version 0.7
March 27th, 2004

L2 Processor Control and Interfaces

Kristian Hahn, Joe Kroll, Paul Keener, Chris Neu, Fritz Stabena
Rick Van Berg, Peter Wittich, Daniel Whiteson
University of Pennsylvania

Abstract

The L2 processing nodes must process events as rapidly as possible, while responding to control commands from the data aquisition system and providing monitoring information to verify performance and aid commissioning. A description of the architecture of the L2 processor system is given and the interfaces between the components is specified.

1 Introduction

The Level 2 trigger upgrade for RunIIb features a small set of commercial CPUs for making trigger decisions and a new interface to the event data in the front-end crates of detector subsystems. The interface to the front-end crates is built in a modular fashion out of PULSAR[**?**] boards, which collect and feed the data to the L2 nodes for event processing.

This note describes the architecture of the L2 nodes and their interface with the data stream, the data acquisition system control software and L2 monitoring software. Section 2 details the requirements placed on the system. Section 3 describes the architecture at the highest level, including the separation of tasks between the nodes and their supervisor, the Trigger Evaluation Director (TED). In Section 4, the control flow and communication of TED are described; Section 5 covers the same topics with respect to the nodes. Section 6 details the interfaces of the common objects passed between the components. Section 7 discusses the intricacies of locking shared data objects.

2 Requirements

The L2 processing system must meet the following requirements with respect to monitoring and control:

2.1 L2 Processor Requirements

- monitoring should not be intrusive
- errors should be reported quickly

2.2 Run Control Requirements

The system should respond to the following commands from Run Control

- Partition
  The system will join the specified partition and listen only to commands from that partition until released. Acknowledgement or error report required.
- Configure
  The system will receive configuration information from RunControl, in the form of information from the Run, Trigger, or Hardware Databases. It will configure the nodes for event processing and respond with success or failure. Acknowledgement or error report required.
- Activate or Run
  The system will prepare for data-taking. Acknowledgement or error report required.

- Halt
  The system will halt processing events. Acknowledgement or error report required.
- Recover
  The system will flush events from the system and reset all queues. Acknowledgement or error report required.
- End
  The system will cease processing events. Acknowledgement or error report required.
- Reset
  The system is released from the specified partition. Acknowledgement or error report required.

2.3 Monitoring Requirements

The system will maintain and provide access to the following monitoring information, both on demand and on a regular basis.

For each node:

- CPU and memory usage for algorithm and IO
- Per trigger and total algorithm time
- Per trigger and total unpacking time
- Interrupt counts
- Ethernet statistics
- Disk usage
- Event size
- Number of trigger objects
- CPU and device IDs
- uptime

For the system:

- L1 Accept rate per trigger
- L2 Accept rate per trigger
- Number of filled buffers

3 System Architecture

3.1 T.E.D.

The individual processing nodes are isolated from the interface to the data acquisition system and from monitoring infrastructure by the Trigger Evaluator Director (TED), see Figure 1;

The functions of TED are separated into pieces which divide the task into the overall control and interface to external components:

- **Run Control Client**: The RCC will communicate with Run Control, translating its requests into appropriate L2 actions, and indicating the status of the L2 nodes.
- **L2 Node Controller**: The L2NC will issue commands to the L2 nodes, including requests for configuration, changes to processing modes and requests for monitoring data. Ine L2NC will handle communication with exactly one node.
- **L2 Node Monitor**: The L2NM will recieve monitoring data from the L2 Nodes. One L2NM will handle communication with exactly one node.
- **L2 Monitor Server**: The L2MS will allow access via a Monitoring GUI to the gathered monitoring information.
- **TED Control**: The TED Controller will own the above interfaces, and determine what actions to take in response to commands from external components.

3.2 L2 Processors

Figure 2 depicts the division of tasks on the L2 processor nodes. Components shown in the diagram are described below.

- **NodeAlgo :** This process deals with the evaluation of the trigger algorithms as well as S-LINK I/O.
- **NodeMonitor :** The NM gathers hardware information from the system and monitoring data from NodeAlgo, packages it and sends it to T.E.D.
- **NodeController :** The NC receives configuration, processing and monitoring commands from the L2NC on T.E.D. The interperted commands guide NC's control of NodeAlgo and NodeMonitor.

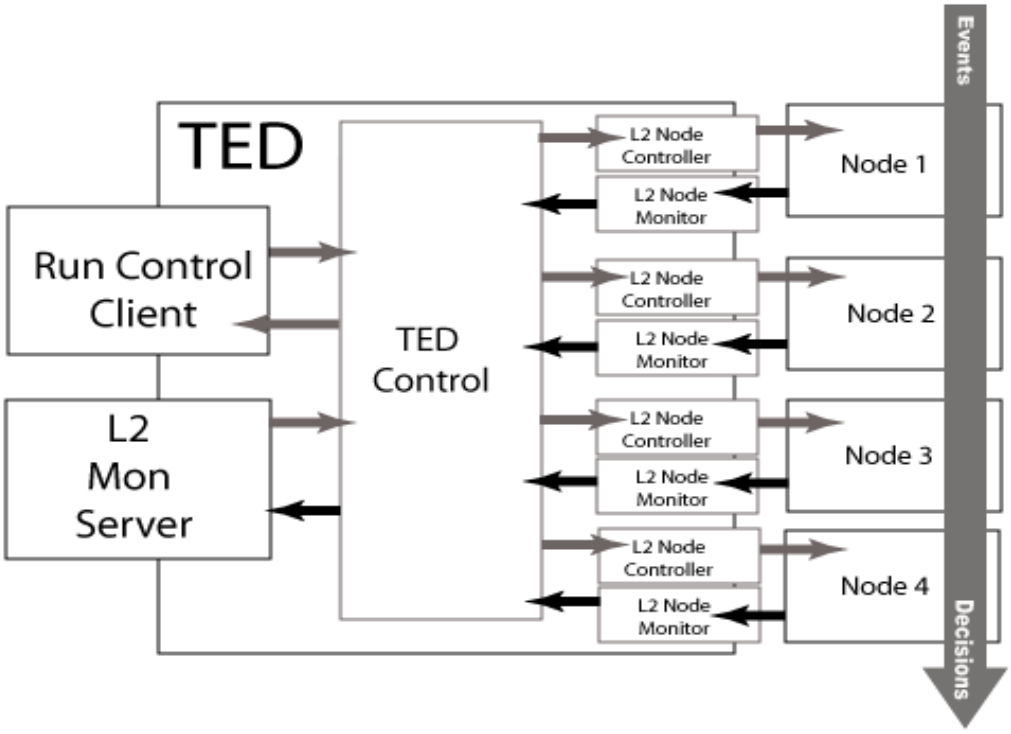

Figure 1: Communcation between pieces of the T.E.D, and their communication with outside components. Grey arrows indicate synchronous control which waits for an indication of success or failure; black arrows indicate asynchronous data flow without blocking for a response.

4 T.E.D. Control Flow

T.E.D. is driven by its response to external asynchronous events; it acts as a central communication point, translating requests from one piece of the system into actions by other pieces.

The pieces of the system which interact with external components (Interface Components) operate asynchronously; external events may occur at any time and at any rate. The TED Control is a fairly simple object, which provides a set of routines which the Interface Components may call when an external event is detected. In this way, these routines provide an interface by which incoming messages could be translated into actions; it defines the response of the system to messages from external components.

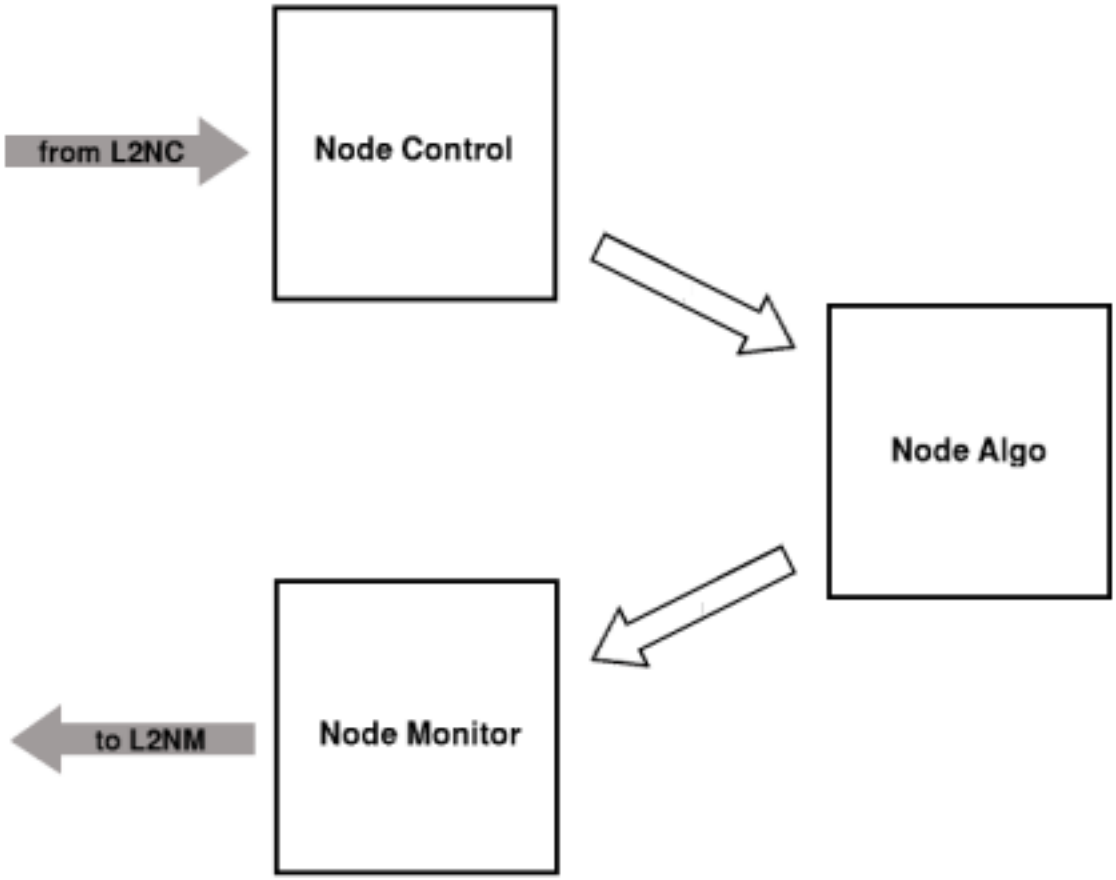

Figure 2: **L2 node Flow Control** This figure depicts the funcional blocks that operate in the L2 nodes. Filled grey arrows imply external communication with T.E.D. The single-headed arrows shown point from initiator to target but some information, such as acknowlgements, will also flow in the counter direction.

4.1 Synchronous Control

RunControl Client

The RunControl Client listens for instructions from Run Control; when one arrives, the RCC is required to communicate its success or failure to RunControl. During the execution of the command, no further RC commands will be seen. Therefore, the RCC client may execute these commands synchronously, waiting to report their completion and status before listening for new commands. The RCC's interface to the TED Control will be as follows:

- `bool TEDControl::Configure(ConfigSpec);`
  Send configuration information to the nodes. For example, ConfigurationData may contain the version of the trigger table and the set of prescales. Return value

indicates success or failure.

- `bool TEDControl::Activate();`
  Indicate to the nodes that they should be prepared for data to arrive. Return value indicates success or failure.

- `bool TEDControl::Run();`
  Prepare for data. Return value indicates success or failure.

- `bool TEDControl::Recover();`
  Flush the system of data and reset all queues. Return value indicates success or failure.

- `bool TEDControl::Halt();`
  Stop processing data. Return value indicates success or failure.

- `bool TEDControl::End();`
  The run has ended. Return value indicates success or failure.

L2 Node Controller

The L2 Node Controller issues instructions to the nodes; it waits to receive an acknowledgement from the nodes that the command has been successfully executed, though it does not retrieve any data nor wait for any further information. If data is sent to T.E.D. in response, it is passed via the L2 Node Monitor.

- `bool L2NodeController::Command(ControlObj &cObj)`
  Issues a control command to the node. The `ControlObj` contains a description of the command and the necessary data, if any. The possible commands include

  → `Configure`: Instructs the node to configure; return result indicates success or failer

  → `GoThirsty`: Instructs the node to prepare for data; return result indicates success or failer

  → `GoDrunk`: Instructs the node to ignore further data; return result indicates success or failer

  → `Flush`: Instructs the node to discard data and clear queues; return result indicates success or failer

  → `RequestMonitoringData`:
  Instructs the node to send updated monitoring data at its next opportunity; return result indicats success or failure.

  See Section 6 for details.

L2 Monitor Server

The L2 Monitor Server listens for requests for updated monitoring data from an external interface. It makes a request to the TED Control, waiting to hear an acknowledgement that the request was successfully transmitted. The monitoring data, however, is returned asynchronously, at the next available opportunity. The L2 Monitor Server makes use of the following interface to TED Control:

- `bool TEDControl::UpdateMonitoringData();`
  Request updated monitoring information from nodes at next opportunity.

4.2 Asynchronous Control

Monitoring and status information from the nodes passes to T.E.D. in an asynchronous manner; the nodes may report a processing error which requires resetting the system (issuing an HRR), or they may simply send monitoring statistics, a low priority task in comparison to data processing.

TED Control

The TED Control will pass monitoring data from the L2 Node Monitor to the L2 Monitoring Server when it arrives from the nodes:

- `bool TEDControl::StoreMonitoringData(NodeID, RunID, MonData);`
  Receive new monitoring data
- `bool L2MonitorServer::StoreMonitoringData(NodeID, RunID, MonData);`
  Receive new monitoring data

RunControl

If one of the nodes reports an error, seen via the L2MonitorServer, TED Control will indicate to the RCC to send an error to Run Control:

- `bool RunControlClient::L2Error(EventID, RunID);`
  Report an error

4.3 Communication

4.3.1 TED-Node

TED communicates with the nodes in two ways: it issues commands, which must be acknowledged, and it listens for monitoring data which arrives asyncronously. These two communications tasks are handled by the `L2NodeController` and `L2NodeMonitor` respectively. Though the two channels are distinct, they will share the basic communication mechanism, see Figure 3. Prior to sending, control objects and monitering data

will be serialized into a character buffer of specified length. Upon receipt, the buffer will be deserialized into the full object. The specification of the control object and the monitoring data object is described below. Each of the objects will be self-serializing, providing their own methods for conversion to and from a buffer.

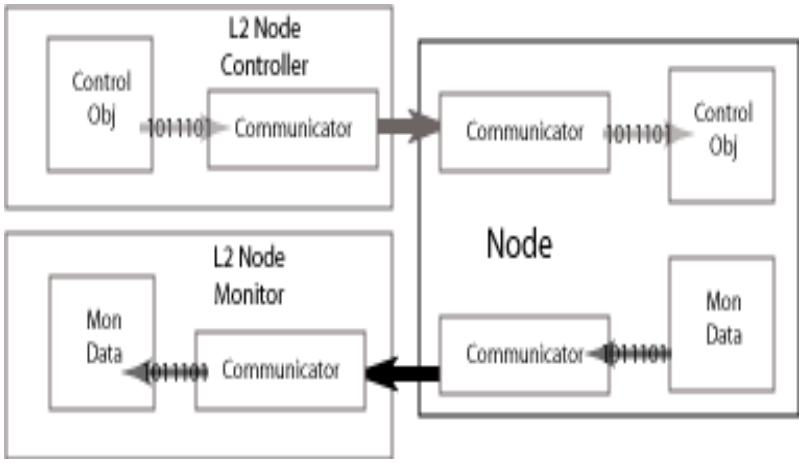

Figure 3: Communcation between the L2NodeController, L2NodeMonitor and the node. Control objects and monitoring data are serialized and then transferred via the Communicator before being deserialized by the receiver.

This leaves the task of communication of the buffer between TED and the node, or vice verse. We encapsulate this task, and isolate the choice of communication protocols into the `Communicator` object, which provides the following interface:

- `Communicator::Communicator(IPAddress add, Communicator::ChannelType)`
  This constructor opens a channel from the local computer to a remote computer at the specified address. The `ChannelType` may be `Communicator::CommandChannel` or `Communicator::MonitoringChannel`. This allows us to open two channels to a specific remote computer; inside the Communicator, one may assign different port numbers to the different channels depending on the precise implementation.

- `Communicator::Communicator(Communicator::ChannelType)`
  This constructor opens a channel from a remote computer of unspecified address which will contact the local computer; the `ChannelType` specifies the port number, for example, where the remote computer will connect.

- `bool Communicator::Send(char *buffer, int l)`
  Sends data to the remote computer. Return indicates success or failure.

- `bool Communicator::Receive(char *buffer, int &l)`
  Returns true when data has been recieved; in this case, the buffer and length are filled; otherwise their values are unspecified.

4.3.2 TED-RunControl

Communication between TED and RunControl is handled by the RunControlClient; this interface is fully specified by RunControl, and requires that the Client subscribe to messages which appear over ethernet in smartsockets packets.

5 Node Control Flow

The evaluation of L2 trigger algorithms is the central process on the nodes. The control and monitoring of this process is directed by T.E.D. through its L2 Node Controller and L2 Node Monitor interfaces. Corresponding interfaces exist on the nodes (NodeController & NodeMonitor) and establish a communication channel with T.E.D. NodeController delivers the commands it receives from T.E.D to NodeAlgo and NodeAlgo pushes monitoring information to NodeMonitor for delivery to T.E.D. NodeMonitor does not receive instructions from T.E.D directly. NodeAlgo instead receives from NodeController commands intended for NodeMonitor and encodes them in the monitoring information they share.

Communication between the *logical*processes on a node just described is implemented using shared data objects rather than interface functions, as is the case on T.E.D. A different approach is necessary because internal communications on the nodes are really interactions between *actual* Linux processes. While NodeController and NodeMonitor are logical processes (in that they do not necessarily represent distinct threads of execution), we stipulate that NodeAlgo be assigned an individual thread so that its task of performing the L2 algorithms can be given high priority. Internal communications on the nodes are limited to those between NodeController/NodeAlgo and NodeAlgo/NodeMonitor and, because NodeAlgo is implemented as an individual thread, these then are actually interactions between distinct threads. Common data structures provide a mechanism to share information between the threads' separate process spaces that best suits needs [1]

5.1 Asynchronous Operations

NodeController → NodeAlgo

The synchronous operation of the L2NC on T.E.D. permits NodeController to process L2NC's instructions serially. NodeController communicates with NodeAlgo through a shared instance of the ControlObj class, described in Section 6.1. This object contains a 'command' field that NodeAlgo accesses to determine the actions it should undertake. NodeAlgo calls the following functions in response to corresponding commands from

[1] Threads may also use signals to communicate however this mechanism is not amenable to the tranfser of more that just state information

T.E.D. NodeAlgo uses their returned values to set an status field in ControlObj, which NodeController sends as ackowledgement to the L2NC on T.E.D.

- **bool NodeAlgo::configure(AlgoConfigSpec)**
  Configuration data, such as prescales and trigger table information, is passed to NodeAlgo.
- **bool NodeAlgo::setThirsty()**
  NodeAlgo is told to prepare for the arrival of data.
- **bool NodeAlgo::setDrunk()**
  NodeAlgo is told to ignore further data.
- **bool NodeAlgo::flush()**
  NodeAlgo is told to discard the current event and clear any queued information.
- **bool NodeAlgo::configureMonitoring(MonConfigSpec)**
  Configuration data, such as sampling rate and ROI, is passed to NodeMonitor.
- **bool NodeAlgo::requestMonitoringData()**
  Request that monitoring data from NodeMonitor be sent right away.

NodeAlgo → NodeMonitor

NodeAlgo monitors aspects of the L2 trigger algorithms at a rate presrcibed by T.E.D. It uses an instance of the MonitorObj class to pass this information to NodeMonitor, which combines the algorithmic profiling data with the hardware monitoring statistics that it collects. In its normal operation, NodeMonitor constructs a local copy of the MonitorObj it receives from NodeAlgo before appending its own data. The shared MonitorObj is a common resource and a local copy allows NodeMonitor to avoid blocking NodeAlgo's access to it while NodeMonitor gathers hardware statistics. When NodeMonitor finishes updating its local MonitorObj the object is serialized and sent to the L2NM on T.E.D.

MonitorObj is also used to relay error conditions. NodeAlgo sets diagnostic fields in MonitorObj and flags it as containing error information when problems are encoutered in the processing of the L2 algorithms. When NodeMonitor finds that MonitorObj has error flags set it immeadiately serializes the MonitorObj and transmits it to T.E.D., foregoing the usual collection of hardware statistics (KH: or not ... maybe we want to send monitoring info along with the errors?).

- **bool NodeMonitor::isError(MonitorObj)**
  NodeMonitor makes a copy of the MonitorObj that has been partially filled with information from NodeAlgo, so as not .

- **bool NodeMonitor::copyMonitorObj(MonitorObj)**
  NodeMonitor makes a copy of the MonitorObj that has been partially filled with information from NodeAlgo, so as not .

- **bool NodeMonitor::acquireHWInfo()**
  Request monitoring infomation from NodeAlgo. The information arrives when it becomes available.

- **bool NodeMonitor::appendHWInfo(MonitorObj)**
  Request monitoring infomation from NodeAlgo. The information arrives when it becomes available.

5.2 Communication

NodeController and NodeMonitor use the communication mechanism described in Section 4.3 to communicate with T.E.D. NodeController incorporates a Communicator object that receives serialized ControlObjs and returns command acknowledgements to the L2NC. NodeMonitor also includes a Communicator, used to send serialized MonitorObj's to and to receive acknowledgement from the L2NM on T.E.D.

6 Common Objects Definitions

Various components of TED and the processing nodes manipulate and transmit control and monitoring information. We define the form of two common objects, one which describes a command, and another which contains monitoring data.

6.1 Control Objects

The ControlObj class represents the commands that TED will issue to the nodes.

6.1.1 ControlObj Interface

There are several aspects common to each of the commands that NodeAlgo will process; a buffer number, the command type, its sequence number and the current state of NodeAlgo's progress in completing the command. These are represented by corresponding public fields in ControlObj.

- `public command_type ControlObj::l2buff`
- `public command_type ControlObj::command`
- `public state_type ControlObj::state`
- `public int ControlObj::seqnum`

Some commands, such as 'Configure', may require that additional information be passed to NodeAlgo. ControlObj therefore contains a private array of auxillary data structures with entries for each of the individual commands. A public pointer within the object will be made to point to the entry that corresponds to the current command. The data structures inherit from a virtual ControlData class which specifies the internal I/O interface to the structures.

- `public ControlData* ControlObj::auxptr`
- `private std::vect<ControlData> ControlObj::auxvec`

ControlObj has the capacity to serialize itself to a delimited ascii string that L2NC can transmit to the nodes. ControlObjs are reconstituted in NodeController by deserializing received strings. These operations are accomplished using ControlObj's toBuff/fromBuff methods.

- `public bool ControlObj::fromBuf( int length, char* buff )`
- `public int ControlObj::toBuff( char* buff )`

6.2 Monitoring Data Objects

The MonitorObj class contains monitoring or error data from a specific node.

6.2.1 MonitorObj interface

MonitorObj contains the following fields which respectively indicate the types of information the object contains, the node's assigned buffer number, this objects sequence number and it's transmission state.

- `public command_type MonitorObj::_type`
- `public int MonitorObj::_l2buff`
- `public int MonitorObj::_seqnum`
- `public state_type MonitorObj::state`

MonitorObj contains member objects to hold algorithmic, hardware and error information. It provides accessors to these objects (AlgoData, HardwareData and ErrorData), which in turn provide accessors to the data they contain. The pointers returned from MonitorObj's accessors will equal NULL if it does not contain the particular data object.

- `AlgoData* MonitorObj::getAlgoData()`
  Total algorithm time

- `HardwareData* MonitorObj::getHardwareData()`
  total unpack time

- `ErrorData* MonitorObj::getError()`
  the L2 buffer number

The data objects derive from a MonitorData parent class which specifies their serialization interface. These are called from within MonitorObj's `toBuff` and `fromBuff` to serialize and deserialization the complete object.

- `public bool MonitorObj::fromBuf( int length, char* buff )`

- `public int MonitorObj::toBuff( char* buff )`

6.2.2 MonitorData interface

describe the acessors to the monitoring data.

7 Data Synchronization on the L2 Processors

7.1 Motivation

The NodeAlgo process on the L2 nodes will generate algorithm monitoring data that is read by the NodeMonitor logical processes. NodeMonitor packages this data and transmits it to the L2NM interface on T.E.D., as described in section 5. NodeAlgo will pass monitoring infomration to NodeMonitor through shared memory segments. Access to these shared regions must be synchronized so as to ensure the integrity of the data contained within. As mentioned in section 2.1, the process of monitoring the algorithms is required to introduce only minimal overhead to the operation of NodeAlgo. The mechanism used to establish data synchronization between NodeAlgo and NodeMonitor must also respect this constraint.

NodeAlgo and ModeMonitor represent logical tasks, *ie:* tasks that do not necessarily correspond directly with heavy-weight Linux processes or threads of execution. The function of NodeMonitor, for instance, may be performed by a process that also performs other tasks, such as that of NodeControl. The distinction between logical and actual processes is important for NodeMonitor but less so for NodeAlgo. It is clear that the high-priority operations of NodeAlgo are best handled by a thread(s) that is not encumbered by any other task. In the text that follows we describe a model for the synchronization of memory accesses between the two threads that implement NodeAlgo and NodeMonitor but the reader should bear in mind that the NodeMonitor thread may also be charged with additional responsibilities.

7.2 Reader & Writer Mutexes

Our preferred approach to the synchronization of data between NodeAlgo and NodeMonitor involves posix mutexes (mutual exclusion devices). Mutexes have two states, locked and unlocked, and can be used to allow single agent access to a protected resource. Locking and unlocking a mutex are "atomic" operations and guaruntee that a thread which has locked a mutex is the only thread to have done so. A more complete description of posix threads and mutexes may be found in Nichols, *et. al* [1].

Access to shared memory resources will be synchronized with two mutexes, as shown in Figure 4. In the course of performing the L2 trigger algorithms NodeAlgo will collect information on events at a specified rate. When NodeAlgo completes the collection of information for a given event it will lock the writer mutex (A) before writing the information to memory (B). The lock will temporarily prevent future access to the shared memory region by NodeAlgo while the already locked reader mutex prevents accesses by NodeMonitor. NodeAlgo finishes writing data and unlocks the reader mutex (C), permitting NodeMonitor to access the data. NodeMonitor is now able to lock the reader mutex (D) and read the monitoring data (E). When finished, NodeMonitor unlocks the writer mutex (F), granting NodeAlgo access to shared memory, and the process begins anew (G).

Attempts by either thread to lock a mutex that it already holds will fail, indicating that the companion thread is accessing the shared data. NodeAlgo will try to obtain a lock on the writer mutex once before it goes on to process additional events. It will attempt to look the mutex again after completeing some specified number of events. NodeMonitor's response to locking faliure remains as an implementation decision.

7.3 Timing

Meausurements of the time required to lock and unlock a mutex on a high-priority thread yield 0.17μs and 0.18μs, respectively. We performed tests using two mutexes and threads as described above. Details of the measurements are provided in a web page [2].

References

[1] the O'Reilley book

[2] http://www-cdf.fnal.gov/ krisitian/pulsar/sync.html

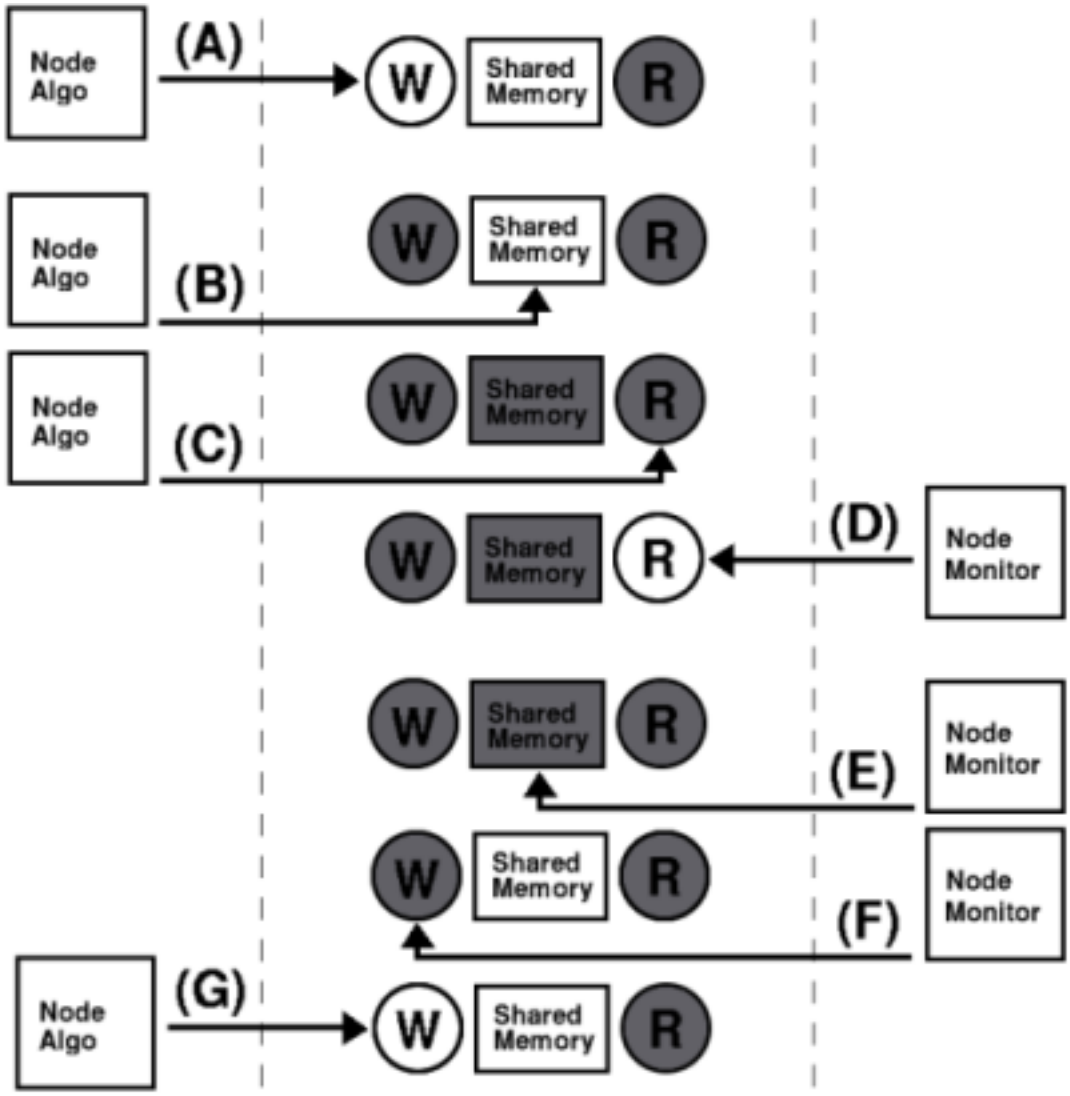

Figure 4: Access to memory shared between the NodeAlgo and NodeMonitor logical processes is synchronized using reader and writer mutexes, depicted here as the circles labeled "R" & "W". Locked mutxes and memory containing valid data are filled with gray. In the process shown NodeAlgo locks the writer mutex, writes monitoring information to shared memory and unlocks the reader mutex. NodeMonitor then locks the reader mutex, reads the data and unlocks the writer mutex, permitting NodeAlgo to access shared memory again.