MONITORING AND STEERING OF DISTRIBUTED SYSTEMS USING LAW GOVERNED INTERACTION (LGI)

BY FAROKH K. VESUNA

A thesis submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Master of Science
Graduate Program in Computer Science
Written under the direction of
Professor Naftaly Minsky
and approved by

New Brunswick, New Jersey
January, 2000
ABSTRACT OF THE THESIS

MONITORING AND STEERING OF DISTRIBUTED SYSTEMS USING LAW GOVERNED INTERACTION (LGI)

by Farokh K. Vesuna

Thesis Director: Professor Naftaly Minsky

With the increasing use of heterogeneous distributed systems, efficient monitoring and steering mechanisms have become essential tools for improving the performance and reliability of these systems. In this view, a monitoring and steering tool should have the following capabilities: (1) It should be useful for a general application domain, (2) its implementation should be easy since the system is heterogeneous and distributed, (3) it should be able to selectively express and process events to be monitored so as to be able to focus on specific events, (4) steering should have the possibility of being executed in a controlled manner and, (5) an adaptable visualization of the monitored system should be provided. The model described in this paper has all the above capabilities. It has been implemented for monitoring and steering distributed heterogeneous applications that use Law Governed Interaction (LGI) for communicating with each other. The model itself uses LGI for achieving all the above mentioned capabilities except for the last one.
Acknowledgements

I’d like to thank my advisor Professor Naftaly Minsky for his time and support in the completion of this thesis.

I’d also like to thank Victoria Ungureanu, Xuhui Ao and Thu Nguyen, whose efforts have made the completion of this thesis possible.
Table of Contents

Abstract .................................................. ii
Acknowledgements ......................................... iii
List of Figures .............................................. vi

1. INTRODUCTION ........................................... 1

2. THE ARCHITECTURE ....................................... 5

3. THE VISUALIZER .......................................... 14
   3.1. Draw Commands ......................................... 15
   3.2. User Interface .......................................... 20

4. THE MONITOR ............................................. 28
   4.1. User Interface .......................................... 29

5. EXAMPLES ................................................ 33
   5.1. General Example ........................................ 33
   5.2. Flow Control Example ................................. 33
      5.2.1. The Secretary Law $L_S$ .......................... 38
      5.2.2. The Client Server Law $L_G$ ...................... 38
      5.2.3. The Monitor Law $L_M$ .......................... 39

6. RELATED WORK .......................................... 55

7. CONCLUSION ............................................... 57
## List of Figures

2.1. Snapshot of Visualizer .................................................. 7  
2.2. Example of a Rule in the Monitor Law \( \mathcal{L}_M \) ..................... 10  
2.3. Monitoring and Steering Architecture .................................. 11  
3.1. Attributes Dialog Box ................................................ 23  
3.2. Default Node Options Dialog Box ................................... 25  
3.3. Default Edge Options Dialog Box .................................. 26  
3.4. Send Message to GenericMonitor Dialog Box ....................... 26  
4.1. GenericMonitor Window .............................................. 28  
5.1. A Generic Group Law \( \mathcal{L}_G \) .................................... 34  
5.2. A Generic Monitoring Law \( \mathcal{L}_M \) .............................. 35  
5.3. Snapshot of Client Server Monitoring Example ..................... 37  
5.4. Rules of the law \( \mathcal{L}_S \) dealing with the Secretary ............... 38  
5.5. Rules of the law \( \mathcal{L}_G \) dealing with the Client Server communication ................. 41  
5.6. Rules of the law \( \mathcal{L}_G \) dealing with the Client Server communication (Continued) ............................................. 42  
5.7. Rules of the law \( \mathcal{L}_G \) dealing with the Client Server communication (Continued) ............................................. 43  
5.8. Rules of the law \( \mathcal{L}_G \) dealing with the Client Server communication (Continued) ............................................. 44  
5.9. Rules of the law \( \mathcal{L}_G \) dealing with the Client Server communication (Continued) ............................................. 45  
5.10. Rules of the law \( \mathcal{L}_G \) dealing with the Client Server communication (Continued) ............................................. 46
5.11. Rules of the law $L_M$ dealing with the Monitoring and Steering of a Group 47
5.12. Rules of the law $L_M$ dealing with the Monitoring and Steering of a Group
(Continued) ......................................................... 48
5.13. Rules of the law $L_M$ dealing with the Monitoring and Steering of a Group
(Continued) ......................................................... 49
(Continued) ......................................................... 50
5.15. Rules of the law $L_M$ dealing with the Monitoring and Steering of a Group
(Continued) ......................................................... 51
5.16. Rules of the law $L_M$ dealing with the Monitoring and Steering of a Group
(Continued) ......................................................... 52
5.17. Rules of the law $L_M$ dealing with the Monitoring and Steering of a Group
(Continued) ......................................................... 53
5.18. Rules of the law $L_M$ dealing with the Monitoring and Steering of a Group
(Continued) ......................................................... 54
Chapter 1

INTRODUCTION

With the increasing use of heterogeneous distributed systems, efficient monitoring and steering mechanisms have become essential tools for improving the performance and reliability of these systems. In such systems, there is a large amount of interaction between heterogeneous subjects. It is important to be able to monitor the subjects without perturbing the subjects operations too much, or knowing much about the subjects implementation. Steering is useful to provide some control over the behavior of the heterogeneous subjects.

Consider a client server situation. The clients send requests to the servers, which reply back to the clients. Both the clients and the servers have a protocol for communicating with each other. Now also consider a manager who would like to supervise (monitor and steer) the interactions between the clients and servers. For instance, the manager might intervene if a client is monopolizing a particular server. In this case the manager might have the client slow down its request rate. Given a single application it is possible to perform such monitoring and steering by explicitly adding the required features in the implementation. However, in a heterogeneous distributed system where the manager might not have knowledge about the client or server implementations, these solutions are not practical. Hence, what is required is a general model to monitor and steer the interactions between heterogeneous subjects without much perturbation in their implementations. Due to the heterogeneity, some subjects might also be hostile and hence the monitoring needs to be as transparent as possible. Our model attempts to provide a monitoring and steering architecture which explicitly addresses the challenges and requirements associated with managing the interactions among such
heterogeneous distributed systems.

The goal of our model is to monitor and steer the interactions between a group of heterogeneous distributed subjects. In order for such subjects to be able to communicate with each other (exchange messages), they need to have some common ground. **Law Governed Interaction** (henceforth referred to as LGI) [6] provides this common ground in our model. In LGI the subjects interact with each other using a prescribed set of rules called a **law**.

Monitoring of distributed systems typically, involves mechanisms of putting sensors in the subjects that need to be monitored. The purpose of the sensors is to collect the basic data which is to be monitored. This basic data is generally called a **primitive event**. The sensors send the primitive events to a process which then reasons about the primitive events and triggers some steering action or some virtual events (combination of primitive events).

Since the subjects are heterogeneous, putting sensors in the subjects wherever the primitive events need to be collected is not feasible. As we are interested in the interactions of the subjects with each other, and since the medium of communication is LGI, the LGI mechanism would provide the primitive events. Thus there is no explicit insertion of sensors in the subjects, since the subjects will be using LGI for communicating with each other. In this way the goal of monitoring with minimum perturbation of the subjects is achieved. Further steering is mainly achieved by modifying the state of the subject in the LGI mechanism since we are concerned only in the interactions between the subjects. In this way steering is achieved with out the need for steering hooks in the subjects.

Therefore in our context, the primitive event is the data that is sent by the LGI mechanism in response to a ruling for a subject. This primitive event would typically contain the message that the subject is sending and the state of the subject. The exact amount and type of information that is present in the primitive event would depend upon the rule in the subject’s law $\mathcal{L}_G$ that is triggered. Thus the entire state of the subject need not be sent, only selective portions of it may be sent. This would depend
on the application. In fact in our model, the monitor itself is an agent which uses LGI for communication purposes. This means that the LGI mechanism at the monitors side, receives the primitive events and the law of the monitor $L_M$ contains the rules for processing the primitive events. Processing the primitive events results in messages being delivered to the monitor. The monitor acts like a collector for these messages. Since large amounts of textual and numerical data are not easy to comprehend, the monitor sends these messages to one or more visualizers which provide for a graph based representation of the state of the monitored system. Thus the messages sent to the monitor from the law $L_M$ are not just any messages, but draw commands that instruct the visualizer what to do. Actually the visualizers are also agents in the LGI mechanism. This allows for the visualizers to act as actuators for steering purposes; the visualizer’s user interface allows for manual steering. Since the steering actions are messages, they are sent through the LGI mechanism. Automatic steering can also be performed in the monitor law $L_M$ at the time when primitive events are received at the monitors end. Control over steering is achieved via the LGI mechanism. The monitor law $L_M$ must allow for steering messages to be sent and the subject’s law $L_G$ must allow for such messages to be received. Thus, steering is performed in a controlled manner and the visualizers do not have unrestricted control over the monitored subjects.

To summarize, our model has the following features:

1. General and Customizable

   It is a general model that can be used for any application using LGI. The visualizer provides for a very simple programmatic interface for displaying application specific data.

2. Non-obtrusive Instrumentation

   It is possible to monitor and steer heterogeneous subjects with minimum perturbation to the subjects, since no explicit sensors are required in the subjects. Hence, it is also possible to monitor and steer the subjects in an almost transparent manner.
3. Local Event Filtering

Since the primitive events are specified in the laws used for communication, only selective information based on the application need to be sent to the monitor. The selection of information is done locally at the subjects end. In the context of online monitoring this would help reduce the amount of primitive events needed to be communicated and processed at the monitors end. Local filtering makes the model scalable in terms of the number of subjects which can be monitored.

4. Controlled Steering

Steering can be provided automatically, or manually. Steering is also performed in a controlled manner so that the monitor or visualizers do not have unrestricted control over the monitored subjects.

In view of monitoring and steering heterogeneous subjects of a distributed system our model has two key contributions:

1. Instrumentation can be achieved easily without having to place explicit sensors in the subjects. This eliminates the need to have detailed knowledge about the subjects’ implementation and hence our model is easy to deploy.

2. Since our model is dealing with heterogeneous subjects, of which little may be known, steering needs to be performed in a controlled manner, so that an attempt to steer a subject does not result in a breakdown or illogical operation of the subject.

This paper describes the general architecture and implementation of our model that uses the experimental group communication system called Moses (for detailed information on LGI and Moses, please refer to [6] from the References). The rest of this paper is organized as follows: Chapter 2 describes the architecture of the system; Chapter 3 describes the visualizer in detail; Chapter 4 describes the monitor in detail; Chapter 5 describes a general example and a client server flow control example used for monitoring and steering; Chapter 6 compares our system with other similar tools and Chapter 7 is the conclusion.
Chapter 2

THE ARCHITECTURE

The system consists of two parts that go hand in hand. The monitor, which provides the interface for accepting monitoring information, and a visualizer, that provides a graphical representation of the system being monitored. Both the monitor and the visualizer run as agents of Moses \(^1\) under a monitoring law \(L_M\). In addition to this, the visualizer, also accepts a set of draw commands from the monitor for displaying the graphical image.

The visualizer is capable of displaying different shaped nodes (possibly to represent a subject), and links (edges) between nodes (possibly to represent transmission of messages between two subjects). Associated with every node is a control state box and an event box. The control state box is simply a rectangular region where the state of a subject can be displayed. For example, in a token ring, the control state could show when the subject has the token. The event box which is also another rectangular region is useful for displaying any other type of information (typically events that occur at the subject). For example if a subject receives a message, an arrived event could then be displayed in the event box. The event box operates like a stack, with the most recent event displayed at the top. Associated with every link is a log box, useful for recording the messages that flow along the link. The log box also operates in the form of a stack. Thus the stack of log messages would represent the history of messages communicated over a link. It should be noted that all the above objects can be used in a manner different from that described above. For example, the control state box of a node could simply be used to display some statistical information. In this case the node itself might

\(^1\)For a detailed description of Moses please refer to [6]
not even represent a subject. A single node might also be used to represent a group of subjects. The exact use of these objects would depend upon the draw commands that the monitor gives to the visualizer, which would naturally depend upon the application that is being monitored.

Figure 2.1 shows a sample screenshot of the visualizer. The figure shows two subjects named *agent1* and *agent2* being represented by corresponding nodes. The box to the left of the subject node is its control state box and the box to the right of the subject node is its event box. In this figure, the visualizer is showing that *agent1* has a token in its control state and is sending a message "hello" to *agent2* whose control state box is empty. The event box of *agent1* shows that the last event that occurred at the subject was the sent event due to the sending of the message "hello" to *agent2*. Similarly, the event box of *agent2* shows that the last event which occurred at the subject was the sent event due to sending of the message "giveToken" to *agent1*. The visualizer also shows a node called *stats* which indicates the total number of messages sent by each subject (in this example 8 and 4 messages for *agent1* and *agent2* respectively). In this case the *stats* node does not represent any subject in the monitored system, but is simply used to provide some additional information about the monitored system. Further, the *stats* node’s event box has been hidden since it is not required.

The subjects of the distributed system (which are to be monitored) are also running under Moses, but using their own application specific group law $L_G$.

As the subjects running under $L_G$ communicate, their respective controllers send the necessary monitoring information to the monitor. NOTE: The monitor is running under a different law $L_M$ while the subjects are running under the law $L_G$. The transmission of messages from $L_G$ to $L_M$ is done using the export/import interface provided by Moses. Thus the law $L_G$ does not need to have any knowledge of the monitoring law $L_M$. All this law needs to do is simply send the necessary information (by means of exports) to the monitoring law $L_M$. It is the responsibility of $L_M$ to decide what it wants to do with the information sent by the subjects running under the law $L_G$.

Once the monitor receives the information from the monitored subjects, it decides
what to do (by way of giving draw commands to the visualizer) based on the monitoring law $\mathcal{L}_M$. The monitoring law $\mathcal{L}_M$ generates a sequence of draw commands for the visualizer. These draw commands are then logged by the monitor and are further transmitted by the monitor to the visualizer. The visualizer, upon receiving these draw commands, will execute them, to provide a graphical representation of the state of the system which is being monitored. The nature of the draw commands is such that each draw command acts on either a node (alongwith it’s associated control state box and event box) or on a link (alongwith it’s log box). Thus a draw command identifies a node or a link and then specifies the attributes that the node or link must have. The attributes of a node are its color, shape, style, visibility, control state and event. The attributes of a link are its color, style, width, direction, visibility and log. All the
attributes are optional. If a particular attribute in a draw command is missing the visualizer will use a default value for that attribute. The default value of an attribute for a previously existing node or link is the existing value of that attribute. Thus there are no specific draw commands for the creation of nodes and links. When the visualizer receives a draw command; if the node or link is already created, then it uses the draw command to modify the values of the attributes present in the command. If on the other hand the node or link does not exist, the visualizer creates a new node or link with the values of the attributes present in the command and uses its default values for the attributes not mentioned in the draw command. Every node or link has a unique identifier which the visualizer uses to establish if the node or link exists or not. The advantage of this is, that no explicit creation/setup is required for the visualizer. This simplifies the monitor law since there is no need to distinguish between the first appearance of a new node and an existing one. This is how monitoring is achieved in our system.

For example, some of the draw commands that might be used for the picture shown in Figure 2.1 are as follows:

- `drawNode(id(a1),
  name(agent1),
  shape(trapezium),
  style(empty),
  nodeVisibility(true),
  csVisibility(true),
  eventVisibility(true),
  cs(token),
  event(sent(to(agent2),hello)))`

The above command tells the visualizer to draw a node with the attributes as specified in the command. The `id a1` will be used by the visualizer to identify the node in future draw commands. If later, after executing the above command, the visualizer receives a `drawNode` command with `id a1` but without the `name, shape`,
style, and visibility attributes, the visualizer will simply modify the cs and event attribute values for the already existing node, thus leaving the previous values for all the other attributes.

- **drawLink(id(a1,a2),**
  
  style(dashed,linewidth(3)),
  arrow(forward),
  color(black),
  linkVisibility(true),
  logVisibility(true),
  log(hello))

The above command tells the visualizer to draw a link from the node with *id* `a1` to the node with *id* `a2` with the attributes specified in the command. While executing this command, if the visualizer realizes that it has never drawn a node with an *id* `a2`, then it will first draw a new node (giving it an *id* `a2`) with the default values for the attributes required in a drawNode command. Similarly if no node exists with *id* `a1`, the visualizer will first draw that node too.

These draw commands would be typically issued by the monitor law $\mathcal{L}_M$ in the monitor’s controller and delivered to the monitor for transmission to the visualizer. The draw commands are not directly sent to the visualizer, since the monitor controls the transmission of these messages. An Example of such a ruling is shown in Figure 2.2.

In addition to the monitoring, the system is also capable of steering the actions of the monitored subjects. Steering is achieved by having the visualizer send a steering message to one of the subjects being monitored. Once again this communication is provided by the import/export interface of Moses. In this case, the law $\mathcal{L}_M$ exports a steering message which is imported by the law $\mathcal{L}_G$. NOTE: Steering is possible, if and only if the law $\mathcal{L}_M$ allows the export of steering messages to the subjects and the law $\mathcal{L}_G$ allows the imports of such messages. Thus the visualizer does not have unrestricted control to steer the subjects. The steering is subject to the policies $\mathcal{L}_M$ and $\mathcal{L}_G$. In
Initially: The first rule is the one that will be triggered in the group law $\mathcal{L}_G$ of the subject and the second rule is the one that will be triggered in the monitoring law $\mathcal{L}_M$.

\textbf{R1.} \texttt{sent(From,hello,Dest) :-}
\begin{verbatim}
  token@,
  monitor(Monitor)$@,
  do(forward(From,hello,Dest)),
  do(export(From,[hello,Dest],Monitor, monitorLaw)).
\end{verbatim}

This rule is triggered when in the group law $\mathcal{L}_G$, a subject (From) sends a hello message to another subject (Dest). The rule will only succeed if the sending subject has the term token in its control state and an address for the monitor (Monitor) in its control state. The result of this rule succeeding is that the message is forwarded to the intended subject and a message is exported to the monitor informing it about this.

\textbf{R2.} \texttt{imported(From,tokenRing,[hello,Dest],Monitor) :-}
\begin{verbatim}
  do(deliver(From,[drawLink(id(From,Dest),
  style(dashed,linewidth(3)),arrow(forward),
  color(black),linkVisibility(true),
  logVisibility(true),log(hello))],Monitor)).
\end{verbatim}

This rule is triggered when in the group law (tokenRing) $\mathcal{L}_G$, a subject, sends a message "hello" to another subject. At that time this message would be exported from the group law $\mathcal{L}_G$ which would result in an imported event being triggered at the monitors controller under the monitoring law $\mathcal{L}_M$. The rule simply delivers a drawLink command to the monitor to show the transmission of the hello message. This example rule is simplified for understanding purposes. In practice, the monitor law might check for some conditions and update some state information that it has about the subject before sending an appropriate draw command.

In the above rules "From" would be bound to the "id" of the sending subject and "Dest" would be bound to the "id" of the receiving subject and "Monitor" would be bound to the "id" of the monitor agent.

Figure 2.2: Example of a Rule in the Monitor Law $\mathcal{L}_M$

this way steering can be achieved in a controlled manner. It is also possible that in response to a particular rule being triggered, the monitor law might automatically send a steering message to a subject.

The visualizer is also capable of sending messages to the monitor. Such messages request the monitor to change the monitoring configuration. These messages are also
sent under the law $L_M$. The monitoring configuration thus depends on the rules written in the law $L_M$. An example of a change of monitoring configuration would be the visualizer asking the monitor to send some additional statistics of the monitored subjects: like the number of messages the subjects have sent. Once again this kind of statistical information would be possible only if the monitoring law $L_M$ provides for it. In the context of the example shown in Figure 2.1, the visualizer might have requested the monitor to display the statistics of the number of messages sent by the subjects, which is why, the monitor might have given the appropriate draw commands to display the stats node.

Figure 2.3 shows the above described monitoring and steering architecture.

As Figure 2.3 shows, the monitor is not restricted to serve only one visualizer.
The monitor can serve zero or more visualizers. Every time the monitor gets a draw command, it will send the command to all its visualizers. This feature allows for people at different locations (terminals) to visualize the monitored system. Furthermore it also adds to the possibility that multiple visualizers are assigned different roles in the monitor law. Therefore this would allow one visualizer to send steering messages, while not allow other visualizers to do the same. While the system is running, visualizers can be added and removed dynamically. Adding a visualizer dynamically, will cause the newly added visualizer to start receiving draw commands which the monitor issues. The monitor does not send the previously received draw commands to the newly added visualizer. This will cause the newly added visualizer to not initially display the entire system. However, as the other subjects in the system communicate and are monitored, the new visualizer will display them. Thus in time the newly added visualizer will get a picture of the entire system. Due to the nature of the draw commands, only the attributes which need to be changed are specified. Depending on the application, this may cause a newly added visualizer to display an object (node or link) with some wrong attribute values, as it is possible that some previous draw command (which the newly added visualizer did not receive) might set the value of a particular attribute for an object (node or link). Now when the new visualizer receives the next draw command for the same object in question, the command might not contain the previously set attribute value. This would cause the newly added visualizer that would be encountering the object in question for the first time to use its default value for that attribute. As an example suppose the monitor receives a drawNode command for a subject $A$ with the color attribute set to "green". The monitor sends this command to the visualizers that it is serving. After that a new visualizer is added and now the monitor receives another drawNode command for subject $A$ with no value specified for the color attribute. When the monitor sends this command out to its visualizers, the newly added visualizer will create a new node for subject $A$ using its default color (which may not be green) while the other visualizers will update the attributes for the node representing subject $A$. This problem can be solved by writing the monitoring law $\mathcal{L}_M$ appropriately. A simple
way would be to always specify all the attributes of a command. Another way would be to have the monitor law keep track of some default information, so that when a new visualizer is added, the law can identify that and simply send a draw command with all the necessary attributes specified. If the monitor does not have any visualizers to serve, then it will simply log the draw commands for offline monitoring of the system.
Chapter 3

THE VISUALIZER

The visualizer provides a graphical interface for displaying the monitored system. It is designed to give a graph based representation of the monitored subjects. Hence the visualizer has the notion of nodes and links (or edges) between nodes. The implementation of the visualizer uses the Java package of Graphviz developed at AT&T research for displaying the nodes and links. In order to be able to provide a graphical representation of the monitored system, the visualizer relies on the monitor to send it draw commands. The draw commands is the key to displaying the graph of the system. It permits the addition, modification and deletion of nodes and links. For any given node or link there is a myriad of possible attributes that can be used to draw them. Therefore, in order to limit the possible attributes, the draw commands are designed to specify only a small set of possible attributes for a node or link; the most essential we thought were color, shape, visibility, control states, events, logs, etc. Thus, the draw commands cannot completely control the appearance of a node or link. The attributes that cannot be specified via draw commands have default values that the visualizer uses, and can be changed for a particular object (node or link) at the discretion of the user using the visualizer. Menu options, dialog boxes, mouse clicks and mouse drags are part of the user interface that allows the user to modify the attributes that cannot be specified in the draw commands. Examples of such attributes are the font color, font size, control state box position and event box position, etc.
3.1 Draw Commands

Once the visualizer is running, it is waiting to receive commands from the monitor. As the monitor sends the visualizer commands, the visualizer executes those commands, which then results in updating the visualizer’s window. The following are the draw commands that the visualizer accepts.

- `drawNode(id(identifier_for_the_node),
  name(label_to_display_in_the_node),
  shape(shape_of_the_node),
  color(color_of_the_node),
  style(style_to_use_for_coloring_the_node),
  nodeVisibility(visibility_of_the_node),
  csVisibility(visibility_of_the_control_state_box),
  eventVisibility(visibility_of_the_event_box),
  cs(control_state_to_be_displayed_in_the_control_state_box),
  event(event_to_be_displayed_in_the_event_box))`

This command is used to draw a node in the window of the visualizer. Every node drawn is associated with a corresponding control state and event box. The description of the parameters is as follows:

- `id(identifier_for_the_node) - The identifier for the node. It should be the id (typically the host and port) of the subject which is being represented by this node. The visualizer uses the id to uniquely identify a node. If the visualizer already has a node for the id, then it will use the above command to modify the node represented by the id. On the other hand, if the visualizer does not have a node for the id, the visualizer will create a new node for the id with the attributes as specified in the above command. When a new node is created, the visualizer will automatically place it in its window along the circumference of a circle that is centered and fits in the visualizers window. This allows for the placement of new nodes in such a way that a newly created...`
node will always be displayed in the visible portion of the visualizers window. Once displayed the user may choose to move the node to any convenient place he/she likes. The *id* parameter in the draw command is the only one that is not optional, since the visualizer must always know on which node it has to perform the required operations. Thus the drawNode command is used to create a new node as well as to modify an existing node.

**name** (label to display in the node) - The label that appears inside the node for the subject. The label should be something meaningful to identify the subject. In general, the *id* of a subject is its name since it uniquely identifies the subject. However, the *id* may not at all times be very user friendly, and hence in such situations a simple name given to the subject would be better. Note: While the *id* for a subject must be unique, it is not required that the name assigned to a subject be unique, since the visualizer does not use the name to identify the subject.

**shape** (shape of the node) - The shape in which the node has to be drawn. Various shapes are possible. The following shapes are supported by the visualizer - box, circle, diamond, egg, ellipse, hexagon, house, octagon, parallelogram, pentagon, roundedbox, square, trapezium, triangle and wedge. The size of the node is automatically scaled to fit around the name of the node.

**color** (color of the node) - The color in which the node has to be drawn. The following colors are supported by the visualizer - white, gray, black, red, brown, orange, yellow, green, cyan, blue, magenta.

**style** (style to use for coloring the node) - The style in which the node is to be drawn. Currently, the visualizer supports two styles - empty and filled. An empty style creates a node whose internal region is in the background color and whose outline is in the color specified in the drawNode command. A filled style creates a node whose internal region is in the color specified in the drawNode command and whose outline is in black color.
nodeVisibility(visibility of the node) - The visibility of the node. This parameter can be either true or false. A value of true indicates that the node will be visible and a value of false, means the node will be hidden.

csVisibility(visibility of the control state box) - The visibility of the control state box. This parameter can be either true or false. A value of true indicates that the control state box will be visible and a value of false, means the box will be hidden.

eventVisibility(visibility of the event box) - The visibility of the event box. This parameter can be either true or false. A value of true indicates that the event box will be visible and a value of false, means the box will be hidden.

cs(control state to be displayed in the control state box) - The control state that needs to be displayed in the control state box. The visualizer will parse each term given in the control state on a new line in the control state box. For example, cs(token, baton, self(first)), would be parsed so that token, baton and self(first) would appear on different lines in the control state box.

event(event to be displayed in the event box) - The event that needs to be appended at the top of event box. The event box is maintained as a stack so as to show the previous events of the subject. The size of the stack can be changed. This will be discussed in section 3.2.

- drawLink(id(starting_node_id, ending_node_id),
  style(a_style),
  arrow(direction_of_arrow),
  color(color_of_link),
  linkVisibility(visibility_of_the_link),
  logVisibility(visibility_of_the_log_box_of_the_link),
  log(message))

This command is used to draw a link between two nodes in the window of the visualizer. Every link drawn is associated with a corresponding log box, which
shows the message that is being passed between the two nodes at the ends of the link. The two ends of a link are always attached to nodes, and will automatically be redrawn if the end nodes are moved. The description of the parameters is as follows:

\[ \text{id(starting\_node\_id, ending\_node\_id)} - \text{The identifier for the link. The identifier of a link is the concatenation of the node identifiers at the endpoints of the link. If any of the nodes at the endpoints of the link are not present, the visualizer will then create the missing node(s) first using its own default values for the attributes of the newly created node(s). Once again, if a link exists for a given identifier, then the visualizer, will use the drawLink command to simply modify the attributes of the existing link, else the visualizer will create the link. The visualizer is sensitive to the order in which the identifiers for the endpoint nodes of the link is specified i.e. } \text{id(id1, id2)} \text{ identifies a link that goes from } \text{id1} \text{ to } \text{id2} \text{, whereas } \text{id(id2, id1)} \text{ identifies a link that goes from } \text{id2} \text{ to } \text{id1}. \text{ Unlike the other attributes of a link, the } \text{id} \text{ attribute of the link is not optional since the visualizer has to know on which link it has to perform the necessary operations.} \]

\[ \text{style(style)} - \text{The style in which the link has to be drawn. The width of the link can be specified as a style. This is done by the attribute linewidth(W), where W is the width of the link. In addition to this, the manner in which the line has to be drawn can also be specified. The visualizer currently supports three styles - dotted, dashed and solid. As an example, the style parameter could be style(linewidth(4),dotted). This would draw a link that is dotted and four units wide.} \]

\[ \text{arrow(direction\_of\_arrow)} - \text{The arrow style for the link. Currently the visualizer supports three kinds of arrow styles for the link - forward, backward and none. An arrow style of forward draws a link that is pointing from the starting node to the ending node; an arrow style of backward draws a link that is pointing from the ending node to the starting node and an arrows} \]
style of none draws a non directional link (a link without any arrows at either end).

color(color of link) - The color in which the link is to be drawn. The same colors as are available for the links as are for the nodes.

linkVisibility(visibility of the link) - The visibility of the link. The value of this parameter can be either true or false. A value of true means the link will be visible and a value of false means that the link will be hidden. Note: if a link is hidden, then automatically, its corresponding log box will also be hidden regardless of what the log visibility is.

logVisibility(visibility of the log box of the link) - The visibility of the log box. The value of this parameter can be either true or false. However, this parameter only has meaning, when the corresponding link visibility is true. If the link visibility is false, this parameter is ignored and the log box will not be visible. A value of true for this parameter means the log box will be visible, and a value of false means the log box will be hidden.

log(message) - The message that needs to be appended to the top of the log box. The log box is maintained as a stack. This allows to show the previous messages that traversed the link. The size of the stack can be changed. This will be discussed in section 3.2.

These two draw commands (drawNode and drawLink) are the ones that are most commonly used. In these two commands, if any attribute other than the \textit{id} is missing, then based on whether the node/link exists, or not, the visualizer will behave differently. If the node/link exists, the visualizer, will use the existing values of the attributes which have not been specified in the draw command. On the other hand, the visualizer will use its default values for the parameters which have not been specified in the draw command. The default values that the visualizer uses, can be changed by the user. This is discussed in section 3.2.

- deleteNode(node_identifier)
This command is used to delete a node by specifying its identifier. If no node exists for the specified identifier, then visualizer will simply ignore the request. If however, a node with the specified identifier exists, then the visualizer will remove the node along with its associated control state box, event box and all links which were the endpoints of the node in question.

- **deleteLink(starting_node_id, ending_node_id)**

  This command is used to delete a link whose identifier is the concatenation of the starting node id and the ending node id. If no link with the given id exists, then the visualizer will ignore the request. However, if the link with the specified id exists, the visualizer will then delete the link and its associated log box.

  Note: The order in which the starting and ending node identifiers are specified must match the order in which they were given when the link was created using the drawLink command.

- **moveNodeBy(id, relative_x, relative_y)**

  This command is used to move an existing node to a different position in the visualizers window. The displacement is relative to the nodes current position. The node is identified by its id and is moved relative x units in the x direction and relative y units in the y direction. Once again if the node does not exist, the visualizer simply ignores the request. For example, a moveNodeBy(a1, 5, -5) will move the node whose identifier is a1 five units to the right in the x axis direction and minus five units in the y axis direction.

### 3.2 User Interface

The user interface provided by the visualizer can be classified into three broad categories:

1. Picture Control
2. Monitor Control
3. Group Control

Let us consider each of the controls in more detail:

1. Picture Control

Picture control allows the user to perform various operations on the picture displayed by the visualizer. Based on the way in which picture control operations affect the display of the picture they can be classified as follows:

Local Control - As the name implies local control operations only affects a particular object (node, link, control state box, event box and log box). Local control is mainly done through the mouse and context popup menus. The following local control operations are possible: In this discussion the term object refers to a node, a control state box, an event box, a link or a log box.

(a) Object Selection - Clicking the left mouse button on a visible object simply selects the object. Object selection means that the object is highlighted in red, to indicate to the user the object on which the operation will be performed. If an object is not visible it cannot be selected and hence operations cannot be performed on it.

(b) Moving Objects - Clicking the left mouse button on a visible object and dragging the mouse will cause the object to be moved inside the visualizer’s window. Moving a node will cause it’s associated control state box and event box to be moved as well with the same relative displacement. This provides for a logical grouping of a node, its control state box and event box. Moving a node will also move the links which are attached to the node. This ensures that the links are always between the centers of two nodes. A control state box and event box cannot be moved arbitrarily away from their associated node, they can only be moved in close proximity of their associated node. Similarly, a log box can be moved only in close proximity of its associated link. Since a links position is determined by the center of its two endpoint nodes, a
link cannot be directly moved, but gets moved indirectly as a result of moving any one of its endpoint nodes.

(c) Visibility Options - Right clicking the mouse on a visible node or link displays a context popup menu that provides for some local visibility changes. The popup menu for a node allows the user to toggle the visibility of the control state box and event box associated with the node and the popup menu for a link allows the user to toggle the visibility of the log box associated with the link. These options are useful because they allow the user to decide what information one would like to focus on. However, if the user has hidden (or shown) an event box, and then the visualizer receives a drawNode command (for the corresponding node) in which the event box visibility is set to true (or false), the visualizer, will then again display (or hide) the event box. If however, in the drawNode command received by the visualizer, the event box visibility is not specified, then the visualizer will leave the event box hidden (or visible). The same is true for the control state box of a node and the log box of a link. In addition, the visibility option of a log box is only active as long as its corresponding link is visible. Receiving a drawLink command with link visibility set to false will cause the visualizer to hide the corresponding log box even though the user might have made it visible. Having this type of control allows the monitor law $\mathcal{L}_M$ to override the users wish in cases where it is necessary for the user to be informed of changes in the box the user hid. Similarly, it is important to hide the box which is not considered important so as to allows the user to concentrate on other more important information.

(d) Attribute Changes - Right clicking on a visible object displays a dialog box with all the attributes for that particular object. In the case of right clicking a node or link a context popup menu is displayed with one of the menu options to view/modify the attributes. The purpose of this
option is to allow the user to change those attributes for a particular object that cannot be specified by draw commands. Examples of such attributes are font, font color, font size, etc. For example, if a user would like to see the control state of a node in a little larger fontsize, then the user can change the fontsize for the control state box and from then on, the visualizer will use the new font size, for that control state box. Figure 3.1 shows the Attribute dialog box for a node.

![Attributes Dialog Box](image)

**Figure 3.1: Attributes Dialog Box**

**Global Control**

These control operations have an effect on the entire environment of the visualizer. Global control is mainly done through menu options. The following global control operations are possible:

(a) Screen Capture - This option is used to capture the window of the visualizer and save the result in a file. Currently, two file formats are allowed - GIF and postscript.
(b) Default Options - The default options for the nodes and links can be specified in the visualizer.

Node Default Values - The default values for a node come into play, only when a new node needs to be created and some of the attributes of the drawNode command are not specified. In this case those attributes which are not specified, are assigned the corresponding defaults specified in the visualizer. The only defaults which cannot be set for a node are the name of a node, the control state of a node and the event of a node. If the name attribute is not set in the drawNode command, then the visualizer assigns a name to the node. The name assigned by the visualizer will be "Agent" followed by a number that the visualizer increments everytime a new node is created. Having a default name for all nodes is not very useful since if many nodes are created using the defaults, then multiple nodes will have the same name. On the other hand if duplicate node names are required for some application (for example to represent some group of subjects) then, it would be easy to specify that in the monitor law $L_M$, thus overriding the default. The default value of the control state and event of a node are set to be a blank space and cannot be changed since it is not very useful to assume a default value for these attributes. However, if an application needs to use such defaults, the monitoring law $L_M$ can always provide for this. This option also allows the size of the event box stack to be specified. This value determines the number of events the visualizer will keep track of for each node.

Figure 3.2 below shows the "Default Node Options" dialog box displayed by the visualizer.

Edge Default Values - Just like a node’s defaults, the default values of an edge are used only when a new edge needs to be created and
some of the attributes of the drawLink command are not specified. In this case those attributes that are not specified are assigned the corresponding defaults specified in the visualizer. This option also allows the size of the log box stack to be specified. This value determines the number of log message that the visualizer will remember. The default value of a log message is set to be a blank space and cannot be changed since it does not make much practical sense to assume a default message that is communicated between two subjects. However, if an application needs to use such a default, the monitoring law $L_M$ can always provide for this.

Figure 3.3 below shows the "Default Edge Options" dialog box displayed by the visualizer.

2. Monitor Control

As mentioned earlier, the visualizer may send a message to the monitor under the monitoring law $L_M$ asking the monitor to change the monitoring configuration. The exact semantics of the messages passed to the monitor would depend on the
law $\mathcal{L}_M$ and the application. For example, in a client server environment, where there are many clients, the user might wish to focus his/her attention on only a specific set of clients and not wish to see the rest of the clients in the visualizer. In this case the user would communicate the clients he/she is interested in by sending a message under the monitoring law $\mathcal{L}_M$ to the monitor. Once again such a feature is possible only if the monitoring law $\mathcal{L}_M$ provides for it. Visualizer also keeps a history of all the monitoring messages sent to the monitor thus far.

Figure 3.4 shows the dialog box displayed by the visualizer to send messages to the monitor.
3. Group Control

This control allows the visualizer to send steering messages to the monitored subjects communicating under the group law $L_G$. Once again this is possible only if the monitoring law $L_M$ and the group law $L_G$ allows for it. The advantage of this is that steering itself can be performed in a controlled manner and a visualizer cannot have unrestricted control over the monitored group unless allowed by the group explicitly. The visualizer’s interface provides a very convenient way of selecting a node (that represents a subject) to send steering messages to. One could think of this type of control to be local since it simply steers a single subject in the group, but it should be noted that such steering need not be limited to a single subject. This would depend upon the monitoring law $L_M$ or even the group law $L_G$. For example, suppose the monitoring law is written so that when the visualizer sends a steering message halt for a particular subject, the monitoring law $L_M$ could as well multicast this to one or more subjects thus halting an entire group of subjects. Thus the exact semantics of the steering messages would depend on the laws.
Chapter 4
THE MONITOR

The monitor receives the draw commands from the monitor law \( L_M \) and sends these commands to the visualizers. The key role that the monitor plays is in controlling the rate at which the draw commands are sent to the visualizers. The monitor’s interface allows for the dynamic addition and removal of visualizers. This allows for more than one person to visualize the monitored system.

Figure 4.1: GenericMonitor Window

Figure 4.1 shows a sample screenshot of the monitor. As the figure shows, the monitor provides a window into the list of draw commands that it has received. The window always highlights the last draw command that the monitor sent to the visualizers. This window of draw commands will be referred to as the command list box. The draw commands are inserted in a stack fashion, with the new commands being added at the top of the list. At the end of each draw command, the monitor displays a number indicating the relative time in milliseconds between the arrival of the previous draw command and the one against which the number is written. So from Figure 4.1 we can see that the
deleteLink command was received by the monitor 20 milliseconds after it received the drawLink command. These relative delays are useful since the monitor uses them to control the rate at which the draw commands are sent depending on which rate control mode the monitor (discussed later in this chapter) is in. These delays are also used by the monitor when it is not running in online mode to simulate the arrival of the draw commands as in the original run.

At any time, the monitor can be in two possible modes - Online and Offline.

Online Mode
When the monitor is in the online mode, it receives its draw command inputs from the monitoring law $\mathcal{LM}$ of which it is a member. This is the mode the monitor would be in if the system were being monitored as the subjects interacted with each other. When in this mode, the monitor also logs all the draw commands it receives (except the ones that result due to a visualizer sending the monitor a message) into a log file.

Offline Mode
In the offline mode the monitor receives its draw command inputs from a log file that contains them. The log file would typically be a result of some prior online run. While in this mode, the monitor can still continue to receive messages from the visualizer via the monitoring law $\mathcal{LM}$.

4.1 User Interface
The user interface of the monitor allows the user to control the rate at which the draw commands are sent to the visualizers and also allows for the addition and removal of visualizers. The monitor allows for four different rate control modes -

1. Real Time
3. Automatic
4. Scale Time

The rate control modes should not be confused with the online and offline modes. The online and offline modes determine the source from where the draw commands are received by the monitor, while the rate control modes determine the rate at which the monitor sends the draw commands to the visualizers. At any time, the monitor must be in either online or offline mode and in any one of the four rate control modes. The details of the four different rate control modes is as follows:

1. Real Time - When in this mode, as the monitor receives draw commands it will immediately display the command in the command list box and send the command to the visualizers. If the monitor is being switched from some other rate control mode to this mode and there is a back log of draw commands, the monitor will immediately clear out the back log by sending the pending draw commands to the visualizers.

2. Manual - When in this mode, the monitor does not process any draw command on its own. The monitor waits for the user to select the next draw command displayed in the command list box and only then sends the command to the visualizer. This allows the user to see the monitored system one step at a time. The user however, cannot go back in the draw command history i.e. the user cannot undo the effect of any previously issued draw command.

3. Automatic - When in this mode, the monitor will send the draw commands (if any) at a fixed interval. The interval time can be specified by the user. For example if the interval time is set to two seconds, then every two seconds the monitor will check to see if there are any unprocessed draw commands. If there are unprocessed draw commands, the monitor will send the oldest unprocessed draw command to the visualizers and will again wait for the set interval of time. If on the other hand there are no unprocessed draw commands, the monitor will simply wait again for the set interval of time and will repeat the above procedure again.
4. Scale Time - When in this mode, the monitor will scale time by a user specified factor. The time scaling factor can be specified by the user. When in the online mode, the time factor can only be used to multiply real time by a positive integer e.g. one real second may be scaled to two seconds. When in the offline mode, the time factor can be used to multiply or divide the real time by a positive integer e.g. one real second may be scaled down to only a quarter of a second. In the offline mode dividing real time is possible since the monitor has access to the log file and can know what the next draw command is and when it will occur. In the online mode, the monitor has no way of knowing what or when the next draw command will occur. Thus in the online mode this feature can be used only to slow down the system but in the offline mode this feature can be used to slow down or speed up the system. The main purpose of this mode is to allow the user to see the sequence of events in the same relative time frame.

These are all the possible ways in which the user can control the rate at which the monitor sends draw commands to the visualizers. The monitor does not display or log the draw commands that occurred as a result of a visualizer sending a message to the monitor. Such draw commands are not subject to any of the above rate control modes. As soon as the monitor receives such commands, it will send out the commands to the visualizers as soon as possible. The reason why these commands are sent out of order is to allow the user to change the monitoring configuration before executing any further draw commands resulting from the monitored subjects. For example, if the monitor is in the manual mode and has ten unprocessed draw commands waiting to be sent, and the user would like to see some additional information about the subjects, before executing the pending draw commands, then he/she is able to send a message to the monitor for the additional information. The monitor will send the draw commands containing the necessary information and then the user can simply step through the pending draw commands with the additional information provided by the monitor.

NOTE: When in the offline mode, the monitor will use the delays stored in the log file to simulate the actual time in which the events had occurred and will use these
delays for all the above rate control modes.
Chapter 5

EXAMPLES

5.1 General Example

Let us consider a generic monitoring example. Every message that a subject sends to another subject is to be monitored. The visualizer will show the nodes representing the subjects and draw links between the nodes to show communication between subjects.

Figure 5.1 shows the group law $L_G$ under which the subjects would be operating.

Figure 5.2 shows the corresponding monitor law $L_M$ used for monitoring the subjects.

5.2 Flow Control Example

The example in Section 5.1 was simplified and did not show all the setup details. The purpose of the simplified example was to show how easy it was to monitor subjects and give an idea of how the laws would look like. Now let us consider a detailed example of using the system to monitor and steer a group of communicating subjects. The example considers a client server situation. The clients send requests to the servers and the servers reply back. So as to perform some kind of flow control, the clients have some delay $D_i$ for each server $S_i$ that they communicate with. A client is not allowed to send messages faster than the specified delay $D_i$ to the server $S_i$. The flow control mechanism is enforced by the law $L_G$ under which the clients and servers are operating. Further the servers have been coded so that if a particular client has more than some upper threshold of requests queued up at a server, the server will ask that client to stop sending it messages. Once the client’s request queue at the server falls below a
Initially: This example law is simplified to show only the general case of monitoring and does not show any details of the control state being monitored or the setup required for communicating with the monitor. It is assumed in this example, that every subject has in its control state a term \( \text{monitor}(\text{Id}) \) that identifies the monitor to whom the information has to be sent.

\[
\text{Portal}(\text{monitor}, \text{URL}(\text{http://tuesday.rutgers.edu:9020/monitor.law}))
\]
The portal allows the exporting/importing of messages between this law and the law specified in the portal.

\[
\text{PolicyName}(\text{generalLaw}) \quad \text{The name of this law}
\]

\[
\begin{align*}
\text{R1. } & \text{sent(From,Message,Dest) : -} \\
& \text{monitor(} \text{Id} \text{)} @, \\
& \quad \text{do(forward(From,Message,Dest)),} \\
& \quad \text{do(export(From,sent(Message,CS,Dest),monitor,Id))}.
\end{align*}
\]
Whenever a subject sends a message to another subject, the monitor is informed about the message being sent as well as the control state of the subject sending the message. While the above rule is very simple, a more complex rule might have certain conditions to be satisfied on the subject’s control state and might even modify the subject’s control state. Therefore the subject’s control state is being sent to the monitor.

\[
\begin{align*}
\text{R2. } & \text{arrived(From,Message,Dest) : -} \\
& \text{monitor(} \text{Id} \text{)} @, \\
& \quad \text{do(deliver(From,Message,Dest)),} \\
& \quad \text{do(export(From,arrived(Message,CS,Dest),monitor,Id))}.
\end{align*}
\]
This rule is triggered in response to a subject receiving a message from another subject. The rule delivers the message to the subject and also informs the monitor about the message arrival and the control state of the subject. Once again, the above rule is very simple, a more complex rule might have certain conditions to be satisfied on the subject’s control state and might even modify the subject’s control state. Therefore the subject’s control state is being sent to the monitor.

Figure 5.1: A Generic Group Law \( \mathcal{L}_G \)

lower threshold, the server will ask the client to resume sending it messages. Between the stop and resume messages, the client is free to send requests to the other servers.
The monitor and visualizer provide a visual picture of the state of the system at any instant. Both the monitor and visualizer are operating under the monitoring law \( \mathcal{L}_M \).
The law \( \mathcal{L}_G \) exports monitoring messages to the law \( \mathcal{L}_M \).

Under the above described setup, in order for the clients to be able to send messages to the servers, they must know where the servers are, more specifically, where the controllers of the servers are. When a client starts operating under law \( \mathcal{L}_G \), the client
Initially: This example law is simplified to show how monitoring could be done without exploiting application specific information.

Portal(generalLaw, URL(http://tuesday.rutgers.edu:9020/general.law))

The portal allows the exporting/importing of messages between this law and the law specified in the portal.

PolicyName(monitorLaw) The name of this law

R1. imported(From, generalLaw, sent(Message, Cs, Dest), Monitor) :-

do(deliver(From, [drawNode(id(From), cs(Cs),
  event(sent(Message, to(Dest)))]), Monitor)),

do(deliver(From, [drawLink(id(From,Dest), style(dotted, linewidth(2)), arrow(forward), log(Message))], Monitor)).

The monitor is being informed that a subject sent a message to another subject. The monitor delivers a drawNode command to update the control state and event box of the subject that sent the message. The monitor also delivers a drawLink command to show the communication of the message between the subjects.

R2. imported(From, generalLaw, arrived(Message, Cs, Dest), Monitor) :-

do(deliver(Dest, [drawNode(id(Dest), cs(Cs),
  event(arrived(Message, from(From)))]), Monitor)),

do(deliver(Dest, [deleteLink(From,Dest)], Monitor)).

The monitor is being informed that a subject received a message from another subject. The monitor delivers a drawNode command to update the control state and event box of the subject that received the message. The monitor also delivers a deleteLink command to show the successful transmission of the message.

Figure 5.2: A Generic Monitoring Law $L_M$

does not know about the servers or its controllers. One way to allow for communication would be to have the client send a message to its controller identifying the servers and their controllers. This method assumes that the client gets this information from somewhere outside the systems mechanism. Another way would be to have the notion of a group for the clients and servers. Thus inorder to be able to communicate, both the clients and the servers must join a group. The book keeping of the group can be done by another agent (called the secretary). Thus when a client or server starts operating under the law $L_G$ it must also join a group that is maintained by the secretary that is operating under a secretary law $L_S$. Now the client and servers only have to know of where the secretary is and once they have entered that information, they can always ask the secretary for the location of the other members in the group. Thus joining the group result in messages being imported and exported between the law $L_G$ and the
secretary law \( \mathcal{S} \). Thus the secretary serves the role of a name server for the client server group. In fact this notion can be extended further, since to be able to send monitoring information it is required to know the location of the monitor and its controller, the secretary can be used for this information as well. The secretary maintains the notion of a group on a per law basis. Therefore since the monitor and visualizer are operating under law \( \mathcal{M} \), they are not part of the client server group. However, the secretary is capable of handling multiple groups and providing information across these groups.

The law \( \mathcal{G} \) is written in such a way, that as soon as a client or server adds the location of the secretary, the law \( \mathcal{G} \) exports a message to the secretary to join the group. Upon receiving an acceptance message from the secretary (appearing as an imported event in the law \( \mathcal{G} \)), the law \( \mathcal{G} \) once again automatically asks the secretary for the location of the monitor running under the law \( \mathcal{M} \). Upon receiving the monitors location from the secretary (once again in the form of imported messages in the law \( \mathcal{G} \)), the law \( \mathcal{G} \) will inform the monitor of the subjects presence. In the case of a client, the law \( \mathcal{G} \) will in addition to the above setup, also ask the secretary for the location of the servers. Once this setup is completed, the communication between the clients and servers will be monitored and can be steered as well. In this setup, it is assumed that the monitor is running before the servers and clients start running. Further, the servers are assumed to be running before the clients start running. This places a specific order in which the subjects need to be started. It should be noted, that this order is not necessary, but for simplicity we have assumed this order. The law \( \mathcal{G} \) is written so that if the clients or servers do not have a monitors address, then they cannot communicate.

In this example to demonstrate steering, the visualizer can send a message to a subject to change its delay \( D_i \) for a server \( S_i \). Thus the visualizer can control the speed at which the clients messages are received by the servers. If the visualizer notes that a particular client seems to be monopolizing a server, it can change the delay for the client. In this example this steering is done manually, however, it can also be done automatically in the monitor law \( \mathcal{M} \).

Figure 5.3 shows a snapshot of the visualizer’s window for this example. In the
figure, the visualizer shows that there is one client and two servers. The client's control state box (to its left) and event box (to its bottom) are visible, while the servers control state boxes are only visible. The control state box of the client shows that it has a delay of 2 and 5 seconds for server1 and server2 respectively. The event box of the client shows that the client has previously tried to send messages to server2 (for which it has a delay of 5 seconds) faster than it was allowed to. The snapshot has been taken, when server1 is sending a halt request to the client. The server is sending this message to the client since the client has 6 requests queued up at the server, and the server wants to halt the client until it has processed some requests.

Figure 5.3: Snapshot of Client Server Monitoring Example
5.2.1 The Secretary Law $\mathcal{L}_S$

Figure 5.4 shows the secretary law $\mathcal{L}_S$.

Initially: The secretary law is simply used to import groupCommand messages from subjects running under different laws and to export the corresponding groupCommandReplies from the secretary to the subjects. An agent can become a secretary using the role 'secretary' while adopting the law. This law does not contain any details for the naming logic and assumes that the secretary's implementation will handle this.

PolicyName(secr) The name of the policy.

\[ R1. \text{ adoptionRequested}(X, \text{secretary}) :- \]
\[ \text{do(adopt}(X)). \]

Any agent can become a secretary by adopting this law

All the groupCommands sent to the secretary are delivered without modification, since the secretary will have the necessary logic to perform the operations. groupCommands are imported since the subjects running under other laws will export these messages to the secretary.

\[ R2. \text{ imported}(X,L,\{\text{groupCommand}(C)\},Y) :- \]
\[ \text{do(addPortalAcq}(L,X)), \]
\[ \text{do(deliver}(X,\{L,\text{groupCommand}(C)\},Y)). \]

All the groupCommands sent to the secretary are delivered without modification, since the secretary will have the necessary logic to perform the operations. groupCommands are imported since the subjects running under other laws will export these messages to the secretary.

\[ R3. \text{ sent}(X,\{L,\text{groupCommandReply}(C,R)\},Y) :- \]
\[ \text{do(export}(X,\{\text{groupCommandReply}(C,R)\},Y,L)). \]

All the groupCommandReplies will be exported directly to the intended subject. In response to some groupCommand, the secretary will perform the necessary operation and then will reply back to the subject using the message groupCommandReply.

Figure 5.4: Rules of the law $\mathcal{L}_S$ dealing with the Secretary

5.2.2 The Client Server Law $\mathcal{L}_G$

Figure 5.5 gives a summary of the terms used in the control states of the subjects and the rules for adopting the law $\mathcal{L}_G$.

Figures 5.6 and 5.7 show the rules that are used to add the id and address of the secretary, monitor and servers.

Figures 5.8 and 5.9 show the rules that are used for the clients to send requests to the servers and the servers to reply back. These figures also show the rules that the
servers use to halt and resume the operation of the clients. The rules in these figures are the ones that are used to implement the flow control mechanism between the clients and servers.

Figure 5.10 shows the rule that is used for the visualizer to perform steering.

5.2.3 The Monitor Law $\mathcal{L}_M$

Figure 5.11 gives a summary of the terms used in the control states of the agents and the rules for adopting the law $\mathcal{L}_M$.

Figure 5.12 shows the rules that are used to add the id and address of the secretary and in the case of monitor agents, the ids and addresses of the visualizers as well.

Figure 5.13 shows the rules that are used to inform the monitor that a client or server has joined the client server group maintained by the secretary. Everytime a client or server joins, the monitor also informs the visualizer of the ids and addresses of the clients and servers it knows of. This is done so that when the visualizer sends a steering message to a subject, it knows the id and address of the subject.

Figure 5.14 shows the rules that are used when a client sends a message to the monitor informing it that it has a new value for a server delay. Such messages would be sent when the client first starts up and asks the secretary for the servers and when the visualizer sends a steering message to a client to change its delay for some server.

Figure 5.15 shows the rules that are used when the client sends a request or the server sends a reply in response to a request.

Figure 5.16 shows the rules that are used when the server receives a request from a client and when the client receives a reply from a server.

Figure 5.17 shows the rules that are used when a client attempts to send a request to a server faster than the specified delay, or has been halted by the server. It also shows the rules that are used when a server sends and a client receives a halt or resume message from the server.

Figure 5.18 shows the rule that causes the visualizer to perform steering on the monitored subjects. More specifically, in this example, the visualizer is allowed to steer
the clients only.

It is actually possible for the system to monitor subjects operating under different laws altogether. For instance, it could be possible that the servers in this example could be operating under a server law $\mathcal{L}_{\text{SERVER}}$ and the clients could be operating under a client law $\mathcal{L}_{\text{CLIENT}}$ and the client and servers communicate with each other using import/export messages. In this case our system would be able to monitor and steer such a situation if both the server law $\mathcal{L}_{\text{SERVER}}$ and the client law $\mathcal{L}_{\text{CLIENT}}$ were to export monitoring information to the monitor.
Initially: Every subject has in its control state the following terms:

- `secretary(SecretaryId)` The Id of the secretary
- `monitor(MonitorId)` The Id of the monitor
- `self(AgentName)` A Name for the subject

In addition to this, the clients also have the following terms:

- `client` Identifying role of subject as client
- `server(ServerId)` The Id of the server.
  - This term is present of each server that the client knows of.
- `delay(ServerId, DelayTime)` The delay time for the server whose id is `ServerId`.
  - This term is present for each server that the client knows of.
- `lastCall(ServerId, [H,M,S])` The time when the client last sent a message to the server with id `ServerId`.
  - Once again this term is present for each server that the client knows of.
- `halt(ServerId)` This term is present whenever the server with id `ServerId` has explicitly halted the client.

The servers have the term `server` identifying their role as servers.

- `Portal(monitor, URL(http://tuesday.rutgers.edu:9020/monitor.law))`
- `Portal(secr, URL(http://tuesday.rutgers.edu:9020/secr.law))`
  - This law has two defined portals, one for exporting/importing messages to the monitor law and the other for exporting/importing messages to the secretary law.

- `PolicyName(clientserver)` The name of this law

- \[ R1. \] `adoptionRequested(X, [agent(Name,client)]) :-
  do(+self(Name)),
  do(+client),
  do(adopt(X)).`
  - **A subject may request to run under this law as a client.**

- \[ R2. \] `adoptionRequested(X, [agent(Name,server)]) :-
  do(+self(Name)),
  do(+server),
  do(adopt(X)).`
  - **A subject may request to run under this law as a server.**

Figure 5.5: Rules of the law $L_G$ dealing with the Client Server communication
The subject sends an \([\text{addSecretary}(S, \text{Addr})]\) message to add the id and address of the secretary. After adding the secretary id and address, the subject will send a join groupCommand to the secretary.

The subject receives an imported message from the secretary telling it that it has joined the group, and then the subject exports a groupCommand message to the secretary asking for the monitor's id and address.

The client receives an imported message from the secretary giving the monitor's Id and address. The client adds this information in its control state and exports a message to the monitor informing the monitor of its presence. The client then exports a groupCommand message to the secretary asking it for the Id and addresses of the servers.

The server receives an imported message from the secretary giving the monitor's Id and address. The server adds this information in its control state and exports message to the monitor informing the monitor of its presence.
\textbf{R7}. \texttt{imported(Y, secr, [groupCommandReply(whoIs(server), [IdList, AddrList])], X) :-}
\begin{itemize}
  \item client@,
  \item do(addAcq(clientserver, AddrList)),
  \item do(deliver(Y, serverList(IdList), X)),
  \item addServers(IdList).
\end{itemize}

The client receives an imported message from the secretary giving the servers Ids and addresses. The client adds the server Ids by calling the rule \texttt{addServers(IdList)}.

\textbf{R8}. \texttt{addServers([H|T]) :-}
\begin{itemize}
  \item do(+delay(H, 5)),
  \item do(+server(H)),
  \item Clock=[H3, M3, S3, MS3],
  \item do(+lastCall(H, [H3, M3, S3])),
  \item monitor(Id)@,
  \item do(export(Self, server(H, 5), Id, monitor)),
  \item addServers(T).
\end{itemize}

The client adds the server Ids into its control state. It also sets the initial delay for the servers to be 5 secs. As each server is added, the monitor is informed of this addition.

Figure 5.7: Rules of the law $L_G$ dealing with the Client Server communication (Continued)
\[R9.\] \texttt{sent(C, M, S) :-}
\texttt{client@,}
\texttt{monitor(Id)@,}
\texttt{server(S)@,}
\texttt{lastCall(S, [H2,M2,S2])@,}
\texttt{delay(S, DT)@,}
\texttt{Clock=[H3,M3,S3,MS3],}
\texttt{T2 is H2 * 3600 + M2 * 60 + S2 + DT,}
\texttt{T3 is H3 * 3600 + M3 * 60 + S3,}
\texttt{T3 >= T2,}
\texttt{not(halt(S)@),}
\texttt{do(lastCall(S, [H2,M2,S2]) <- lastCall(S, [H3,M3,S3])),}
\texttt{do(forward(C, M, S)),}
\texttt{do(export(C, messageSent(M,S), Id, monitor)).}

The client sends a request message to the server. The client is allowed to send the
message only if it has an Id for the monitor, has an Id for the server, is not sending the
request sooner than the specified delay for the server and is not halted by that server.
If the client sends the message, it also informs the monitor about this.

\[R10.\] \texttt{sent(C, M, S) :-}
\texttt{client@,}
\texttt{server(S)@,}
\texttt{delay(S, DT)@,}
\texttt{monitor(Id)@,}
\texttt{do(deliver(C,[cannotSend],C)),}
\texttt{do(export(C, cannotSend(M, S), Id, monitor)).}

The client sends a request message to the server. The client is not allowed to send the
message since it is sending it to the server sooner than the allowed delay. The client
also informs the monitor about this.

\[R11.\] \texttt{sent(S, M, C) :-}
\texttt{server@,}
\texttt{monitor(Id)@,}
\texttt{do(forward(S, M, C)),}
\texttt{do(export(S, messageSent(M,C), Id, monitor)).}

The server sends a reply back to the client. The reply is generated in the server’s
implementation in response to a request from the client. The reply is sent if the server
has an Id for a monitor. The server also informs the monitor about this reply.

\[R12.\] \texttt{arrived(X, M, Y) :-}
\texttt{monitor(Id)@,}
\texttt{do(deliver(X, M, Y)),}
\texttt{do(export(X, messageArrived(M, Y), Id, monitor)).}

A message arrived for a client/server. The message is delivered to the client/server and
the monitor is informed about it. The message could be either a request message from
a client to a server or a reply message from a server to a client.

Figure 5.8: Rules of the law $L_5$ dealing with the Client Server communication (Continued)
\textbf{R13. sent}(S, [\text{halt}], C) :-
server@, monitor(Id)@, 
do(forward(S, [\text{halt}], C)),
do(export(S, \text{halt}(C), Id, monitor)).

The server sends a \text{[halt]} message to the client. The message is sent provided the server has an Id for a monitor. The server also informs the monitor about this. The logic for when to send this message is in the server’s implementation.

\textbf{R14. arrived}(S, [\text{halt}], C) :-
client@, monitor(Id)@, server(S)@, 
do(+\text{halt}(S)),
do(deliver(S, [\text{halted(S)}], C)),
do(export(C, \text{halted(C,S)}, Id, monitor)).

The client receives a \text{[halt]} message from a server indicating to it to stop sending more messages. If the client has an Id for a monitor and an Id for the server in question, then the client will add the term \text{halt}(S) to its control state to indicate that it cannot send messages to the server with Id S. The client will also inform the monitor about this.

\textbf{R15. sent}(S, [\text{resume}], C) :-
server@, monitor(Id)@, 
do(forward(S, [\text{resume}], C)),
do(export(S, \text{resume(C)}, Id, monitor)).

The server sends a \text{[resume]} message to the client indicating to the client that it can start sending it messages again. The message will be sent only if the server has an Id for a monitor. The monitor is also informed about this. The logic for when to send this message is in the server’s implementation.

\textbf{R16. arrived}(S, [\text{resume}], C) :-
client@, monitor(Id)@, halt(S)@, 
do(-\text{halt}(S)),
do(deliver(S, [\text{resumed(S)}], C)),
do(export(C, \text{resumed(C,S)}, Id, monitor)).

The client receives a \text{[resume]} message from the server indicating that it can start sending messages to the server. If the client has an Id for a monitor and has been halted for the server in question, then the client will remove the \text{halt}(S) term from its control state indicating that it can start sending messages to the server. The monitor is also informed about this.

Figure 5.9: Rules of the law $\mathcal{L}_G$ dealing with the Client Server communication (Continued)
R17. imported(X,monitor,[changeDelay(Val,S)],C) :-
    client@,
    monitor(Id)@,
    server(S)@,
    delay(S, DT)@,
    do(delay(S, DT) <- delay(S, Val)),
    do(deliver(S, [changeDelay(Val)], C)),
    do(export(C, server(S,Val), Id, monitor)).

The client receives a message from the monitor law (visualizer) telling it to change its delay value for a particular server. This change will succeed only if the client, has an Id for the monitor and has an Id for the server. The new delay value is also communicated to the monitor.

Figure 5.10: Rules of the law $\mathcal{L}_G$ dealing with the Client Server communication (Continued)
Initially: The control state of the monitor has the following terms:

- **monitor** This term identifies the role to be that of a monitor
- **secretary(SecretaryId)** The Id of the secretary
- **visualizerList(VisualizerIds)** The Ids of the visualizers that the monitor sends the draw commands to.
- **queue(ServerId, Count)** The total number of requests that are queued up at the server whose id is ServerId.
  
  For each server, the monitor will have this term.
- **client(ClientId,ServerDelayList)** The list of servers along with their delays for the client with Id ClientId. This represents the delays that are imposed on the client for each server that the client knows about.
  
  For each client, the monitor will have this term.
- **matrix(ServerId,ClientRequestQueue)** The list of clients with their pending requests at the server whose id is ServerId.
  
  For each server, the monitor will have this term.

The control state of the visualizer has the following terms:

- **visualizer** This term identifies the role to be that of a visualizer.
- **secretary(SecretaryId)** The Id of the secretary

- **Portal(clientserver,URL(http://tuesday.rutgers.edu:9020/clientserver.law))**
- **Portal(secr,URL(http://tuesday.rutgers.edu:9020/secr.law))** This law has two defined portals, one for exporting/importing messages to the client server law and the other for exporting/importing messages to the secretary law.

- **PolicyName(monitor)** The name of this law

\[ R1. \text{adoptionRequested}(X, [monitor]) :- \]
\[ \text{do(+monitor)}, \]
\[ \text{do(adopt(X))}. \]

*An agent may request to run under this law as a monitor.*

\[ R2. \text{adoptionRequested}(X, [visualizer]) :- \]
\[ \text{do(+visualizer)}, \]
\[ \text{do(adopt(X))}. \]

*An agent may request to run under this law as a visualizer.*

Figure 5.11: Rules of the law $\mathcal{L}_M$ dealing with the Monitoring and Steering of a Group
3. sent(X, [addSecretary(S, Addr)], Y) :-
   do(addAcq(secr, [S, Addr])),
   do(+secretary(S)),
   do(export(X, [groupCommand(jon(Name, CS)], S, secr)).

   The monitor or visualizer sends an [addSecretary(S, Addr)] message to add the id and address of the secretary. After adding the secretary id and address, the agent will send a join groupCommand to the secretary.

4. imported(Y, secr, [groupCommandReply(jon, R)], X) :-
   monitor@,
   do(export(X, [groupCommand(whoIs(visualizer)], Y, secr)).

   The monitor receives an imported message from the secretary telling it that it has joined the group, and then the monitor exports a groupCommand message to the secretary asking for the ids and addresses of all the visualizers that the secretary knows about.

5. imported(Y, secr, [groupCommandReply(C,R)], X) :-
   C=join.

   The visualizer receives an imported message from the secretary telling it that it has joined the group.

6. imported(Y, secr, [groupCommandReply(whoIs(visualizer), [Id, AddrList])], X) :-
   monitor@,
   visualizerList(Ident)@,
   do(visualizerList(Ident) <- visualizerList(Id)),
   do(addAcq(monitor, AddrList)).

   The monitor receives an imported message from the secretary giving the visualizer Ids and addresses. The monitor adds the visualizer Ids and addresses in its control state. This rule is triggered if previously the monitor already knew of some visualizers. Typically this rule is used whenever the monitor adds or removes visualizers.

7. imported(Y, secr, [groupCommandReply(whoIs(visualizer), [Id, AddrList])], X) :-
   monitor@,
   do(+visualizerList(Id)),
   do(addAcq(monitor, AddrList)).

   The monitor receives an imported message from the secretary giving the visualizer Ids and addresses. The monitor adds the visualizer Ids and addresses in its control state. This rule is triggered when the first time the monitor requests the secretary for the visualizer ids and addresses.

8. sent(X, [groupCommand(M)], Dummy) :-
   secretary(S)@,
   do(export(X, [groupCommand(M)], S, secr)).

   The monitor or the visualizer can send any groupCommand to the secretary. The monitor uses this rule to ask the secretary for an updated list of visualizers whenever it adds or removes them. The visualizer uses this to inform the secretary of its removal from the monitor group.

Figure 5.12: Rules of the law $\mathcal{L}_M$ dealing with the Monitoring and Steering of a Group (Continued)
\texttt{\textbf{R9.} imported(X, clientserver, join(Name,server), Y) :-}
\begin{verbatim}
do(+queue(X,0)),
do(deliver(X,[drawNode(id(X),name(Name),shape(octagon),
color(cyan),style(filled),nodeVisibility(true),
csVisibility(true),eventVisibility(true),cs(server(X),queue(0)),
event(admitted(server,X))]), Y)),
acq(clientserver,L@,
visualizerList(Id)@,
do(multicast(Y, appendAcq(L), Id)).
\end{verbatim}

A server has just joined the client server group and is informing the monitor about it. The monitor starts to keep track of the requests at the server, delivers to itself a draw command, and multicasts to all its visualizers the ids and addresses of all the clients and servers that it knows of so far. This multicast is used to allow the visualizer to know of the ids and addresses of the clients and servers so that it can send them steering messages.

\texttt{\textbf{R10.} imported(C, clientserver, join(Name,client), Y) :-}
\begin{verbatim}
do(deliver(C,[drawNode(id(C),name(Name),shape(house),color(cyan),
style(filled),nodeVisibility(true),csVisibility(true),
eventVisibility(true),cs(client(C)),event(admitted(client,C))]), Y)),
acq(clientserver,L@,
visualizerList(Id)@,
do(multicast(Y, appendAcq(L), Id)).
\end{verbatim}

A client has just joined the client server group and is informing the monitor about it. The monitor multicasts to all its visualizers the ids and addresses of all the clients and servers that it knows of so far. This multicast is used to allow the visualizer to know of the ids and addresses of the clients and servers so that it can send them steering messages.

\texttt{\textbf{R11.} arrived(M, appendAcq(L), V) :-}
\begin{verbatim}
visualizer@, do(addAcq(clientserver,L)).
\end{verbatim}

The monitor is informing the visualizer of the ids and addresses of any new clients and servers that it knows of.

Figure 5.13: Rules of the law \( \mathcal{L}_M \) dealing with the Monitoring and Steering of a Group (Continued)
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| R12. | imported(C, clientserver, server(S, D), Y) :-
|      | client(C,L)@,
|      | isServerPresent(server(S, D), L),
|      | updateClient(server(S, D), L, NewL),
|      | do(client(C,L) <- client(C,NewL)),
|      | do(deliver(C, [drawNode(id(C),cs(client(C), NewL))], Y)),
|      | do(deliver(C, [deleteLink(S,C)], Y)). |
| R13. | imported(C, clientserver, server(S, D), Y) :-
|      | client(C,L)@,
|      | do(client(C,L) <- client(C,[server(S,D)|L])),
|      | do(deliver(C, [drawNode(id(C),cs(client(C), [server(S,D)|L])], Y)),
|      | do(deliver(C, [deleteLink(S,C)], Y)). |
| R14. | imported(C, clientserver, server(S, D), Y) :-
|      | do(+client(C,[server(S, D)])),
|      | do(deliver(C, [drawNode(id(C),cs(client(C), [server(S,D)])), Y])). |

The above three rules have to do with a client informing the monitor about an addition/change of a delay for one of its servers. The first rule is triggered if there is a change in the delay of an existing server that the client knows of. The second rule is triggered if there is a new server that the client just added and this is not the first time that the client is adding a new server. The third rule is triggered if the client is for the first time adding a new server.

R15. isServerPresent(server(S, D), [server(S, Old)|Tail]) :- !.

R16. isServerPresent(server(S, D), [server(X, Y)|Tail]) :-
|      | isServerPresent(server(S, D), Tail).

The above two rules are used by the monitor to check if a client already has a delay for a server.

R17. updateClient(server(S, D), [server(S, Old)], [server(S, D)]).

R18. updateClient(server(S, D), [server(X, Y)], [server(X, Y)]).

R19. updateClient(server(S, D), [server(S, Old)|Tail], [server(S, D)|NewL]) :-
|      | updateClient(server(S, D), Tail, NewL).

R20. updateClient(server(S, D), [server(X, Y)|Tail], [server(X, Y)|NewL]) :-
|      | updateClient(server(S, D), Tail, NewL).

The above four rules are used to update the delay value of a server that the client already knows about.

Figure 5.14: Rules of the law $L_M$ dealing with the Monitoring and Steering of a Group (Continued)
imported(X, clientserver, messageSent(M, Dest), Y) :-
  queue(X, Count) @,
  matrix(X, L) @,
  decrementServerQueue(L, Dest, NewL),
  NewCount is Count - 1,
  do(matrix(X, L) <- matrix(X, NewL)),
  do(queue(X, Count) <- queue(X, NewCount)),
  do(deliver(X, [drawLink(id(X, Dest), style(solid, linewidth(3)),
                  arrow(forward), color(black), linkVisibility(true),
                  logVisibility(true), log(M))], Y)),
  do(deliver(X, [drawNode(id(X), cs(server(X), queue(NewCount),
                 NewL))], Y)).

The server is informing the monitor that it sent a reply to a client. The monitor
decrements the total number of requests pending for the server and also decrements the
number of requests pending for the client at the server. It delivers to itself a drawLink
command to show the transmission of the reply and a drawNode command to show the
updated control state of the server.

R 22. imported(X, clientserver, messageSent(M, Dest), Y) :-
  do(deliver(X, [drawLink(id(X, Dest), style(solid, linewidth(3)),
                  arrow(forward), color(blue), linkVisibility(true),
                  logVisibility(true), log(M))], Y)).

The client is sending a request message to the server. The monitor simply sends itself
a drawLink command to show the transmission of the request.

R 23. decrementServerQueue([q(Client, Y)], Client, [q(Client, YY)]) :-
    YY is Y - 1.

R 24. decrementServerQueue([q(X, Y)], Client, [q(X, Y)]).

R 25. decrementServerQueue([q(Client, Y)|Tail], Client, [
    q(Client, YY)|NewL]) :-
    YY is Y - 1,
    decrementServerQueue(Tail, Client, NewL).

R 26. decrementServerQueue([q(X, Y)|Tail], Client, [q(X, Y)|NewL])
    :- decrementServerQueue(Tail, Client, NewL).

The above four rules are used by the monitor to decrement a clients request queue at a
particular server.

Figure 5.15: Rules of the law \( \mathcal{L}_M \) dealing with the Monitoring and Steering of a Group
(Continued)
R27. imported(X, clientserver, messageArrived(M, Dest), Y) :-
    queue(Dest, Count) @,
    matrix(Dest, L) @,
    incrementServerQueue(L, X, NewL),
    NewCount is 1 + Count,
    do(matrix(Dest, L) <- matrix(Dest, NewL)),
    do(queue(Dest, Count) <- queue(Dest, NewCount)),
    do(deliver(X, [deleteLink(X, Dest)], Y)),
    do(deliver(X, [drawNode(id(Dest), cs(server(X), queue(NewCount), NewL))], Y)).

R28. imported(X, clientserver, messageArrived(M, Dest), Y) :-
    queue(Dest, Count) @,
    NewCount is 1 + Count,
    do(+matrix(Dest, [q(X, 1)])),
    do(queue(Dest, Count) <- queue(Dest, NewCount)),
    do(deliver(X, [deleteLink(X, Dest)], Y)),
    do(deliver(X, [drawNode(id(Dest), cs(server(X), queue(NewCount), [q(X, 1)])], Y)).

The above two rules are informing the monitor that the server received a request message from a client. The monitor will increment the total requests queued at the server as well as the request queue of the client at the server. It will then deliver to itself a deleteLink command to show the completion of the request transmission. It will also deliver a drawNode command to show the updated control state of the server.

R29. imported(X, clientserver, messageArrived(M, Dest), Y) :-
    do(deliver(X, [deleteLink(X, Dest)], Y)).

The client is informing the monitor that it received a reply from a server. The monitor delivers to itself a deleteLink command to show the completion of the reply transmission.

R30. incrementServerQueue([q(Client, Y)], Client, [q(Client, YY)]) :-
    YY is Y + 1.

R31. incrementServerQueue([q(X, Y)], Client, [q(X, Y)]).

R32. incrementServerQueue([q(Client, Y)|Tail], Client, [q(Client, YY)|NewL]) :-
    YY is Y + 1,
    incrementServerQueue(Tail, Client, NewL).

R33. incrementServerQueue([q(X, Y)|Tail], Client, [q(X, Y)|NewL]) :-
    incrementServerQueue(Tail, Client, NewL).

The above four rules are used by the monitor to increment a client's request queue at a particular server.

Figure 5.16: Rules of the law $\mathcal{L}_M$ dealing with the Monitoring and Steering of a Group (Continued)
The client is informing the monitor that it tried to send a message to the server, but was not allowed to either because it tried to send messages faster than permitted for the server or the server has explicitly halted it. The monitor sets up an obligation of one second and delivers a drawNode command to itself, painting the client node red. The obligation is set so that after one second, the monitor will paint the client cyan again, thereby giving a flash effect, to attract the attention of the user at the visualizer.

The obligation to change the color of the client node back to cyan has triggered, so the monitor delivers to itself a drawNode command to do the needful.

The server is informing the monitor that it is sending a halt message to a client, since its queue of requests for that client has exceeded some upper threshold. The monitor delivers a drawLink command to show the transmission of the message.

The client is informing the monitor that it received a halt command from a server. The monitor delivers a deleteLink command to show the completion of the halt message transmission.

The server is informing the monitor that it is sending a resume message to a client to allow the client to start sending messages to it again. The monitor delivers a drawLink command to show the transmission of the message.

The client is informing the monitor that it received a resume message from a server. The monitor delivers a deleteLink command to show the completion of the resume message transmission.
R40. sent(X,[changeDelay(Val,S)],Y) :-
    visualizer0,
    do(export(X,[changeDelay(Val,S)],Y,clientserver)).

The visualizer is sending a steering message to a client agent. The steering message is exported to the client server law and tells the client to change its delay value that it has for the server with id S.

Figure 5.18: Rules of the law $L_M$ dealing with the Monitoring and Steering of a Group (Continued)
Chapter 6

RELATED WORK

Several different types of monitoring and steering tools have been developed each with its own advantages and disadvantages. We will compare some of these tools with respect to our model.

1. General and Customizable

Our model is a general tool for monitoring and steering distributed systems using LGI for communication. Just like our tool, ETE (End-To-End) [4], dQUOB (dynamic QUery OBjects) [7], Xab (X-window Analysis and deBugging) [2], HeNCE (Heterogeneous Network Computing Environment) [3] and ParaGraph [5] are general purpose tools for a specific application domain. The HiFi (Hierarchical Filtering) [1] tool is more general in the sense that it is not developed for any specific distributed application domain, but for distributed systems in general. Our visualizer provides a simple programmatic interface for displaying application specific data. The monitoring law $\mathcal{L}_M$ would determine what draw commands are to be given to the visualizer based on the application that is being monitored. ETE [4] has a fixed type of graphical display, while dQUOB [7] and HiFi [1] have no standard visualization tools. Xab [2] provides for simple event visualization and cannot be customized for any application specific visualization. HeNCE [3] too like Xab [2] provides for only standard monitoring displays with no customizable facilities. ParaGraph [5] allows for user defined tasks, however the possible visualizations for these tasks are once again coded in the implementation. While ParaGraph’s [5] visualization capabilities can be extended for a specific application, it requires non trivial X-Windows programming.
2. Non-obtrusive Instrumentation
In our model, since the monitoring and steering capability is provided in the LGI mechanism, monitoring and steering of the heterogeneous subjects can be done with minimum perturbation. In this sense, the ETE [4], dQUOB [7] and HiFi [1] tools are not very suitable for monitoring heterogeneous subjects of whose implementation little is known, since these tools assume the explicit insertion of sensors and steering hooks in the subjects implementation. The Xab [2], HeNCE [3] and ParaGraph [5] tools are also able to provide monitoring with minimum perturbation of the applications, since the applications are run in a parallel programming environment like PVM or MPI (Message Passing Interface) and use the tracing facilities of such environments. ETE [4], Xab [2], HeNCE [3] and ParaGraph [5] do not have any support for steering.

3. Local Event Filtering
In our model, primitive events are specified in the laws used for communication, thus allowing for only selective information to be passed to the monitor. The ETE [4], dQUOB [7] and HiFi [1] models also provide for some sort of event filtering mechanisms. The Xab [2], HeNCE [3] and ParaGraph [5] tools have no event filtering mechanism and process all the events that are given to them from the underlying parallel programming environments (PVM or MPI).

4. Controlled Steering
In our model steering can be performed in a controlled manner. Not only should the monitoring law $L_M$ allow the steering message to be sent, but also the subjects law $L_C$ should allow for steering messages to be received and processed. The dQUOB [7] and HiFi [1] tools rely on explicit steering hooks to be called in the subjects implementation. As mentioned earlier ETE [4], Xab [2], HeNCE [3] and ParaGraph [5] have no steering capabilities.
Chapter 7

CONCLUSION

The model developed in this paper is very useful. The utility of the model however, depends on the amount and kind of information sent by the monitored subjects as well as upon the way the monitoring law is written. This model is capable of not only monitoring a single group of interacting subjects, but also monitoring multiple groups of interacting subjects. It is possible to show inter and intra group interaction using this model. While this model has been designed specifically for subjects using LGI, we do think that it could be useful for monitoring non LGI messages, as long as the messages are passed in the correct format to the monitoring law $\mathcal{L}_M$. This, however, requires the subjects to have some mechanism (like sensors) to send the monitoring information. The visualizer is written in a way that it does not necessarily have to communicate with our monitor, all it requires is the delivery of draw commands. While we have tried to keep the monitor general, it is possible to have a different type of monitor to communicate with the visualizer.

In conclusion, our model is able to monitor and steer heterogeneous subjects in a deployable and non-obtrusive manner. The model is scalable in terms of the number of subjects which can be monitored since the subjects’ law $\mathcal{L}_G$ locally filters the monitoring information that is sent to the monitor. The model is not scalable in the number of monitoring messages, since there is a single monitor that receives all the monitoring messages sent by the subjects’ law $\mathcal{L}_G$. It achieves controlled steering by requiring not only the monitoring law $\mathcal{L}_M$ allow steering messages to be sent, but also the subject’s law $\mathcal{L}_G$ allow for the arrival of such messages. Further, it is in the subject’s law $\mathcal{L}_G$ where the steering action would take place. This prevents the monitor or visualizer
to steer a heterogeneous subject (of which it may not have complete knowledge) in a manner that causes the subject to crash or behave illogically. Such controlled steering is a necessity in a heterogeneous distributed system.
References


