A Study of RMF Monitoring Using DEVS Simulation

by

Min Shao

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Electrical Engineering
and
Bachelor of Science in Electrical Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999

© Min Shao, MCMXCIX. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author..........................................................

Department of Electrical Engineering and Computer Science

May 21, 1999

Certified by......................................................

Hari Balakrishnan
Assistant Professor
Thesis Supervisor

Accepted by......................................................

Arthur C. Smith
Chairman, Department Committee on Graduate Students

Using DEVS Simulation
A Study of RMF Monitoring Using DEVS Simulation

by

Min Shao

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degree of
Master of Electrical Engineering
and
Bachelor of Science in Electrical Science and Engineering

Abstract

In this thesis, I designed and implemented an object-oriented simulation framework which provides information that can be used to optimize end-system monitoring design. The simulation was based on the DEVS formalism which provides a formal representation of discrete event systems capable of mathematical manipulation [8] [9]. The simulation engine is implemented in Java. It has a hierarchical structure and is easy to manage and expand. A sample simulation setup, inputs and results were provided show the effectiveness of the simulation framework.

Thesis Supervisor: Hari Balakrishnan
Title: Assistant Professor
Acknowledgments

I would first like to thank my mentor at IBM, Steve Wood, who provided invaluable help and guidance throughout the course of this research. I would also like to thank the entire MultiMedia group for giving me the opportunity of this research.

At MIT, I would like to thank my thesis supervisor, Prof. Balakrishnan. This thesis would not be possible without his extremely helpful comments and advice.

I would like to thank my parents, for their love, support and encouragement throughout my life and especially this past five years. I would also like to thank all my friends.
# Contents

1 Introduction .................................................. 8

2 DEVS Formalism ............................................... 11
   2.1 Simulation Operation ....................................... 12
   2.2 The DEVS Model ........................................... 13
      2.2.1 Atomic Model ......................................... 15
      2.2.2 Coupled Model ........................................ 17
   2.3 DEVS Processor ........................................... 19
      2.3.1 What is a Processor ................................... 19
      2.3.2 Message Passing ...................................... 21
      2.3.3 Processor Classes .................................... 22
   2.4 Interface ................................................ 25
      2.4.1 Model-Simulator Interfaces .......................... 25
      2.4.2 Simulator-to-Simulator Interface ....................... 26

3 Simulation Engine Implementation ......................... 28
   3.1 Processor ................................................ 28
      3.1.1 Basics .................................................. 30
      3.1.2 Classes of Processor .................................. 33
   3.2 Basic DEVS Models ....................................... 35

4 System Modeling .............................................. 39
   4.1 System Layout ............................................ 40
4.2 Random Request Generator ........................................ 40

4.3 Load Distribution Unit .............................................. 43

4.4 Multi-Media Server .................................................. 45
  4.4.1 Front End ....................................................... 47
  4.4.2 Resource Allocation Unit ...................................... 47
  4.4.3 Multimedia Streaming Control ................................. 49
  4.4.4 Resource Release Unit ......................................... 50
  4.4.5 Resource ........................................................ 50
  4.4.6 Resource Monitor ............................................... 51

5 Simulation Result & Conclusion .................................. 53
List of Figures

2-1 Separation of Model and its Processor ........................................ 11
2-2 Hierarchical Tree Structure ....................................................... 12
2-3 Operation of Simulation ........................................................... 14
2-4 Phase and Sigma ........................................................................ 15
2-5 Ports of a DEVS model ............................................................... 16
2-6 Symmetric Structures of Simulators and Models ............................ 20
2-7 Simulators and Their Messages ................................................... 24

3-1 Simulation Engine Class Diagram .................................................. 29
3-2 Recursive Execution ................................................................... 32
3-3 Three Base Models of DSFM ......................................................... 36

4-1 The Overall Layout of the Models .................................................. 41
4-2 Request Generator’s Phase Transition ............................................. 42
4-3 Class Server and its Sub Models .................................................... 46
4-4 State Transition of the Server Front End ......................................... 47
4-5 State Transition of The Resource Allocation Unit ............................ 48
4-6 The Resource Release Model’s Phase Transition .............................. 50
# List of Tables

2.1 Structure of Atomic Model ........................................ 17
2.2 Structure of a Coupled Model .................................... 18

3.1 Fields of Class Simulator ......................................... 33
3.2 Fields of Class Coordinator ....................................... 34

4.1 Ports of The Resource Allocation Unit .......................... 48
4.2 Ports of The Active Streaming Control ......................... 50
4.3 Ports of the Resource Release Unit .............................. 51
4.4 Ports of the Resource Unit ........................................ 51
4.5 Ports of Resource Monitor ......................................... 52

5.1 Simulation Input .................................................. 54
5.2 Simulation Result .................................................. 54
Chapter 1

Introduction

Improvement in general network computing performance and emergence of Internet2 have propelled growth of high quality Multimedia services. Distributed multimedia applications, such as VoD (Video on Demand), computer conferencing and distance learning, require guarantees on QoS (Quality of Service) parameters, such as network bandwidth, throughput, end-to-end delay, etc [4]. While Internet2 resolves QoS issues on the network/transport level, we still need a resource management framework to provide QoS monitoring and control at end-to-end level[1]. The reason is that many multimedia applications and distributed collaborative environments have dynamically changing and potentially unpredictable QoS requirements. Large scale deployment of such applications will easily impose various constraints on all infrastructure components.

The Resource Management Framework (RMF) implements QoS management functions such as resource reservation, admission control and dynamic resource adaptation. These QoS management functions aim to allocate, distribute and reserve end-systems’ resources to achieve QoS guarantee[2]. An end-system monitoring is to provide these functions with end-system’s resource utilization information.

End-system Monitoring

An end-system monitoring system needs to perform three tasks: 1) to measure (or monitor) a server’s resource utilization. Most of end-systems in the RMF are multimedia streaming
servers. Resources of these servers include CPU, network bandwidth, cache, disk I/O bandwidth etc; 2) to map the measurements from task 1) to an load status indicator which can be understood by the RMF; 3) to update the RMF when a server’s load indicator changes[3][4].

Implementing an efficient, scalable and reliable end-system monitoring system is a difficult task. The monitoring system needs to hide the heterogeneous nature of the network away from RMF. Different types of multimedia servers and sub-networks have different sets of resources that are required to be monitored on for QoS management. On the other hand, the monitoring system needs to present the RMF a homogeneous network environment in which servers’ have a uniform set of load indicator to represent their resource utilization. These indicators are pooled in the RMF for other QoS management functions to use. It is important to use appropriate indicator updating algorithm to ensure that indicators reflect actual server’s load status. Over-updating wastes end-system and network’s resources and under-updating causes indicators to be inaccurate.

**Motivation**

Multimedia streaming applications require resource management infrastructure such as the RMF. An efficient and reliable end-system’s resource monitoring system can improve the RMF’s QoS management performance. Designing such a monitoring system is difficult because of complexity of the RMF. It is desirable to find a way to test and compare efficiency of different end-system monitoring schemes. This helps us optimize the design of the end-system monitoring.

**Problem Statement**

The RMF requires an end-system’s resource monitoring system. However, there is no systematic way to compare performance of different end-system monitoring schemes. And there is no existing resource management framework to test on, either. Evaluating a monitoring scheme is an empirical process and is difficult to use a mathematical mode. The question is: “how do we evaluate this?”.
Thesis Scope

This thesis is to investigate end-system resource monitoring scheme for the RMF. End-system is a multimedia streaming server for Internet2. Resource mainly refers to server’s hardware resources such as CPU, memory, cache, network bandwidth etc. This thesis does not try to identify the best end-system monitoring scheme for one specific RMF. Instead, we are to develop a systematic method of evaluating and testing an end-system monitoring scheme. Because the empirical nature of the problem, our research focuses on simulation.

Contribution

Our research is part of end-to-end QoS management research. The simulation framework and results provides tool and information to optimize architecture design of the infrastructure. Our research is an attempt to develop a systematic way of evaluating network infrastructures in terms of QoS management effectiveness.

About Simulation

Our research involves constructing a simulation framework. Simulation itself is a vast topic. We choose DEVS (Discrete Event System Specification) formalism [8][9] as the basis of our simulation framework for the following reasons. DEVS simulation environment is object-oriented. This makes the simulation framework easy to manage and expand. DEVS simulation environment offers very flexible modeling. This allows us to change or replace models for different simulations.

Road Map

Chapter 2 gives an overview on DEVS formalism. Chapter 3 discuss the implementation issues of the simulation engine. Chapter 4 is the system modeling and section 4.4 is the modeling of an multimedia server. Chapter 5 concludes the thesis.
Chapter 2

DEVS Formalism

The Discrete Event System Specification (DEVS) formalism introduces a very systematic way of building a highly modular and hierarchical simulation system. Models in such a system are independent of other model's implementation. Interactions between models are regulated by well defined interfaces. In this chapter we are to introduce the DEVS formalism that provides underlying framework for our simulation framework.

The first basic idea of the DEVS formalism is separation of modeling and execution of models. This is like the separation of building a car and driving a car. Each model is a finite state machine and its execution is controlled by its corresponding processor, as shown in Figure 2-1. The processor controls a model through an interface.

Figure 2-1: Separation of Model and its Processor
A simulation system based on the DEVS formalism deploys a tree structure as shown in Figure 2-2. A tree structured system can easily have components that are hierarchical and modular. Such a system can be easily managed, modified and expanded for different simulation usages.

In this tree structure, a component can be a leaf or a node. If a component is implemented as a node, that means it is a multi-component module; if a component is implemented as a leave, then it is a complete module by itself. A leave can be expanded to a node without affecting the rest of the tree as long as the interactions with other component stay the same.

The rest of the chapter discuss the following aspects of a DEVS simulation system: execution, models, processors and interfaces that regulates interactions between components.

![Hierarchical Tree Structure](image)

Figure 2-2: Hierarchical Tree Structure

### 2.1 Simulation Operation

Before we discuss the DEVS formalism in great details, it is useful to have some basics knowledge about simulation execution procedure before we dive into details of DEVS formalism. Figure 2-3 illustrates execution steps of a simple DEVS simulation. In this simple simulation tree, we have a transducer that generates jobs with in random intervals. Jobs are sent to a server to be processed. The server consists of two components, a job queue and
a processor. Processing time of a job is fixed at five simulation time unit (stu). If new job arrives when the processor is busy, it will be enqueued into the job queue.

Figure 2-3 shows how the simulation gets carried out along the time line and interactions between models. This figure is intended to give an overview to DEVS simulation. It does not show much details of simulation execution. For example, how does the system know when and which model to trigger and how does each individual model keep track of time and what happens if one model is out of sync with the global clock. We will answer these questions in later sections.

As we can from the figure, execution of a simulation is essentially is a process of event scheduling. In this simple example, the system has to be able to keep track of events that are scheduled to happen at $t=4$, $5$, $10$, $12$, $14$ ... In a more complex simulation system in which there are many more components and much more interactions, without a well defined event scheduling it would be impossible to develop any large scale simulation framework. The DEVS formalism provides elegant solutions to this event scheduling problem. We shall start with the model in a DEVS simulation system.

### 2.2 The DEVS Model

In a DEVS simulation environment, everything is a model. Every entity in the real world domain is mapped to a model in the simulation domain. The nodes and leaves of tree structure of the DEVS simulation system are models.

In this section, we will discuss the structure of the DEVS models. There are essentially two types of models: Atomic Model and Coupled Model. Atomic model corresponds to the leaves in Figure 2-2 and coupled model corresponds to the nodes, including the root of the tree. We will define a few terms related here as references.
Figure 2-3: Operation of Simulation
2.2.1 Atomic Model

An atomic model is the smallest unit in a DEVS simulation environment. They are the bottom of the hierarchical tree and they do not have any sub models or so called child models. Atomic model represents one single entity as a finite state machine (FSM) and it describes an entity’s behavior with state transition specification. A FSM usually has a set of state variables. For a atomic model, there are two key state variables: phase and sigma. Phase indicates what current state a model is in; Sigma is the time left for the current phase.

![Diagram of Phase and Sigma](image)

Figure 2-4: Phase and Sigma

An atomic model has two types of phase transition (or phase transition) to handle internal and external phase transitions respectively: Figure 2-4 illustrates both internal and external phase transitions of a very simple model of a server. We assume that no job arriving interval is greater than server’s processing time for simplicity.

- **internal phase transition**
  
is phase transition that is triggered by model itself. Internal transition occurs when sigma is equal to zero, which means that the model will go to a new phase.

- **external phase transition**
is phase transition that is caused by external events. External transition occurs when the model receives an external event which forces the model to undergo phase change.

A model is useless if it can interact with other models in the system to carry on simulation. The example in Figure 2-4 shows that external transition is triggered by an incoming job. The question is what’s the mechanism that regulates data exchange like this. DEVS model employs port to control incoming and outgoing traffic of the model, shown in Figure 2-5.

![Figure 2-5: Ports of a DEVS model](image)

The model in Figure 2-5 is the server in Figure 2-4. A port can be either an input port or an output port, or both. When input port is accessed, it indicates that the model will probably go through an external phase transition; Every time when an out-port is accessed, it indicates that the model has just finished an internal phase transition. A port can handle multiple types of data. However, if a port receives a unrecognizable data type, the model will take some action such as throwing an exception.

Atomic model has an output function that controls outgoing traffic. This function outputs external events via model’s ports. An atomic model also has a time advance function which keeps track of times of most recent event and next scheduled event.

We can summarize the structure of atomic models in a more standard and mathematical form shown as the expression and Table 2.1.
Table 2.1: Structure of Atomic Model

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>is the set of external input event types</td>
</tr>
<tr>
<td>S</td>
<td>is the sequential state set</td>
</tr>
<tr>
<td>Y</td>
<td>is the set of external event types generated as output</td>
</tr>
<tr>
<td>$\delta_{int}$</td>
<td>is the internal transition function dictating state transition due to internal events</td>
</tr>
<tr>
<td>$\delta_{ext}$</td>
<td>is the external transition function dictating state transition due to external events</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>is the output function generating external events as the output</td>
</tr>
<tr>
<td>$ta$</td>
<td>is the time advance function</td>
</tr>
</tbody>
</table>

$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ [8]

2.2.2 Coupled Model

Atomic model, in most cases, can accurately model a system entity with detailed state transitions specification. However, there are two serious limitations of atomic model. First, atomic model doesn’t scale well with the size of model. If an entity has very complicated behavior, then it becomes very difficult and tedious to model it with a single atomic model because the phase transition functions will be very complex. It is desirable to modularize the modeling. Second, since atomic model does not have child-models, we cannot have a hierarchical simulation structure. In order to solve these two problems, DEVS formalism uses coupled model.

A coupled model has one parent-model, just as an atomic model does, but it has child-models, which themselves can also be coupled models. In Figure 2-2, node $A$ is a coupled model and $B$ and $C$ are $A$'s children and $D$ is $A$'s parent. Notice that $C$ itself is also a coupled model with two atomic models as its children. A coupled model itself does not have phase transition functions, instead, it has functions that coordinates all sub-models.
Table 2.2: Structure of a Coupled Model

\[ D \] is the set of component names;
\[ M_i \] is a basic component model for each \( i \) in \( D \);
\[ I_i \] is a set, the influences of \( i \) for each \( i \) in \( D \);
\[ Z_{i,j} \] is a function, the i-to-j output translation for each \( j \) in \( I_i \);
\[ select \] is a function, the tie-breaking selector.

A coupled model exchanges events with other models through ports, just like an atomic model. However, a coupled model does not process these events directly. Instead, it forwards the event to corresponding ports of sub-models. If the child-model is a coupled model, the message will be kept on forwarded until it reaches an atomic model where it gets processed.

A coupled model does not have its own time advance function like an atomic model. Instead, it selects next most recent scheduled event from its children models and reports that to its parent model. That child model then becomes imminence child of this coupled model. For a coupled model, event scheduling is transformed into a process of selecting imminence child.

A coupled model itself does not process incoming external events, nor does it generate outgoing external events. It only forwards events from or to its child models. Routing of external event is based on the port coupling within the model. Port coupling describes how input ports and output ports are connected. A coupled model keeps a list of port couplings. When there is either outgoing or incoming external event, the model first checks which the event is from and then it checks the port coupling list and finds destination port(s). The corresponding child models are called influences.

We can summarize the structure of a coupled-model in the DEVS formalism as following:

\[ DN =< D, M_i, I_i, Z_{i,j}, select > \] [8]
2.3 DEVS Processor

Both atomic and coupled models are passive, that is, they need some other modules to call its phase transition functions, output functions, and external event handling functions, etc. We have talked about separation of modeling and simulating in the DEVS formalism. We also discussed the structures of models. Now it is time to look at the core of a DEVS simulation system, the processors.

2.3.1 What is a Processor

In a DEVS simulation system, the role of processor to a model is like a pilot to an air plane. Once an air plane is built, we need a pilot to operate it so that it can carry passengers or freight back and forth between airports safely. The simulation operates in a similar way. A processor needs to perform two tasks:

1. execute model’s functions through an interface.

2. cooperate a model into overall simulation system.

3. synchronize model’s action with the global clock.

Figure 2-6, a modified version of Figure 2-2, shows the symmetric structures of processors and models and one-to-one relationship between processor and model in a DEVS simulation system. Notice that links between models are removed. Only the links between simulators are shown. The reason is that links between models are conceptual, but they do not actually exist. The physical bonding that glues a simulation system together is at simulator level.

One important function of processor is global clock distribution and synchronization. Processors communicate with each other by passing Messages. Every message is stamped with current global time when leaves root simulator. A message updates a processor’s internal clock when it arrives in that processor.

A processor synchronizes model’s action with global or local clock. It keeps track of two synchronization variables: LastEventTime and NextEventTime. The first of two records the
model and processor have symmetric structure

In actual implementation
The connections are only between processors.

Figure 2-6: Symmetric Structures of Simulators and Models
time of a event that has happened, either internal or external; the latter one records the
time of an internal event is expected to take place. When a simulator receives a message.

2.3.2 Message Passing

We mentioned “message” in the previous section. Messages are passed between processors
at different levels to keep simulation running. There are four types of messages and they are
described in the following list[8].

- *message
  is passed from processors at level $N - 1$ to processors at level at $N$ to trigger internal
  phase transition at level $N$. This type of message is generated at root of simulation
tree and propagated down to corresponding leaves of the tree.

- x_message
  is passed from processors at level $N - 1$ to processors at level $N$, transporting an
  external events.

- y_message
  is passed from processors at level $N$ to processor at level $N - 1$, transporting an external
  event. It is originated from a leave of tree. It keeps going up upward on a tree. The
  external event will eventually be sent down in the tree wrapped in x_message.

- done_message
  is passed from processors at level $N$ to processors at level $N - 1$, indicating that
  the message sender has completed either an external or an internal transition. A
  done_message will eventually be propagated back to the root of tree.

For a processor, simulation runs in cycles. A simulation cycle starts when it receives a
x_message or a *message and ends when it sends a done message. A simulation cycle will
not end until all sub processors end their simulation cycle.
2.3.3 Processor Classes

There are three types of processor in the DEVS formalism. They are Simulator, Coordinator and Root Simulator. Figure 2-7 shows what types of messages are passed in and out for different classes of simulators.

The Simulator class is used on atomic model. The following pseudo-code describe how simulator process incoming messages and controls its model. Each block of code is a simulation cycle.

```plaintext
when receive *_message
    update clock with time stamp in the message
    if clock == NextEventTime
        execute model's output function
        execute model's internal transition function
        set LastEventTime = clock
        set NextEventTime = clock + model's new \( \sigma \)
        send done_message
    else
        throw synchronization error.
when receive x_message
    update clock with time stamp in the message
    calculated time elapsed since last event
    if (LastEventTime \leq clock \leq NextEventTime)
        execute model's external transition function
        set LastEventTime = clock
        set NextEventTime = clock + model's new \( \sigma \)
        send done_message;
    else
        throw synchronization error.
```

Coordinator's operation is more complex. As shown in Figure 2-7, it has to be able to process
all four types of messages.

when receive *_.message
  clock update
  if clock == NextEventTime
    send *_.message to its imminent child
    set LastEventTime = clock
    set NextEventTime = newNextEventTime
    send done_.message
  else
    throw synchronization error

when receive x_.message
  clock update
  if LastEventTime ≤ clock ≤ NextEventTime
    identify the source port
    find corresponding sink ports from port coupling list
    send x_.message to all receivers
    set LastEventTime = clock
    set NextEventTime = newNextEventTime
    send done_.message
  else
    throw synchronization error

when receive y_.message
  port translation
  if message needs to be forwarded to its parent
    send y_.message
    find influences
    send x_.message to all influences

when receive done_.message
set newNextEventTime

The class, Root Simulator increments simulation clock and is the beginning of a simulation cycle. It is not attached to any model. It only receives done_message from out-most processor.

when receive done_message
    set clock = NextEventTime from out-most coordinator[8]

These codes serve as the templates for actual Java implementation of the engine, which is discussed in next chapter.
2.4 Interface

In order to implement an object-oriented and hierarchical system, interfaces between modules and layers must be well defined. There are two types of interfaces in the DEVS simulation implementation: the interface between model and simulator; the interface between simulators themselves.

2.4.1 Model-Simulator Interfaces

Since in the DEVS formalism, the models are passive, the interfaces are defined on the models instead of on simulator. There are actually two different interfaces: Simulator-to-Atomic Model and Coordinator-to-Coupled Model.

*Processor-to-Atomic Model*

- *External Phase Transition Function*  
  takes external events as inputs and does not output any external events. Calling this method triggers an atomic model to go through an external phase transition. This method is defined as abstract method and inherited by all atomic models.

- *Internal Phase Transition Function*  
  does not take any inputs and does not output any external events, either. Calling of this method triggers an atomic model to go through an internal phase transition.

- *Output Function*  
  decides at what types of events to be sent out at the end of one phase. This method is called before internal phase transition function is called because in the DEVS formalism a model can only output external event before its internal phase transition.

- *Time Advance Function*  
  returns time for next internal phase transition to occur.

*Coordinator-to-Coupled Model*
- **Get Model Name Function**
  returns a model’s name. Simulator needs to name itself during simulation initiation process. Simulator and its corresponding model need to have the same name.

- **Get Receiver Function**
  finds child-models that should receive an incoming external event. This method does not return anything. Instead, it generate a list of names of models that should receive messages.

- **Get Influences Function**
  finds child-models that should receive external events from a child within the same coupled model. It generates a list of model’s names that should receive messages.

- **External Port Translation Function**
  returns a to_port given from_port and child-model. This function only translates external ports.

- **Internal Port Translation Function**
  returns a to_port given from_port and child-model. This function only translates internal ports.

### 2.4.2 Simulator-to-Simulator Interface

There are eight functions that composes simulator-to-simulator interface. They are used to send and receive four different types of messages[9].

- **Send *_message Function**

- **Receive *_message Function**

- **Send x_message Function**

- **Receive x_message Function**

- **Send y_message Function**
• Receive y-message Function

• Send done-message Function

• Receive done-message Function

In order to The interface in Figure 2-1 defines the interaction between a DEVS model and simulator. The interface has to allow model’s simulator to change its phase; to know how much time is left for current phase; to receive outputs from the model; to forward input to the model;
Chapter 3

Simulation Engine Implementation

This chapter discusses the implementation of the simulation engine based on the DEVS formalism. The goal of our implementation includes: 1) to design a simple but complete processor; 2) to develop basic templates for complex modeling; The simulation engine is implemented in Java. We choose Java because it is an object oriented programming language and its feature of being platform independence makes the code portable. We name our simulation engine DSFM (Devs Simulation Framework for Modeling). DSFM provides a ground work for complex modeling and simulation execution.

Figure 3-1 is a diagram of all Java classes of DSFM. The classes listed below interface devsProcessor are used by the three different types of processors; the classes listed below abstract class model are used by the two types of models. Class List holds a list of model name. This class is shared by both processors and models.

3.1 Processor

Processor is the building block of the simulation engine. For the purpose of our research, the implementation of the processor needs to be simple, complete and efficient. Most importantly they have to be easy to added to removed from the engine. The engine is included in Java package devs.
Figure 3-1: Simulation Engine Class Diagram
3.1.1 Basics

A processor has an interface that defines all necessary methods for processors to carry out simulation. The following is a list of methods defined in `devsProcessor`. The most important ones are the eight methods in slanted font. We will show how program executes recursively by passing these message around.

- *get_simulation_clock*
  
  returns current simulation time in a processor.

- *get_last_event_time*
  
  returns simulation time at which the most recent event happens. This event can be either internal or external.

- *get_next_nent_time*
  
  returns simulation time at which next internal phase transition is scheduled to take place.

- *add_child_processor*
  
  adds a child processor or sub processor to a processor. This is intended to ease the sub-processor attachment procedure. It takes care all the processor bonding, clock synchronization. Those “leaf” processors do not need to use this method.

- *set_parent_processor*
  
  sets the parent processor for a processor. This method is used by all processors.

- *get_model_name*
  
  returns its DEVS model’s name.

- *receive_start_signal*
  
  handles incoming `,message` and starts its attached model’s internal transition.

- *receive_output_from_parent*
  
  handles incoming `,message`, which causes its attached model to undergo external transition.
- **receive_output_from_child**
  handles incoming \textit{y-message}, which causes a Coordinator to either forward the message to its parent or send it to corresponding child processor.

- **receive_done_signal**
  handles incoming \textit{done_message} from a coordinator’s child processor.

- **send_start_signal**
  corresponds to \textit{send\_message} function of a coordinator or root simulator.

- **send_output_to_child**
  corresponds to \textit{send\_x_message} function of a DEVS processor.

- **send_output_to_parent**
  corresponds to \textit{send\_y_message} function of a DEVS processor.

- **send_done_to_parent**
  corresponds to \textit{send\_done_message} function of a DEVS processor.

Detailed implementations of these methods follows the templates of pseudo codes in the previous chapter. However it is difficult to see recursive execution from these code. Figure 3-2 illustrates passage passing between different levels of a simulation system. Methods in slanted type are defined for message passing between processors to carry on simulation.

The processor is single threaded instead of multi-threaded. From the Figure 3-2, the execution point moves between nodes and leaves. All the events and activities of models and processors are serialized and handled by a single thread.

There is a trade off between multi-thread and single-thread approaches. Simulation naturally has more than one execution points in the program because in the real world things happen simultaneously. Therefore, multi-thread approach suits the task better in this sense. It also may improve the simulation running speed because in many cases a simulation can take hours to run. However, the implementation of a multi-threaded simulation engine can be
Figure 3-2: Recursive Execution
substantially more complex and difficult to debug. To the scope of our research, we choose single thread for its simplicity.

Because of the one-to-one relationship between model and processor, a processor is initiated by its model which calls the processor's constructor. Then the processor will be initialized. However, the processor will not have a parent processor until its model has a parent model.

A processor handles synchronization errors. When its own clock does not match global clock, it halts the simulation by throwing exceptions. This is important because often time there are many problems in the modeling part which cause synchronization errors. With this exception handling ability, we can detect the modeling problem easily.

### 3.1.2 Classes of Processor

#### Simulator

The first type of the DEVS processors is Simulator. As shown in Figure 3-1, Simulator implements interface devsProcessor. However, it is not necessary to implement all the methods. Four major methods are implemented: ReceiveStartSignal, ReceiveOutputFromParent, SendOutputToParent and SendDoneSignal. The code implementation of the class follows the pseudo codes in Section 2.3.3.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastEventTime</td>
<td>int</td>
<td>time at which last event takes place</td>
</tr>
<tr>
<td>NextEventTime</td>
<td>int</td>
<td>time at which next internal event is expected to take place</td>
</tr>
<tr>
<td>Clock</td>
<td>int</td>
<td>a simulator's internal clock</td>
</tr>
<tr>
<td>ModelName</td>
<td>String</td>
<td>serves as simulator's identification</td>
</tr>
<tr>
<td>devsModel</td>
<td>AtomicModel</td>
<td>the model that simulator is attached to</td>
</tr>
<tr>
<td>Parent</td>
<td>devsProcessor</td>
<td>the coordinator that is parent to the simulator</td>
</tr>
</tbody>
</table>
Table 3.2: Fields of Class Coordinator

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastEventTime</td>
<td>int</td>
<td>time at which last event takes place</td>
</tr>
<tr>
<td>NextEventTime</td>
<td>int</td>
<td>time at which next internal event is expected to take place</td>
</tr>
<tr>
<td>newNextEventTime</td>
<td>int</td>
<td>a new NextEventTime that will be assigned to NextEventTime before a done_message is sent to its parent.</td>
</tr>
<tr>
<td>Influences</td>
<td>List</td>
<td>a list of names of models that should receive an external event from a child.</td>
</tr>
<tr>
<td>Receivers</td>
<td>List</td>
<td>a list of names of models that should receive an external event from their parent child.</td>
</tr>
<tr>
<td>ImminentChildren</td>
<td>List</td>
<td>a list of names of models that are scheduled to have internal event next.</td>
</tr>
<tr>
<td>Clock</td>
<td>int</td>
<td>a simulator’s internal clock</td>
</tr>
<tr>
<td>ModelName</td>
<td>String</td>
<td>serves as simulator’s identification</td>
</tr>
<tr>
<td>devsModel</td>
<td>CoupledModel</td>
<td>the model that coordinator is attached to</td>
</tr>
<tr>
<td>Parent</td>
<td>devsProcessor</td>
<td>the coordinator that is parent to the simulator</td>
</tr>
<tr>
<td>ImminentChild</td>
<td>devsProcessor</td>
<td>the coordinator that is scheduled to undergo internal transition.</td>
</tr>
</tbody>
</table>

Coordinator

Class Coordinator implements interface devsProcessor. The constructor of the class takes a CoupledModel as input. It implements all eight methods that handle message passing.

The two name lists, Influences and Receivers, are initiated by the class constructor. When the coordinator receives a x_message from its parent, it asks its attached model to fill the list of Receivers and forwards the message to all of processor whose names are on the lists. When it receives done_message from all the recipients, it resets Receivers and sends a done_message to its parent processor. When the coordinator receives a y_message from one of his child processor, it consults with its model to fill the list of Influences and forward the message to all influences as x_message. When it receives done_message from all the influences, it resets Influences and sends a done_message to its parent processor to
complete one simulation cycle.

Coordinator chooses its imminent processor based on their \textit{NextEventTime}. In other words, a coordinator has to find a processor with minimum \textit{NextEventTime}. The following is the procedure of finding a imminent processor for next simulation run.

1. coordinator sends out a \textit{*-message} or \textit{x-message} and fills corresponding name lists.

2. coordinator waits until it receives \textit{done-message} from all listed processors.

3. receives a \textit{done-message} from one child processor and uses time stamp in the message to update \textit{newNextEventTime}. If all processors have sent \textit{done-message}, move onto Step 2. If not, loop back to Step 2.

4. find an imminent child whose \textit{NextEventTime} equals the coordinator's own \textit{newNextEventTime}. If there are more than one imminent children, use \textit{tie-breaker} to decide the imminent child for next simulation cycle.

5. coordinator sets \textit{NextEventTime} equal to \textit{newNextEventTime} and sends out a \textit{done-message} to its parent processor.

\textbf{RootSimulator}

The implementation of class \texttt{RootSimulator} is straight forward. It does not have to implement all the methods that have been defined in the \texttt{devsProcessor}. It only has one key field: \texttt{OutMostCoordinator}. Choosing imminent child process is a trivial case here since there is only one child.

\textbf{3.2 Basic DEVS Models}

All models in a DEVS simulation environment are different as they model different objects in the real world. On the other hand, they share the same interfaces to the simulation engine. Many functions defined in these interfaces have the identical implementation. It is efficient
and convenient to extract these functions and group them into a few basic models that can be extended or inherited by other models. These models are called *Basic Models*.

DSFM provides three types of basic models: *Base Model*, *Base Atomic Model* and *Base Coupled Model*. Base Model is the parent for all models. The other two are implemented as two abstract classes that are intended to be inherited by atomic model and coupled model, respectively.

![Diagram of three base models of DSFM]

**Figure 3-3: Three Base Models of DSFM**

**Base Model**

Class *Model* is the parent model for all models in the DSFM. This model mainly define three variables:

- **Priority**
  
  is used by tie-breaker function. Model with higher priority are selected over ones with lower priority if they are both imminent children.

- **ModelName**
is specified by a model that inherits either atomic or coupled model. This variable is passed to name model’s processor.

- **Processor**
  is the reference to the processor that model is attached to. This variable is assigned when a model is initiated.

**Base Atomic Model**

The Base Atomic Model inherits the Model class. It defines three abstract methods to be implemented by its sub models.

- **internal phase transition specification**
- **external phase transition specification**
- **external event generation specification**

The first two are for the internal and external phase transition functions respectively. The third specification requires all atomic models to implement an output function.

**Base Coupled Model**

Unlike the base atomic model, the base coupled model has only one abstract method to be implemented by its subclasses. The reason is that coupled models are more generic and their structures are identical. The major difference is their imminent child processor selection policy.

- **imminent child select specification**

The base coupled model implements several methods to control and manage its sub-models.

- **add child model**
  does not simply adds a model as its sub-model. It also performs two additional operations to complete the model bonding procedure. The two steps are: 1) add the
sub-model’s processor as a sub-processor to this model’s processor; 2) set the processor to be the parent of the sub-processor. After all

- get influences
  performs a for loops search on internal port coupling list to create a list of names of ports that should receive y-messages.

- get receivers
  performs a for loop search on external port coupling list to create a list of names of ports that should receive x-messages.

- port translation specification
  includes internal and external port translation procedures. This method requires a list of port coupling.

A coupled model also have internal and external port translation functions. These two function are standard for all sub classes.
Chapter 4

System Modeling

This chapter discusses the modeling part of our simulation system. The objective of the simulation is to investigate end-system monitoring scheme for the RMF. In order to obtain data from the simulation system, we have to do the following:

1. To map the entities in the real world domain to models in the simulation domain. Although theoretically we can model all entities in the RMF to their finest grain, this brute-force approach is not desirable and certainly not possible in practice. Such a simulation will be too computational intensive to run.

2. To describe an entity’s role and its functionalities using phase transition specifications or multiple models’ networks.

3. To identify appropriate input data set to the simulation. Often times it is difficult to see what are the key parameters that affect output data.

Where to Start

Modeling depends on the objective of our simulation. Input variables and output variables and modeling specifications will be constantly changing. Initial modeling of the system should be as simple and flexible as possible. The modeling of the system should take full advantage of hierarchical structure of DEVS simulation system.
4.1 System Layout

The first step in modeling is mapping of key entities to models. The RMF is to monitor and manage resource for multi-media streaming. Naturally, "resource" is an important entity to model. The term resource lumps all types of resources, including, CPU usage, network bandwidth, disk I/O performance etc, into one single parameter. To have something simple to start with, it is reasonable to use an abstract representation of the resource at the server end. This makes modeling much easier. The second entity to model is the monitor of the end-system monitoring. We can test different monitoring scheme easily by modifying the monitor model. At this point, we consider that most of resources and the resource monitor are physically located in a media server. Therefore, we will have a media server as an independent coupled model. A load distribution unit is needed to test different task loading policies. A job generator is necessary, which serves as the source of a request, while the media servers as the sink of a request.

In addition to those models, we also need additional models to make the simulation framework fully functional. Figure 4-1 is the overall layout of the system. Later sections discuss each individual model.

4.2 Random Request Generator

This module is designed to generate random pseudo requests for multimedia streaming services. It inherits AtomicModel of the DEVS simulation environment. Its state transition is independent of what goes on in the rest of the simulation system. At the end of each send phase, the generator randomly selects a request from a list of pre-made requests, and sends the request by the OutputFunc function. The total number of requests that are to be generated is decided by the parameter TotalRequest in the input file to the simulation.

By employing different GenerateRequest(), we can simulate different request arrival patterns to investigate effectiveness of different monitoring schemes. At this point, the arrival rate of requests is modeled as a Poisson process.
requests are distributed by the load distributor

servers have different processing capacities

we can modify the monitor to test different monitoring schemes
we can expand the resource model.

Figure 4-1: The Overall Layout of the Models
The random nature of request is generated by the self-evident class RandomGenerator. The class contains a nextPoisson() method that returns random number based on a given mean of Poisson process. This mean is determined in the input file to the simulation.

The state transition is simple. Figure 4-2 is a transition diagram. The Sigma of of each send phase is a random variable with normalized PDF. Once the generator has output TotalRequest request, it will go to rest phase.

![Transition Diagram]

Figure 4-2: Request Generator’s Phase Transition

**Ports**

Two ports are needed in this model, one input and one output.

- **generateRequest**
  Output port from which requests are sent.

- **terminate**
  Input port. Accessing of this port means the model has generated required number of requests and the model will go into rest phase.

Each request contains a multimedia clip object, mmclip. In order to keep track of information of how a request is processed in the system, other variables are added. The following is a list of some important variables.
- **GeneratedTime**
  Simulation time when the request is generated.

- **AcceptedTime**
  Simulation time when the request is actually get served by a multimedia server.

- **ServerIndex**
  Index of a multimedia server to which the request is directed.

- **RequestAttempts**
  Number of attempts made by the load distribution unit to direct to a multimedia server.

- **Key**
  This parameter is used by OrderedRequestQueue class as a clue for ordered insertion. The value assigned to this parameter changes as a request goes through different stages in a simulation run. For instance, when a request is actually being served by a server, this variable will be equal to running time of a multimedia clip. So the server can line up all the requests according to their running time to determine which multimedia streaming will end first.

### 4.3 Load Distribution Unit

This load distributor assigns task as follows: it always sends a request to the least loaded server. If two servers are equally loaded, it will choose the one that has been assigned a task least recently. The distributor enqueues incoming requests for multimedia service and distribute them to different servers. It also takes requests that have been rejected by a server due to insufficient resources at the server end. Those rejected requests will be enqueued and re-sent when there are servers becoming available.

The distributor keeps load status information of all servers in a lookup table. This table gets updated by load monitor at server end. When all media servers are busy or overloaded, the distributor stops sending requests and it will wait until one or more servers become
less loaded. The distributor uses two different queues to buffer new incoming requests and rejected requests. The rejected requests have higher priority over the incoming requests.

- **newRequests**
  
  This queue is an instance of class `RequestQueue`, which is an ordinary FIFO queue. This allows us to serve incoming requests on a first come first serve basis.

- **rejectedRequests**
  
  This queue is an instance of class `OrderedRequestQueue`. This class is almost identical to `RequestQueue`, except for `enQueue()` function.

**Ports**

Total four different types of ports are defined for this model. Two of them are actually arrays. The length of the arrays are equal to the number of servers in the simulation system.

- **StreamRequest[]**
  
  are output ports connected to media servers to redirect multimedia requests to servers. The index of array corresponds to the server array’s index.

- **ReportCentral[]**
  
  are input ports connected to media server’s resource monitoring unit. Server’s load status updates come in through these ports.

- **RejectService**
  
  is an input port that receives requests rejected by servers.

- **generateRequest**
  
  is the input port that receives new incoming requests from the request generator.

**State Transition**

The state transition involves three states. The model enters
• idle
state when there are no requests to be processed in either request queue. The Sigma of this phase is infinite.

• work
state when it starts to processes a request. Only one request can be handled at a time. The Sigma of this phase is usually set to 1 simulation time unit.

• hold
state when it is notified that all servers are busy or heavily loaded. It will stop all the on-going processing and wait for one of the servers to become available. The Sigma of this phase is infinite.

4.4 Multi-Media Server

This section includes the implementation of resource, monitor and media server itself. Resource and monitor are treated as server's sub models. The server must be able to handle multiple multimedia streaming simultaneously. This includes starting and terminating of a multimedia service; resource allocation and release for that multimedia service. In order to perform these operations, server adds four sub models.

The model needs to have a resource unit, which includes all resources of a server, such as network bandwidth, CPU, I/O disk bandwidth. The model needs to have a resource monitor unit which converts the status of the resource unit into a simplified server load status indicator and report it to a central monitor.

It is difficult to achieve the goals listed above by using only one model. Server inherits base coupled model and Figure 4-3 shows all the sub models of a server and how a server handles a multimedia request.

Notice that monitor and resource are the models that we want to have. Other four models, on the other hand, are there to ease the task and reduces complexities in monitor and resource.
1. request rejected if not enough resource
2. resource allocated and streaming begins
3. streaming ends, and release resource

Figure 4-3: Class Server and its Sub Models
4.4.1 Front End

This component is to buffer and serialize incoming requests from the load distributor. This module has the longest state transition cycle. This is to guarantee that no requests will be sent into the server before previous request gets processed. This model extends from AtomicModel.

Ports

- **fromPrev**
  is an input port which receives requests that arrive randomly.

- **toNext**
  is an output port that sends requests out in a fixed time interval.

State Transition

The state transition is shown in Figure 4-4. The model stays in idle when there are no requests. Otherwise, it stays in busy. At the end of each busy phase, a request will be sent out to next model via toNext. The names of these models are self-evident.

![State Transition of the Server Front End](image)

Figure 4-4: State Transition of the Server Front End

4.4.2 Resource Allocation Unit

This model allocates necessary resource for a multimedia request in order to start a multimedia streaming. When it accepts a request, it will negotiate with the Resource unit.
<table>
<thead>
<tr>
<th>Port Type</th>
<th>Port Name</th>
<th>When accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>input port</td>
<td>StreamRequest</td>
<td>accept request from the front unit</td>
</tr>
<tr>
<td></td>
<td>NotEnoughResource</td>
<td>insufficient resource notice from the Resource unit</td>
</tr>
<tr>
<td></td>
<td>Allocated</td>
<td>resource has been allocated for a request</td>
</tr>
<tr>
<td>output port</td>
<td>RequestResource</td>
<td>attempts to allocate resource</td>
</tr>
<tr>
<td></td>
<td>BeginStream</td>
<td>a multimedia streaming is set to start</td>
</tr>
<tr>
<td></td>
<td>RejectService</td>
<td>a request is sent back to the load distributor</td>
</tr>
</tbody>
</table>

If there are sufficient resources, the request will be sent to the StartStream unit to start a multimedia streaming session. Otherwise, it will be rejected and sent back to the load distributor. No request queue is needed because it is guaranteed that no request will come in until its precedent request has been processed. This model extends from AtomicModel. Table 4.1 lists all the ports of the models.

There are total five phases for this model. Figure 4-5 shows the state transition of the model. The transition cycle always starts and ends at idle phase. The duration of the cycle is zero simulation time.

1. receive Request at port StreamRequest
2. send ResourceRequested via port RequestResource
3. receive rejection at port NotEnoughResource
4. receive allocated resource at port Allocated
5. send Request via port BeginStream
6. send Request back via port RejectService

Figure 4-5: State Transition of The Resource Allocation Unit
4.4.3 Multimedia Streaming Control

This model is designed to manage active multimedia streams. It starts a multimedia streaming for an incoming request and ends the stream when it reaches At this point, the server has allocated resources for the request. multimedia streaming will be started. Because clips have different running time,

One key component of this model is ActiveStreams of class OrderdeRequestQueue. The ActiveStreams is initiated as an empty ordered list whose elements will be inserted according to their keys in an increasing order. The key in this case is a multimedia clip’s running time. depicts how we utilize this ordered list to manage the starting and ending of multiple services.

At time $\tau$, there are $N$ multimedia clips being streamed and each has $0 \leq t_1 \leq t_2 \leq t_3 \ldots \leq t_N$ simulation time unit left. Sigma of idle is set to $t_1$. Two possible cases are to be considered here.

case 1: no new streams added before $\tau + t_1$

1. at time $\tau + t_1$ the model changes its phase to finish and terminates stream 1 by dequeuing it from ActiveStreams.
2. decrement $t_2, t_3 \ldots t_N$ by $t_1$ using ActiveStreams.WalkThrought($t_1$) function. Now we have a new $t_2', t_3' \ldots t_N'$ where $t_L' = t_L - t_1$.
3. set Sigma of idle equal to $t_2'$. If there are still no new stream started before $\tau + t_2$, case 1 will repeat itself. Notice that $\tau + t_2 = \tau + t_1 + t_2'$.

case 2: new streams added before $\tau + t_1$

1. a new stream will be added at $\tau^* = \tau + e \leq \tau + t_1$. decrement $t_1, t_2 \ldots t_N$ by $e$.
2. start the new stream by inserting it to ActiveStreams, which is an ordered list. Therefore, the new list will be $t_1 - e \leq t_2 - e \leq t_3 - e, \ldots \leq t_{N+1} \leq \ldots \leq t_N - e$.
   Of course, $t_{N+1}$ can be anywhere in the list depending on the value of its Key.
3. set Sigma of idle equal to the first number on the list and go to a new idle phase.
The Active Streaming Control

<table>
<thead>
<tr>
<th>Port Types</th>
<th>Port Name</th>
<th>When accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>input port</td>
<td>BeginStream</td>
<td>start streaming request multimedia clip</td>
</tr>
<tr>
<td>output port</td>
<td>StreamEnds</td>
<td>release allocated resource</td>
</tr>
<tr>
<td></td>
<td>RejectService</td>
<td>a request is rejected because the server is handling maximum number of streaming.</td>
</tr>
</tbody>
</table>

4. if there are more new streams added before the end of idle, the process will repeat case 2, otherwise it will go through case 1.

This unit controls all active streamings to clients.

### 4.4.4 Resource Release Unit

The model releases resource that has been allocated for a multimedia service. It also serves as a sink for all requests generated in the simulation. The operation of this model is very simple: it receives a request and notify the Resource unit that some resource has been returned. It has a job queue for randomly incoming jobs (resource release notification).

Table 4.3 describe the two ports and Figure 4-6 shows the phase transition of the model.

![Figure 4-6: The Resource Release Model's Phase Transition](image)

### 4.4.5 Resource

This class models a multimedia server’s overall resource. At this point, the modeling is simple. All types of resources, from CPU to network bandwidth, are lumped into one integer. We assume a linear relationship between workload and resource utilization. This may not be true.
in some cases. Yet, linear model can be just as effective as other modeling approach. CPU, network bandwidth, memory, cache and other essential hardware and software components can all be considered as resources at server end for multimedia streaming.

The model is simple, but can be expanding by implementing it as a coupled model. Then we can have separate models for CPU, network bandwidth, cache etc.

### 4.4.6 Resource Monitor

This unit monitors a server’s overall resources. It checks the resource unit periodically; converts monitored data into a simple load status indicator; and updates the central monitor.
Table 4.5: Ports of Resource Monitor

<table>
<thead>
<tr>
<th>Port Type</th>
<th>Port Name</th>
<th>Value Type</th>
<th>When accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>input port</td>
<td>MonitoredResult</td>
<td>Load Information</td>
<td>the model converts load information to load status indicator</td>
</tr>
<tr>
<td>output port</td>
<td>Monitoring</td>
<td>no value</td>
<td>the models actively monitors utilization of resource</td>
</tr>
<tr>
<td></td>
<td>ReportCentral</td>
<td>load indicator</td>
<td>the model sends load status indicator to central monitor</td>
</tr>
</tbody>
</table>

with the indicator.

The conversion of load status indicator is pretty simple in this case. Thresholds are selected arbitrarily to determine the status of a server.
Chapter 5

Simulation Result & Conclusion

After the basic simulation frame work has been set up, we can now try to get some data from this simulation system. For instance, we wish to know how request wait time varies as the monitor’s idle time.

**request wait time** is defined as the time between the request is generated and the time when the request gets served.

**monitor idle time** is the time the monitor in the idle state. The longer the idle time, the less frequently the monitor checks the resources. Therefore, the indicator in the central monitor are likely to be out of date.

Table 5.1 is the set of input variable to the simulation. Variable names are very self-evident. Table 5.2 is the result, from which we can see that the correlation between the parameter is not strong. Only when the monitor becomes much less frequent to check the resource, we start see the effects.

This is just one example of using our simulation system. Most of simulation runs will likely have the same setup except for a few parameters. Some testing may require changing of models and simulation networks, which is a relatively easy task in our object-oriented and hierarchical simulation environment.
Table 5.1: Simulation Input

<table>
<thead>
<tr>
<th>Input Variable</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimulationRun</td>
<td>1</td>
<td>n.a.</td>
</tr>
<tr>
<td>SimulationFinishTime</td>
<td>1800</td>
<td>stu</td>
</tr>
<tr>
<td>serverProcessTime</td>
<td>1</td>
<td>stu</td>
</tr>
<tr>
<td>DistributionProcessTime</td>
<td>1</td>
<td>stu</td>
</tr>
<tr>
<td>MonitorIdleTime</td>
<td>5</td>
<td>stu</td>
</tr>
<tr>
<td>serverCount</td>
<td>7</td>
<td>n.a.</td>
</tr>
<tr>
<td>clipCount</td>
<td>10</td>
<td>n.a.</td>
</tr>
<tr>
<td>RequestVolumeMean</td>
<td>10</td>
<td>stu</td>
</tr>
<tr>
<td>totalRequestGenerated</td>
<td>300</td>
<td>n.a.</td>
</tr>
<tr>
<td>Increment</td>
<td>20</td>
<td>stu</td>
</tr>
<tr>
<td>MaxIncrementTo</td>
<td>206</td>
<td>stu</td>
</tr>
<tr>
<td>AverageClipRunningTime</td>
<td>700</td>
<td>stu</td>
</tr>
<tr>
<td>ClipRunningTimeRange</td>
<td>10</td>
<td>stu</td>
</tr>
<tr>
<td>AverageResourceCost</td>
<td>10000</td>
<td>n.a.</td>
</tr>
<tr>
<td>ResourceCostRange</td>
<td>500</td>
<td>n.a.</td>
</tr>
<tr>
<td>AverageServerResource</td>
<td>20</td>
<td>n.a.</td>
</tr>
<tr>
<td>MaxActiveStreamRange</td>
<td>1</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

Table 5.2: Simulation Result

<table>
<thead>
<tr>
<th>idle time</th>
<th>5</th>
<th>25</th>
<th>45</th>
<th>65</th>
<th>85</th>
<th>105</th>
<th>125</th>
<th>145</th>
<th>165</th>
<th>185</th>
<th>205</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait time</td>
<td>600</td>
<td>600</td>
<td>600</td>
<td>600</td>
<td>606</td>
<td>600</td>
<td>600</td>
<td>600</td>
<td>937</td>
<td>600</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2686</td>
</tr>
</tbody>
</table>
Conclusion

Our research developed a discrete event driven simulation framework that aims to solve design problems of end-system monitoring for the RMF. This simulation system is highly object-oriented and hierarchical and manageable. We can expand this simulation system easily to implement a large scale simulation.

The overall base frame is set up as following. The simulation is driven by a random request generator. The load distribution policy was used to assign tasks between media server. The load distributor always tries to send a request to the least busy server. If a request is rejected due to insufficient resource at the server end, the load distributor will wait until one of the servers become available and re-send the request. It will keep trying until the request gets served. Using this task assignment policy and the set of parameters in Table 5.1, we are able to show that there is no direct correlation between monitor idling time and request wait time.

The DEVS formalism turns out to be a very good theory basis for our research. We believe that our approach at empirical problems like this is effective and can be applied to other system and architecture design process to achieve optimization.
Bibliography


