if successful and an integer value of 1 along with a unique error message of no more than 1024 characters in the **`conditionMessage`** variable. If a warning occurs the **conditionData()** function will return (***with results***) a value -1 along with the warning messages stored in the **`conditionMessage`** variable.

The local and global node information are also passed into the **conditionData()** function using the MPI communicator structure **comm**. Using this structure it is possible for **conditionData()** to identify the local node, the total nodes in the communicator and to make node-to-node communications if this is the desired way to split up input data. NOTE: It is important to consider in the design of **conditionData()** that it will be be called multiple times as part of a loop within the wrapperAPI. Also, each time that it is called the MPI communicator may have been modified to include a different set of nodes. Thus, the **conditionData()** may require static internal data to recognize when to repeat steps or when to redo steps associated with conditioning data, including node-to-node communications which may be used to pass data to **templateFilter()** instead of using the conventional path through the **inPut** data structure passed by the call to **templateFilter()**. This may be useful, for example, to distribute local data in a distributed FFT algorithm.

d) The fourth required function is the parallel search engine. It is used to apply a particular set of template filter parameters to the data and to return the results of the filtration algorithms. It will typically be called repeatedly in a loop on each slave process until all template filters have been analyzed. As such it has a highly specialized set of input and output structures which must be general enough to carry out template filter gravitational wave searches from interferometer data:

```
SYNOPSIS

extern “C” {

#include <mpi.h>
#include “lalDataTypes.h”
#include “wrapperInterface.h”
```

```
INT4 beginTemplate;
INT4 endTemplate;
CHAR filterMessage[1024] = "";
inPut data[];
outPut result[1+endTemplate-beginTemplate];
MPI_Comm* comm;

INT4 templateFilters( int beginTemplate, int endTemplate,
const inPut* data, outPut* result, CHAR* filterMessage,
MPI_Comm* comm );
}
```

The **templateFilter()** function is called only by the active slaves over and over again in a loop with two integers **beginTemplate** and ending at **endTemplate** until all templates have been analyzed. These two variable can be identical, signifying that only one template will be applied to the data found in the inPut data structure prior to returning the results, otherwise each template will be analyzed using the data in **inPut** before returning. The **templateFilters()** function must, internal to the dynamically loaded library, associate this template index to a set of physical parameters associated with the template model. The **templateFilters()** function must cast the **inPut** into LAL standard data structures appropriate for the internal LAL algorithms being used. The **templateFilters()** function must cast results into the **outPut** structure specified above.

The typedef **inPut** structure will be an array of data sequences associated with pre-conditioned interferometer channel date. In most cases only one sequence containing the pre-conditioned strain signal from the interferometer will be contained in this structure. However, using the array construct, more complex templates using multi-channel data sequences are supported. **NOTE:** The **templateFilters()** function must not modify the contents of this typedef **inPut** structure as it may be needed for further analysis by the slave process using a different set of template indices.

The typedef **outPut** structure will be an array large enough to hold all results from all templates (1+**endTemplate**-**beginTemplate**) analyzed during the call to the **templateFilters()** function.

The results of the **templateFilters()** function are stored in the **results** array of **outPut** typedef structures. The wrapperAPI is responsible for evaluating the **significant** attribute of each element of the **results** array and in the event it is **TRUE** making a deep copy of the contents of the **dataBase** doubly linked list typedef structure in the memory space of the wrapperAPI, allowing the **templateFilters()** function to reuse this variable space. NOTE: This includes making a copy of the **optional** array of sequence typedef structure attributes if they are not NULL.

The **templateFilters()** function will return an integer value of 0 if successful and an integer value of 1 along with a unique error message of no more than 1024 characters in the **filterMessage** variable. If a warning occurs the templateFilters() function will return (***with results***) a value -1 along with the warning messages stored in the **filterMessage** variable.

The local and global node information are also passed into the **templateFilters()** function using the MPI communicator structure **comm**. Using this structure it is possible for **templateFilter()** to identify the local node, the total nodes in the communicator and to make node-to-node communications if this is the desired way to split up the filter analysis. NOTE: It is important to consider in the design of **templateFilters()** that it will be be called multiple times as part of a loop within the wrapperAPI. Also, each time that it is called the MPI communicator may have been modified to include a different set of nodes. Thus, the **templateFilters()** may require static internal data to recognize when to repeat steps or when to redo steps associated with analyzing data, including node-to-node communications. This may be useful, for example, to distribute local data in a distributed FFT algorithm.

e) The last required function is used to free up any dynamically allocated memory that may be associated with the template filter search algorithms found in the dynamically loaded library. It is called once at the completion of all calls to **templateFilter()** by the master and all slaves of the MPI_COMM_WORLD.

SYNOPSIS

```
extern "C" {

#include "lalDataTypes.h"
#include "wrapperInterface.h"

CHAR freeMessage[1024] = "";
INT4 freeFilters( CHAR* freeMessage );
}
```

The specifics of this function will depend on the internal workings of the algorithms found in each particular dynamically loaded search library. For example, if a one-to-one identification map was dynamically constructed as an index, then it must be freed upon completion of the search. This function will return an integer value of one upon successfully deallocating memory and a 1 if the memory deallocation fails along with an error message of no more than 1024 characters in the **freeMessage** variable.

f) The definitions for these interface functions and data types outlined above will be included into code via the **wrapperInterface.h** file. This header file will contain valid ANSI C syntax. Also, this header file must be guarded and must be included within an **extern "C" {}** block for C++ code as previously illustrated. The structures defined within `wrapperInterface.h` are given below:

```
/* prevent multiple inclusions of header file */

#ifndef WRAPPER_INTERFACE
#define WRAPPER_INTERFACE

typedef enum {timeD, freqD, bothD} domain;

typedef enum { boolean_1u, char_s, char_u,
int_2s, int_2u, int_4s, int_4u, int_8s, int_8u,
real_4, real_8, complex_8, complex_16
} datatype;

/* include this file to get LAL datatypes */
#include <mpi.h>
#include "lalDatatypes.h /* LAL header file */

typedef union { /* these pointer types MUST exist in LAL! */
BOOLEAN *boolean; /* pointer to BOOLEAN type */
CHAR *chars; /* pointer to CHAR */
UCHAR *charu; /* pointer to UCHAR */
INT2 *int2s; /* pointer to INT2 */
UINT2 *int2u; /* pointer to UINT2 */
INT4 *int4s; /* pointer to INT4 */
UINT4 *uint4u; /* pointer to UINT4 */
INT8 *int8s; /* pointer to INT8 */
UINT8 *int8u; /* pointer to UINT8 */
REAL4 *real4; /* pointer to REAL4 */
REAL8 *real8; /* pointer to REAL8 */
COMPLEX8 *complex8; /* pointer to COMPLEX8 */
COMPLEX16 *complex16; /* pointer to COMPLEX16 */
} dataPointer; /* union supporting pointer type checking */

typedef struct {
UINT8 numberSamples; /* no. of data samples in interval */
UINT8 startSec; /* GPS start time in seconds */
UINT8 startNan; /* GPS start time in nanoseconds */
UINT8 stopSec; /* GPS stop time in seconds */
UINT8 stopNan; /* GPS stop time in nanoseconds */
REAL8 timeStepSize; /* uniform step size in seconds */
} gpsTimeInterval; /* time domain interval */

typedef struct {
UINT8 numberSamples; /* no. of data samples in interval */
UINT8 gpsStartTimeSec; /* GPS start time in seconds */
UINT8 gpsStartTimeNan; /* GPS start time in nanoseconds */
REAL8 startFreq; /* starting frequency in hertz */
REAL8 stopFreq; /* ending frequency in hertz */
```

```
REAL8 freqStepSize; /* uniform step size in hertz */
} frequencyInterval; /* frequency domain interval */

typedef struct {
UINT8 numberSamples; /* no. of data samples in interval */
UINT8 gpsStartTimeSec; /* GPS start time in seconds */
UINT8 gpsStartTimeNan; /* GPS start time in nanoseconds */
UINT8 gpsStopTimeSec; /* GPS stop time in seconds */
UINT8 gpsStopTimeNan; /* GPS stop time in nanoseconds */
REAL8 startFreq; /* starting frequency in hertz */
REAL8 stopFreq; /* ending frequency in hertz */
REAL8 timeStepSize; /* uniform step size in seconds */
REAL8 freqStepSize; /* uniform step size in hertz */
} timeFreqInterval; /* frequency domain interval */

typedef union {
gpsTimeInterval dTime; /* time domain interval info */
frequencyInterval dFreq; /* frequency domain interval info */
timeFreqInterval dBoth; /* time+frequency domain interval info */
} interval;

#define maxHistoryName 64
#define maxHistoryUnits 64

typedef struct dcHistoryTag {
struct dcHistoryTag *previous; /* previous data cond. filter */
CHAR name[maxHistoryName]; /* data conditioning filter name */
CHAR units[maxHistoryUnits]; /* data conditioning filter units */
datatype type; /* data type for column */
UINT4 numberValues; /* no. rows to add to column */
dataPointer value; /* pointer to table's column data */
struct dcHistoryTag *next; /* next data cond. filter */
} dcHistory; /* this is a bi-directional linked list */

#define maxStateName 64

typedef struct stateVectorTag {
struct stateVectorTag *previous; /* previous state vector */
CHAR stateName[maxStateName]; /* name of state*/
multiDimData *store; /* reuseable state vector data store */
struct stateVectorTag *next; /* next state vector */
} stateVector;

typedef struct {
CHAR name[256] /* name of the data in dataPointer */
CHAR units[256] /* comma separated units of the data */
domain space; /* either time, frequency or both domain */
datatype type; /* type of data in pointer */
interval range; /* epoch of time/frequency for data */
UINT4 numberDimensions; /* no. of dimensions in data */
UINT4 dimensions[]; /* no. of elements along each dimension */
```

```
dcHistory history; /* data conditioning history */
dataPointer data; /* pointer to multi-dimensional data */
} multiDimData;

typedef struct {
UINT4 numberSequences; /* number of data channels in inPut */
stateVector states; /* input state vector information */
multiDimData sequences[]; /* array of conditioned data */
} inPut;

typedef enum { binaryInspiral, ringDown, periodic, burst,
stocastic, timeFreq, instrumental, protoType, experimental
} catagory; /* astrophysical/instrumental search catagories */

#define dbNameLimit 19 /* Note DB2 limits names to 18 letters */

typedef struct dataBaseTag {
struct dataBaseTag *previous; /* previous table data set */
CHAR tableName[dbNameLimit]; /* name of LDAS table */
CHAR columnName[dbNameLimit]; /* column name in LDAS table */
datatype type; /* data type for column */
UINT4 numberRows; /* no. rows to add to column */
dataPointer rows; /* pointer to table's column data */
struct dataBaseTag *next; /* next table data set */
} dataBase; /* this is a bi-directional linked list */

typedef struct {
INT8 templateNumber; /* number of template results in outPut */
catagory search; /* type of astrophysical/instrumental search */
BOOLEAN significant; /* signals that require post-processing */
stateVector states; /* output state vector information */
dataBase results; /* template results to be ingested into DB */
multiDimData *optional; /* optional sequences (1/template) */
} outPut;

/* ANSI C prototypes for four interfacing functions */

INT4 filterInit( INT4 argc, CHAR* argv[], CHAR* initMessage );

INT4 indexFilters( CHAR* indexMessage );

INT4 conditionData( inPut* data, CHAR* conditionMessage,
MPI_Comm* comm );

INT4 templateFilters( INT4 beginTemplate, INT4 endTemplate,
const inPut* data, outPut* result, CHAR* filterMessage,
MPI_Comm* comm );

INT4 freeFilters( CHAR* freeMessage );

#endif
```

This `wrapperInterface.h` header file learns about the LAL standard datatypes by including the `lalDatatypes.h` standard header file. This header file must at a minimum contain the following definitions:

```
#ifndef lalDatatypesH
#define lalDatatypesH

#include "config.h"

typedef char CHAR;

typedef unsigned char UCHAR;

typedef unsigned char BOOLEAN;

#if SIZEOF_SHORT == 2
typedef short INT2;
typedef unsigned short UINT2;
#elif SIZEOF_INT == 2
typedef int INT2;
typedef unsigned int UINT2;
#else
#error "ERROR: NO 2 BYTE INTEGER FOUND"
#endif

#if SIZEOF_INT == 4
typedef int INT4;
typedef unsigned int UINT4;
#elif SIZEOF_LONG == 4
typedef long INT4;
typedef unsigned long UINT4;
#else
#error "ERROR: NO 4 BYTE INTEGER FOUND"
#endif

#if SIZEOF_LONG == 8
typedef long INT8;
typedef unsigned long UINT8;
#elif SIZEOF_LONG_LONG == 8
typedef long long INT8;
typedef unsigned long long UINT8;
#else
#error "ERROR: NO 8 BYTE INTEGER FOUND"
#endif

#if SIZEOF_FLOAT == 4
typedef float REAL4;
#else
#error "ERROR: NO 4 BYTE REAL FOUND"
```

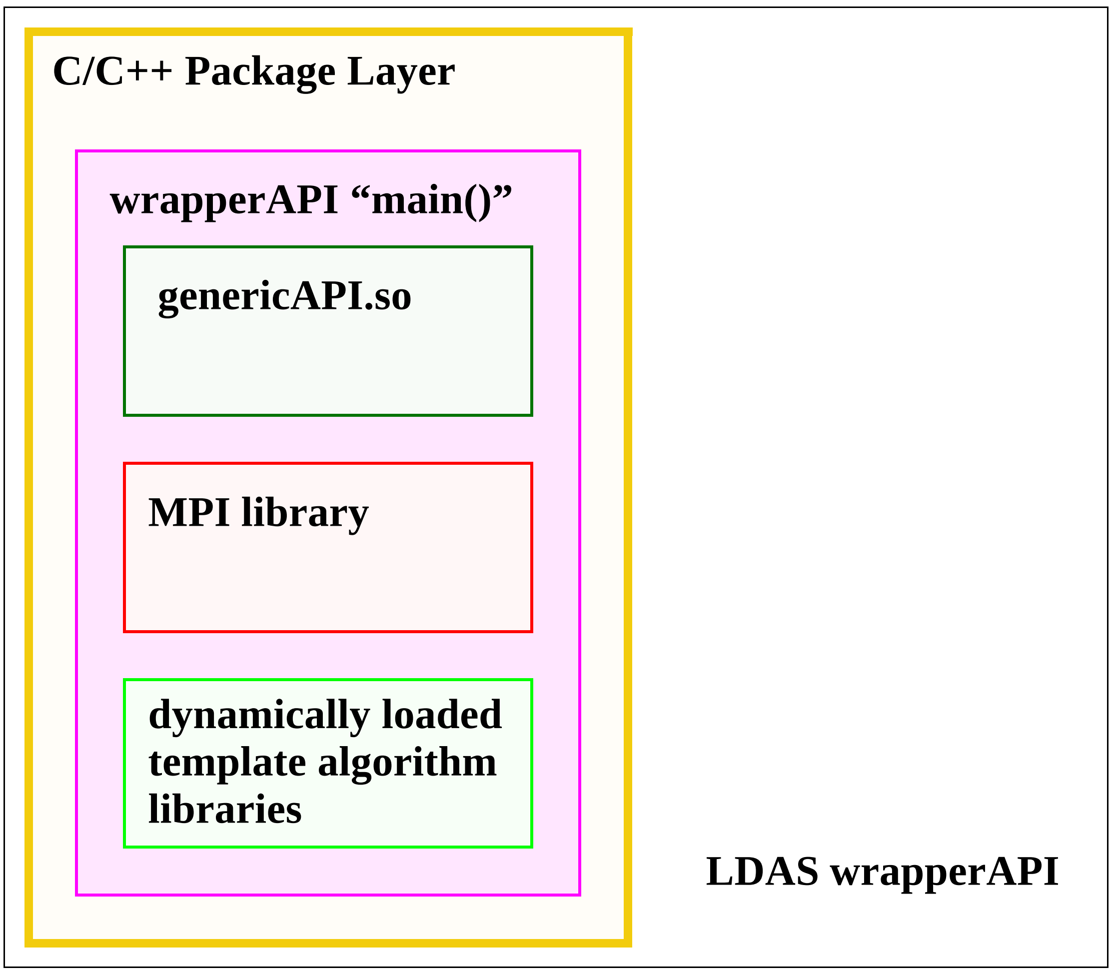
```
#endif

#if SIZEOF_DOUBLE == 8
typedef float REAL8;
#else
#error "ERROR: NO 8 BYTE REAL FOUND"
#endif

typedef struct {
  REAL4 re;
  REAL4 im;
} COMPLEX8;

typedef struct {
  REAL8 re;
  REAL8 im;
} COMPLEX16;

#endif
```

Notice that this **lalDatatypes.h** header file also must include the LDAS **config.h** header file which is generated by the LDAS autoconfig scripts at configuration time. In the event that this file is not available, the necessary definitions found in this file for both SPARC Solaris and Intel Pentium Linux computers using the GCC 2.95.2 compiler can be placed in a mock-up **config.h** file which looks like the following:

```
/* for Intel Pentium Linux and SPARC Solaris using */
/* GCC version 2.95.2 */

#ifndef TypeConfigH
#define TypeConfigH

/* The number of bytes in a double.  */
#define SIZEOF_DOUBLE 8

/* The number of bytes in a float.  */
#define SIZEOF_FLOAT 4

/* The number of bytes in a int.  */
#define SIZEOF_INT 4

/* The number of bytes in a long.  */
#define SIZEOF_LONG 4

/* The number of bytes in a long long.  */
#define SIZEOF_LONG_LONG 8

/* The number of bytes in a short.  */
#define SIZEOF_SHORT 2

#endif
```

This mock-up file may not contain any other definitions that conflict with LDAS. In the integrated build of the wrapperAPI, the config.h file generated by the LDAS autoconfig scripts shall be used to define these sizes. NOTE: The SIZEOF_UNSIGNED_* are not needed as they are guaranteed to be consistent with signed sizes on the LDAS target platforms at this time (early 2000).

II. Component Layers of the LDAS wrapperAPI



A. LDAS wrapperAPI:

1. The LDAS wrapperAPI is made up of a single C/C++ layer.

   a) 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 C/C++ package layer consists of three 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.
   a) The genericAPI.so - this shared object contains the C++ classes and C interface functions needed to communicate LDAS ilwd data as C++ objects through sockets. It will be linked to the wrapperAPI executable.
   b) The MPI library - this is the Message Passing Interface library used to communicate MPI based messages and data types between nodes of the parallel process.
   c) The dynamically loaded template algorithm library - this is the library that contains the algorithms and functions necessary to carry out template based parallel filtering (searches) of LIGO's data. It will be loaded as a shared object using the Unix dlopen interface calls.

III. LDAS interfaces to wrapperAPI

A. Initiation

1. The LDAS mpiAPI will initiate the wrapperAPI as a stand-alone executable using the mpirun command script. The mpiAPI will be responsible for constructing all command line arguments to the wrapperAPI, this includes options for mpirun as well as options for wrapperAPI.

B. Commands

1. The wrapperAPI will open a Unix socket connection with the mpiAPI's job-state port for the purpose of sending and receiving text commands used to load balance and report status.

C. LDAS Data

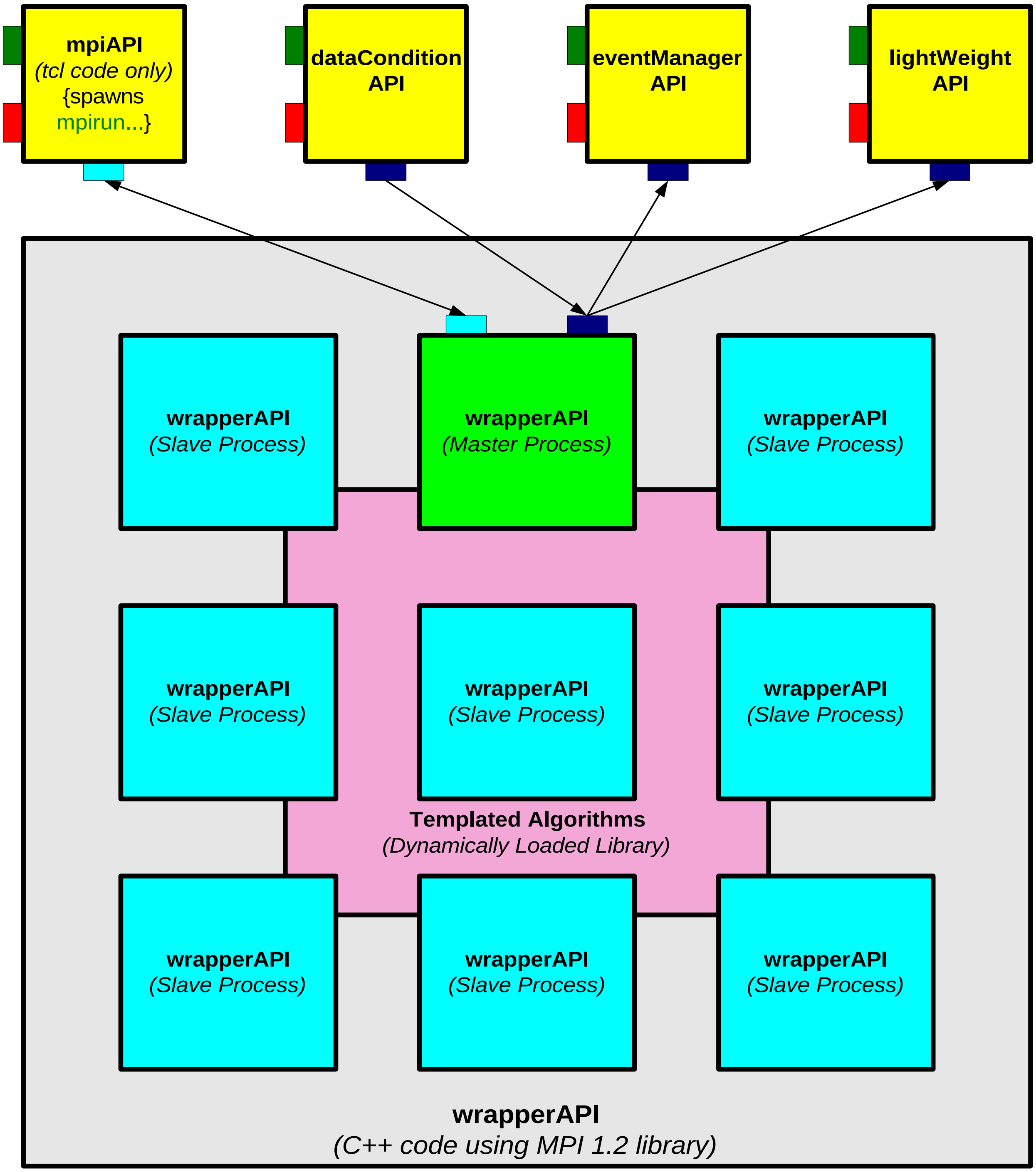
1. The wrapperAPI will use the LDAS data sockets for communicating ILWD data with the LDAS system. The functionality to create and manage these data sockets will be derived from the genericAPI's shared object library.

D. MPI

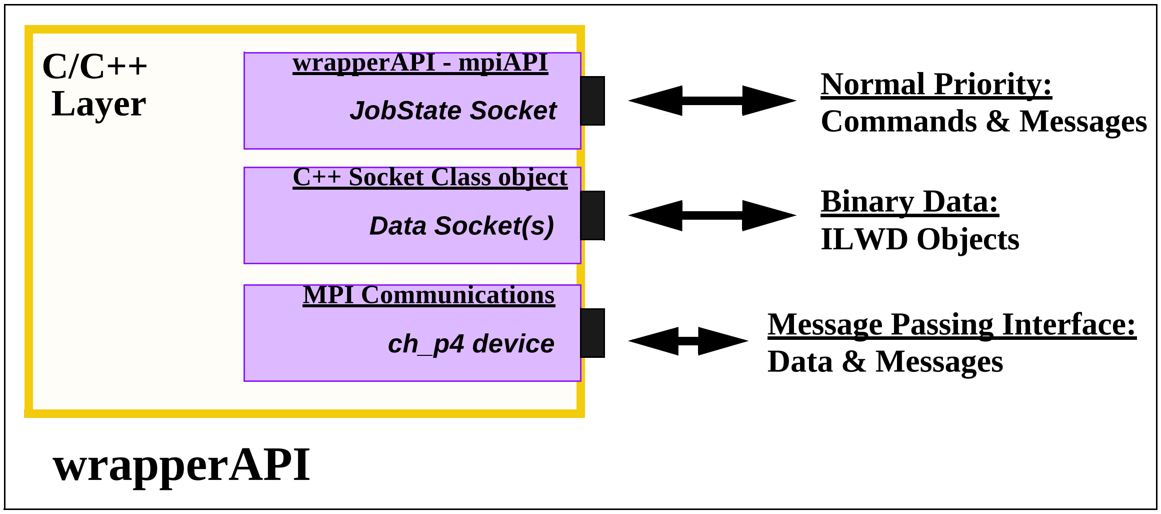
1. The wrapperAPI is a parallel process running on many different nodes of a clustered topology of computers. The software for communicating data and messages between these distinct processes will be the Message Passing Interface (MPI) library. The wrapperAPI will support simplistic parallel processing based on a large set of filter algorithms (templates) being applied in parallel to a single segment of data. The master process will be responsible for sending the data and receiving results of the analysis. The slave processes will carry out the algorithms on the data.

E. Dynamically Loaded Template Algorithm Shared Objects

1. The wrapperAPI will dynamically load any shared object that conforms to the design standard for templated filter algorithms outlined in this document. The algorithms found in this shared object will be performed within the slave process of the parallel process.



IV. Communications in wrapperAPI



A. Socket Based Communications in wrapperAPI:

1. The wrapperAPI will establish a Unix socket for communicating jobstate commands and messages between the mpAPI and itself. This socket will be managed by the master process.
2. The genericAPI will provide the wrapperAPI with dynamically allocated TCP/IP sockets within the C/C++ layer that is used to communicate LDAS data (*typically binary data*) in the form of streamed binary data or distributed ILWD 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 LDAS data sets. Requirements on these sockets are defined by the genericAPI.
3. The MPI library will provide the communications interface (typically the ch_p4 device for clusters) used to share MPI data types and messages between nodes of the parallel process.

V. 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 has been 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. PERCEPS is a documentation system for C/C++. It generates HTML docu-

ments, 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.

VI. WrapperAPI Pseudo-Code

A. Illustration of flow control:

1. The following pseudo-code illustrates the steps take by the wrapperAPI. The details are left for implementation.

```
/ wrapperAPI pseudo-code
extern "C" {
#include "wrapperInterface.h"
}
// On every node in MPI_COMM_WORLD perform:
{
parseCommandLineOptions();
loadDynamicSharedObjects();
errorTestInit( filterInit( argc, argv, initMessage ) );
errorTestIndex( totalTemplates = indexFilters( indexMessage ) );
LBComm = createLoadBalanceCommunicator();
}
// On every node in Load Balance Communicator perform:
{
while ( notFinished() ) {

errorTestDC( conditionData( data, conditionMessage, LBComm ) );
errorTestTF( templateFilters( beginTemplate, endTemplate,
data, result, filterMessage, LBComm ) );
masterNodeGathersResults();
masterNodeCalculatesLoad();
masterNodeInformsLDASmpiAPI();
// On every node in MPI_COMM_WORLD perform:
{
LBComm = createLoadBalanceCommunicator();
}
}
}
// On every node in MPI_COMM_WORLD perform:
{
freeFilters( CHAR* freeMessage );
sendResultsToResultAPI();
MPI_Finish();
}
```