

# A Monitor for Distributed Systems

## Course Project for IT-630

J. Ramanand (05329402)  
Roshan Kulkarni (05329401)

May 2nd 2006

### **Abstract**

A Monitor for a Distributed System is a process that receives messages from nodes in that distributed system. From these messages, it can reconstruct process histories and potentially provide useful visualisations and inferences. This course project constructed such a Monitor framework for distributed applications. The project also implemented the Asynchronous LCR Algorithm for a ring as the distributed system to be monitored. Nodes and links are represented by processes, vector clocks are associated with the nodes and help label events. Each process additionally logs its events to the Monitor process which reconstructs a representation of the system. This report describes the design, architecture and issues encountered in this project.

## **1 Introduction**

### **1.1 Monitor**

Processes in a distributed system interact heavily with each other and communication usually is in the form of messages exchanged between processes. Processes also take significant internal actions. The theory of the “happened-before” relation along with constructs like logical clocks help impose a partial order on these events and understand inter-dependencies among processes.

The concept of a “monitor” is used to observe the distributed system as a whole. Processes in a system have local knowledge coupled with some global information obtained from neighbouring processes. Observing such a system can be difficult, and monitors can provide visualisations on the system. They can also help in arriving at consistent cuts of the system and evaluating global predicates.

In this project, the scope of a monitor was restricted to obtaining messages from processes and constructing the sequence of actions that the process has been involved in. The monitor then correlates a “Send” event from a process with its corresponding “Receive” event. It detects failed “Send”

messages and incorrectly logged "Receive" events if any were present. A colour coding is used in the visualisation to indicate these valid and invalid events.

## 1.2 Leader Election

Leader Election is a well-known distributed computation that involves the election of a node from among the process in the system as a "leader" on the basis of some distinguishing characteristic such as process identifier. From the several well-known leader election algorithms, this project implemented the asynchronous LCR leader election algorithm

## 2 Key Goals and Assumptions

### 2.1 Goals

1. The project sought to implement a "Process Model" for a distributed system. In order to do this, two kinds of processes were defined, one to represent a node and another to simulate a link. A process, rather than a direct IPC mechanism, was chosen so that in the future, the link could be associated with parameters such as probability of dropping messages and latency of message delivery.
2. The project aimed to implement practically some of the basic distributed system algorithms such as Leader Election and Vector Clocks and to understand how to translate the theoretical concepts to a design and implementation. An important sub-goal was to study the integration of these concepts (such as the increment of Vector Clocks along with actions during the LCR algorithm) studied so far in isolation.
3. A general framework for design of the Monitor, communication with the monitor, message design and necessary data structures was to be created
4. Lastly, the monitor was required to produce a snapshot of the entire system as constructed from the messages obtained so far from the processes. This took the form of an HTML file displaying the system as a deposit.

### 2.2 Assumptions

1. The processes are arranged in a ring topology
2. The links are unidirectional
3. There are no link failures

4. Processes log all events and send them to the monitor without error
5. Links guarantee FIFO message ordering
6. The monitor doesn't not have an entry in the vector clock
7. Messages sent to the monitor do not cause the vector clock of the process to be incremented
8. The number of processes in the system is known to all processes and the monitor

### 3 System Components

Below, we describe the components in the system.

#### 3.1 Process Node

This represents a single processing element in the distributed system. A system comprises of several such process nodes. The process node can be connected to other nodes, with a set of *ProcessLinks*. In our specific scenario, we have a set of process nodes arranged in a unidirectional ring topology.

Fig-2 represents the components within a process node, described below:

##### 3.1.1 Receiver Thread

This thread binds to a in-bound port of this node and receives incomming connections from other nodes. It is responsible to read inbound messages and put them in a *IncommingQueue*. This incomming queue has a FIFO behaviour. The Processor thread eventually reads messages from the queue and processes them.

##### 3.1.2 Processor Thread

This thread removes messages from the incomming queue and hands them to the *operation()* method of a sub-class implementation of the Processor Thread. The sub-class *operation()* method can implement the functionality of any distributed algorithm - like the LCR Leader election. Outbound messages are put by this process to the end of the *OutputQueue*.

##### 3.1.3 Sender Thread

This thread reads messages at the head of the output queue and sends them, using a socket connection, to the subsequent link. This operation is reliable in nature. The process node retries the operation until the message is successfully sent to the adjacent link.

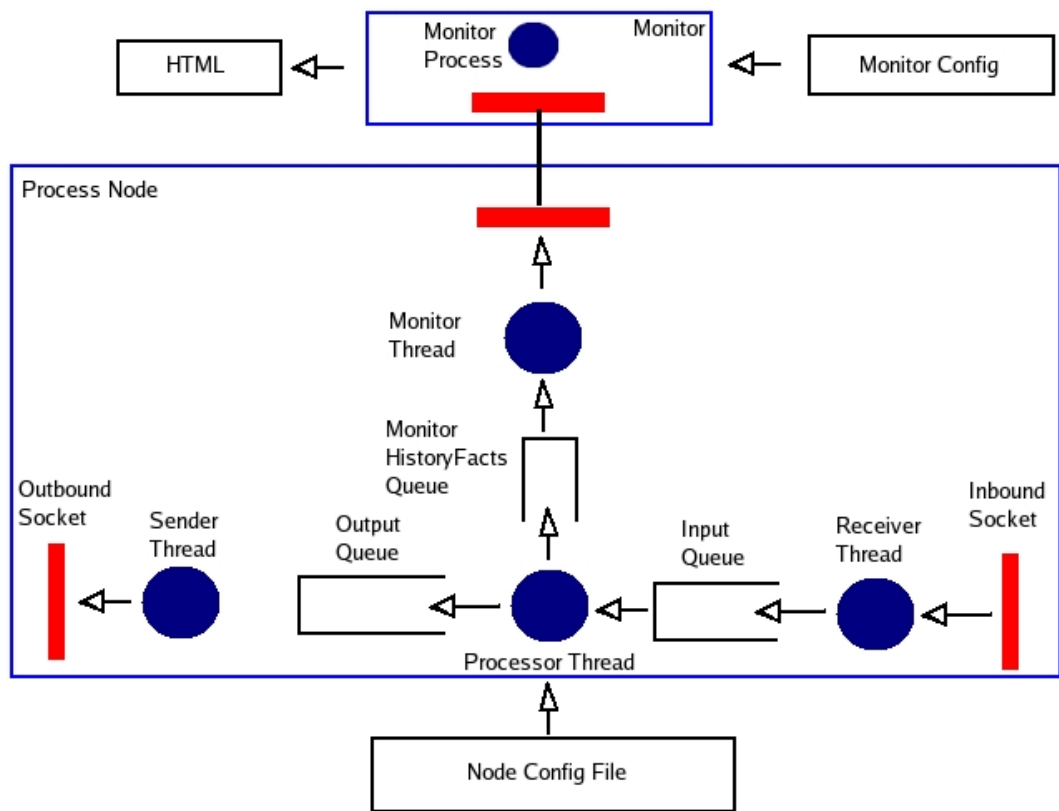


Figure 1: Process Node

### 3.1.4 Monitor Thread

The Processor Thread logs the events in the system - Send, Receive, Internal - to the Monitor queue. The monitor thread, picks up messages from the head of the queue and sends them over to the monitor node over a socket connection. The monitor thread waits until the size of the message queue is greater than some threshold value or until the time elapsed, since the last log message sent, is greater than some threshold value. In this case, the monitor thread will flush out the entire monitor queue and send it to the monitor node. This queueing is done for the purposes of optimization.

### 3.2 Process Link

The link process represents a link between two *ProcessNodes*. It is aware of the PIDs of the two process nodes that it is connected to. It binds to a port listening for in-bound messages. Each message is tagged with the PIDs of the source and destination nodes. Depending on the PID of the

destination, the link forwards the message, over another socket connection, to the destination node. The link however is unreliable. It will drop the message, in case the receiving process does not accept the message, in the first attempt.

### 3.3 Messages

The *Message* class defines a serializable base class containing the source and destination PIDs and a vector clock. Specific message classes inherit from this base class. We have the following types of messages in the system at present:

1. LCR Message: Message used in the LCR algorithm, to send the larger PID to the next neighbour.
2. Leader-Chosen Message: Message used in the LCR algorithm, to indicate the PID of the chosen leader.
3. Monitor Log Message: Message used to send a set of History Facts to the Monitor Log.

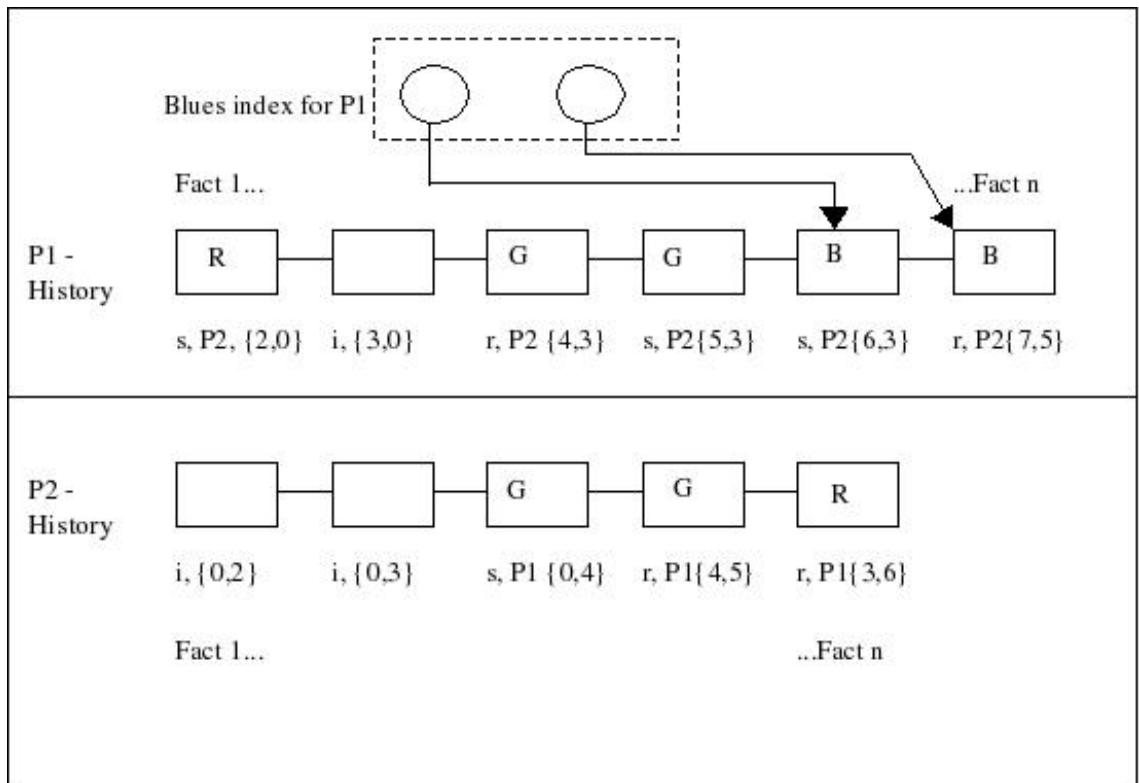
### 3.4 Monitor

The Monitor is a single process that receives messages from processes describing their events. For each process, the monitor constructs and maintains a "history". A history is a collection of "facts" that contain information about the different events involving the monitor. The events are of three types: SEND, RECEIVE and INTERNAL. So, instead of maintaining states, the history contains of events that caused state transitions in a process's happened before diagram.

On performing a send, receive or any significant internal action, a process updates its vector clock. Additionally, a new "fact" is generated containing the pid of the process, the vector clock associated with the event, the corresponding process in case of a send or received (called the "brother" process), and the type of the event. Such facts are accumulated and sent in a bunch to the monitor. On receiving these new facts, the monitor adds them to the history corresponding to the sending process.

Each fact is associated with a colour as follows:

- Black: For an internal action
- Blue: For a send or receive whose corresponding receive or send has not been reported yet in the brother process
- Green: For a send or receive whose corresponding receive or send has been reported



Monitor: Examples of histories, facts, colours for facts, Blues index

Figure 2: Monitor Internals

- Red: For a failed send or invalid receive message

A blue fact can turn green or red and remains stable after changing its colour.

When a new fact of type "SEND" arrives, the history of the brother process is visited to attempt to find the complementary "RECEIVE". If found, both facts are marked Green, otherwise the new fact is set to Blue. Similarly, if the new fact was a "RECEIVE", the brother history is visited to locate a possible "SEND" match. After each new batch of facts has been processed, the blue facts in each history are checked to see if a inference of permanent failure can be made. For a blue "SEND" message, this occurs if another message from its parent process was sent to the same brother process successfully in the future. By the FIFO assumption, it can be claimed that

this "SEND" message is lost. Hence the fact is marked Red. Similarly, a blue "RECEIVE" message can be marked Red if the state corresponding to the sending of the message received (obtained using the vector clock) has occurred in the brother process but did not indicate any such send. The receive is then thought to be erroneous and marked Red.

To optimise lookups on the histories, an index of blue facts is maintained for each history. Since for each new send or receive message, a corresponding blue fact in the brother process is sought, only these blue facts can be visited to check for pairs. Additionally, since only blue facts can turn red, the non-blue facts can be bypassed using the index. When a fact turns red or green, it is removed from the index.

Visualisation is achieved by iterating over the histories and generating states and actions, suitably coloured, providing a user-friendly deposit view. Examples of these are available in the appendix of this report.

## 4 Issues, Design Decisions

In this section we describe some of the issues we faced during the development of the system. Also, we describe some of the design decisions we have taken.

- **Single vs Multi Threaded Process Nodes:** Using a single threaded process node, does not correctly emulate the behaviour of an asynchronous system model. The single thread blocks on send/receive operations and the system eventually behaves in synchrony. Moreover it does not correctly represent fairness of the various tasks in the node.
- **Vector Clock Data Structure:** Most theoretical implementations identify the vector clock as an array of integer values. Our current system also implements the vector clock as an array. However, in scenarios where the number of nodes in the system is not known, an array implementation of the vector clock will not work. It would hence be desired to implement vector clock as a hash-map of (PID, Clock) values. This makes the clock update-algorithm more complex as compared to the simple array implementation.
- **Incrementing Vector Clock:** The operations to increment the vector clock on a send/receive have to be implemented in the correct order so that happened-before relationships are not violated. Merely implementing the vector clock operations in the send/receive threads, disregarding the ordering wrt the internal state changes, would result in incorrect vector clock values associated with send/receive operations.

## 5 Future Work

The following extensions are proposed on the foundation of the work done in this project:

1. Provide an extensive failure model for the links. This can be achieved by adding parameters to the link processes controlling whether messages are dropped on a probabilistic basis and to control the latency of message delivery. Links can also be made unavailable occasionally.
2. Useful inferences about consistent cuts and global/conjunctive predicates can be made based on the collected information
3. The monitor's output can be extended to indicate more information about cuts etc.

## 6 Conclusion

This course project implemented a distributed system consisting of processes engaged in asynchronous LCR Leader Election in a unidirectional ring along with processes acting as links. A multithreaded implementation of the process was created. The system implemented vector clocks along with this computation. A monitor process was constructed that could recreate the history for each process as reported. It then used this information to make inferences about valid and invalid send and receive events. Some optimisations on maintaining the histories were also implemented.

## A Appendix: Screen Shots

The screen shot 3, is the output of the debugger for a run of the LCR with four nodes. The output identifies the set of state transitions and the events in the system. Each event is represented within curly braces and is of type: Send, Receive or Internal. Send/Receive events are colored: Blue, Green and Red. A green color for a event indicates that a complementary receive/send has been also logged for this send/receive event. A red color indicates that the complementary send/receive has not been received at all. Blue color indicates that the complementary log message has not yet been received. However, there is still a possibility of this message arriving in the future.

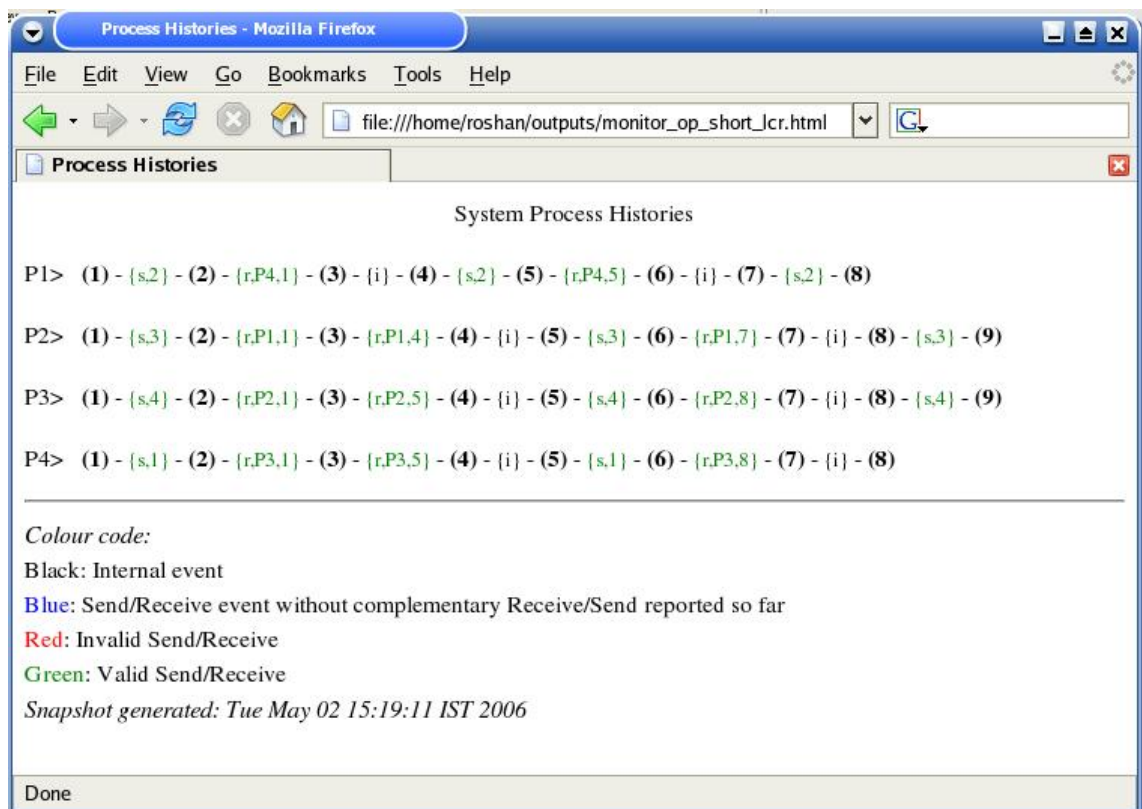


Figure 3: Debugger Output: LCR Algorithm with four nodes