A Modular Client-Server Discrete Event Simulator for Networked Computers *

David Wangerin, Christopher DeCoro, Luis M. Campos, Hugo Coyote and Isaac D. Scherson
{dwangeri, cdecoro, lmcampos, coyote, isaac}@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

A novel simulation framework is described that uses a client-server paradigm combined with a discrete event simulator to allow the simulation modules to be distributed around a centralized core. The framework also simplifies the creation of new simulations by achieving the goals of being general purpose, easily extensible, modular, and distributed. A prototype of the Client-Server Discrete Event Simulator (CS-DEVS) has been developed as a proof of concept. Two examples are presented to demonstrate the power of the new paradigm and prove the correctness of CS-DEVS. The first example is a modeling of the classical bank queuing simulation. The second example models the simulation of a distributed computing system, with complete modules for workload generation, architecture description, and scheduling and load balancing algorithms.

1. Introduction

In the current simulation literature, few simulators are designed to be general purpose enough to serve a wide range of discrete event simulation tasks while at the same time remaining easily extensible and having features such as network transparency and an open and well-defined interface. To fill this need, we have developed the concept of the Client-Server Discrete Event Simulator (CS-DEVS) and implemented a preliminary prototype.

CS-DEVS is a general-purpose discrete event simulator framework that allows for distributed computation over a network. It operates from a centralized core that serves to synchronize and facilitate communication between a set of distributed modules (event generators, resource handlers, etc.) in addition to being the discrete event manager. Thus, CS-DEVS is comprised of both a simulator core and a set of standard runtime libraries for each module. The modules can be either used from a repository of pre-existing modules or written from a template to perform a specific task. The methods for communicating through the core are well defined, so modules created by different users can seamlessly and easily interact with each other. CS-DEVS allows dynamic runtime binding between disparate modules. This binding can be between a set of modules on a single machine or on machines located anywhere on the Internet.

It is important to note that modules do not communicate directly with each other; rather all communications are directed through the core. The core handles all messages to and from modules, as well as the timing of when messages are sent. Indirect communication is a key concept to CS-DEVS, as modules do not know and do not care about whom they are communicating with. To realize the usefulness of indirect communication, consider a situation where multiple modules are generating messages of the same type. Since receiving modules do not care about the source of messages, all the messages are handled identically, regardless of which module generated each message.

CS-DEVS, like all discrete event simulators, handles simulation time in a non-linear fashion. Execution time is not dependent on the length of the simulation time, but is dependent on the number of events that occur in the simulation. A second effect of using discrete events is that modules can submit events before they will actually happen. Since modules are not dependant on a clock, each module is able to execute at maximum possible speed. The core decides when events are triggered and thus synchronizes all modules without using an explicit clock or other timer methods. Modules do not act randomly by submitting events at undefined times, but rather submit events in response to other events. This method gives consistent behavior for all simulations and makes CS-DEVS robust enough to be capable of handling delays in communications that could cause synchronization problems in other styles of simulators.

This paper details the theory and implementation of CS-

*This work was supported in part by NASA-JPL and NASA-GSFC under grants number 1216595 and NAG5-9695 respectively

DEVS and displays how simulations can take advantage of the distributed nature of CS-DEVS. Two examples are given that both demonstrate the process of developing simulations for the framework and validate the advantageous features of CS-DEVS.

2. Related work

2.1. YADDES

Yaddes is an acronym for Yet Another Distributed Discrete Event Simulator [4]. The Yaddes system is a tool for constructing discrete event simulations. The principle features of the Yaddes system are the Yaddes simulation specification language and compiler, libraries that support time synchronization methods, a pseudorandom number generator package, and a distributed statistics collection and reporting package. The Yaddes user prepares a specification of the desired simulation. Yaddes then compiles the specification into a collection of C language subroutines. These subroutines are then compiled using the C compiler and then linked to a synchronization method library to form a complete program that performs the desired simulation.

The advantage of the Yaddes system over other discrete event simulation packages is that it uses a modeling approach that supports several different synchronization methods. In particular, the synchronization methods currently provided are: sequential (event list driven) simulation, distributed simulation using multiple event lists, conservative (ChandyMisra) distributed simulation, and optimistic (virtual time based) distributed simulation. The Yaddes user need not be concerned with the synchronization method used. In fact, every Yaddes specification can be executed using any method merely by linking to the appropriate library. Provided that the specifications are coded properly, the results of a simulation are independent of the synchronization method used.

2.2. OMNeT++

OMNeT++ is a discrete event simulation tool [6]. It is primarily designed to simulate computer networks, multiprocessors and other distributed systems, but it may be useful for modeling other systems too. The goal of the OMNeT++ project is to create a free discrete event simulation system that can be a real alternative to expensive commercial products in the same application area. OMNeT++ is comprised of several components, including a simulation kernel library, a compiler for the NED topology description language (nedc), utilities (random number seed generation tool, makefile creation tool, etc.), and sample simulations.

2.3. SIMSCRIPT II.5

SIMSCRIPT is a commercial discrete event simulator that has been in use for over thirty years. SIMSCRIPT contains a full development environment, development tools, and sample applications. The development environment is built around the programming language SIMSCRIPT II.5. The language is a free-form, English-like simulation development and modeling language. SIMSCRIPT II.5 is designed use with discrete-event and combined discrete/continuous simulations.

3. A client-server simulation framework

CS-DEVS uses a client-server approach to simulations. By combining the client-server paradigm with a discrete event simulator, simulation modules can be distributed across a network. If the dependencies between modules can be minimized, modules can execute at peak efficiency and take advantage of a distributed computing environment. The simulator core acts as a server to provide a consistent set of functions to many different modules, and modules act as clients that use the server's functionality to accomplish a task.

3.1. Simulator core

CS-DEVS distinguishes itself from other simulators in that it uses a centralized core to synchronize all communications. Modules never communicate directly with each other but rather send and receive all events through the core. This paradigm allows the core to be flexible enough to handle virtually any type of simulation, as the core is focused exclusively on the timing of events and not on the details of any events. The paradigm also allows all modules to be concerned with only the internal operations of the module and not timing and communications details.

The core is deceptively simple in both its operations and its implementation. The core consists of a single events queue, a master clock for the simulation, and a list of event types that should be sent to each module. When events are submitted to the core, they are placed in the events queue according to their execution time. Events are only sent to modules when they are at the front of the queue. By maintaining a time-based priority queue for the events, many synchronization problems are removed, as events can never happen out of order. The master clock is tied to the events queue and can be thought of as a side-effect of the maintenance of the events queue. Time is only advanced when all events for the current time have been executed and all modules have responded to the current events. Since the core waits for responses from modules before dispatching the next event, synchronization problems can never arise.

Time is not advanced linearly, but jumps to the time of the next event in the queue. The list of event types to be sent to modules is a filter that ensures that modules only receive the events that affect them. In this way, modules only know about events that are directly related to their operations and a good separation of concerns can be achieved. In addition, the core only waits for responses from modules that can possibly submit new events.

It is important to note that the core does not handle any processing of data for events and does not have predetermined event types. Any data can be attached to events, but the only data of concern to the core is when the event should occur and what is the type of the event. These two data items are standardized in a particular format for events being submitted to the core, and any additional data contained in the event submission is generically considered to be extraneous data for the event. The core never examines the extraneous data and thus the data cannot affect the timing or execution of the core. In addition, the core can handle any type of event without needing any modifications or tailoring to a particular simulation.

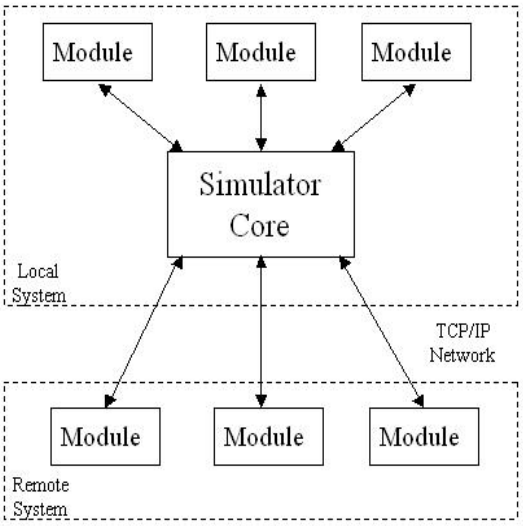

Figure 1. An overview of the layout of the simulator core and modules

3.2. Simulation modules

The interface for modules is also quite simple. Modules register with the core by stating the module name and the types of events it needs to receive. The name is only used for tacking purposes by the core and is not seen by any other modules. After the modules are registered with the core, they submit events to the core and receive events when they are executed. The format of submitted events is the type of event, the relative time of when the event will occur, and any other data that is needed to describe the event.

3.3. Simulation messages

The core and modules communicate through a small set of structured messages. Some messages, such as module registration, are used only for the management of simulations. The majority of messages are submit event messages and event occurring messages, which are directly related to the execution of a simulation. The submit event message has the format of the event type, the relative time of when the event will occur, and any other data for the event. The event occurring message has the format of the type of event, the current time (in absolute time), and the other data that is attached to the event. All other events have a format that is abstracted through the programmer's interface to the module. The exact format of all messages is detailed in A General Purpose Discrete Event Simulator [1]

All modules communicate with the core through TCP/IP network connections. Modules can be located on either the same computer as the core or on different computers attached to a network. The passing of messages over the network is handled by the programmer's interface to modules. Since modules do not need to be on the same computer as the core, and the core does not need to be customized to specific simulations, modules can be independently developed by different programmers at different locations and still interact seamlessly.

3.4. Simulation control flow

Many synchronization problems can arise in simulations when modules respond at different speeds and thus have events occur at different times. CS-DEVS overcomes these problems through the use of relative time and a carefully crafted control flow.

3.5. Relative time versus absolute time

A key concept to the core is that all modules work by relative time and not absolute time. The time field of events submitted to the core is in the format of how many time units in the future the event will happen. This prevents any modules from scheduling events in the past, and thus rollbacks are not necessary.

At first, this handling of time may seem to be a limitation of the core rather than a feature, but closer examination will reveal than any type of simulation can be handled within this framework. In simulations, no actions are completely random. All actions are determined either independently of other actions or are determined in response to another action. For consistency, these two types of actions will be re-

ferred to as independent events and reactive events, respectively. An example of an independent event would be a simulation of computer hardware faults where a fault will occur every two to ten seconds. The timing of faults does not depend on what is happening in the hardware, and thus the timing of all faults can be determined independently from any other events occurring in the simulation. In fact, within the framework of CS-DEVS, all faults could even be determined at the start of a simulation and all fault events could be submitted at once. An example of a reactive event would be a simulation of a computer user where the user performs an action within either one second or one minute of a computer action. Whenever a computer action is executed, the user module will decide if the user will react within one second or one minute, and submit an event accordingly. Note that even though the user is acting in a fashion that may seem random (the choice between reacting in one second or one minute could be data independent), the exact response can be determined at the specific time of the computer action.

3.6. Simulation flow

The control flow of the simulator ensures that no modules can become unsynchronized with the timing of the simulation. Since all modules work in relative time to the current simulation time, the main concern of the control flow is ensuring that modules respond in a synchronized fashion.

Simulations begin by the core issuing a message that a simulation is going to begin. All modules then submit a list of initial events or no events. The simulation will not begin until all modules have responded. Once all initial events are entered, the event queue is checked for the first event. The master clock is updated to the time of the first event and the event is sent to all modules that need to receive the event. All modules that receive the event will either submit new events or no events. When all modules that received the event have responded, the core once again checks the event queue for the next event, updates the master clock, and sends out the events to any modules that need the event. This process continues until all events have been processed (or the core is interrupted by a user).

By waiting for all modules to respond before advancing to the next event, there is no possibility of any event being scheduled at an incorrect time. It also alleviates any problems related to network latency or slow computers needing extra time to compute events. Since events are only sent to modules that need the events, the overhead of waiting for module responses is reduced to a minimum.

4 Simulation of a bank

A standard example often used in literature on simulation is the modeling of daily operations of a bank, with multiple tellers serving multiple customers from one or multiple lines. We use this common example to provide a simple introduction to the process of developing CS-DEVS modules.

The simulated bank, like any real bank, has a set of customers that arrive and wish to perform certain tasks, such as deposits, withdrawals and money transfers. When a customer arrives at the bank, he or she will enter a line. If multiple lines are available, the customer will choose to enter the shortest line. Once in a line, the customer will patiently wait until they are in the front of the line, at which time they will be sent to a teller to handle their business. The teller will complete the customer's tasks in a finite amount of time, and once the tasks are completed, the customer will leave, the teller will service another customer, and so on for the rest of the banking day.

The underlying purpose of the bank simulation is to allow a user to test the effectiveness of various queuing and scheduling schemes. These include: a single first-in, first-out line for all tellers (as is most common in a real-world bank), an individual line for each teller (similar to the lines often seen in a supermarket) and various other systems, including dynamically re-prioritized lines and scheduling systems without the first-in, first-out restriction.

This section describes how this general idea of a bank simulation maps onto the specific set of functionality provided by CS-DEVS. Specifically, this involves describing each of the modules in the system, the events created and received by each module, and the response each module will perform given a particular event.

4.1 Bank simulation modules

The bank simulation is divided into four modules: the Customers and Tellers, the Bank Line, and the Bank Auditor. The Customers module will produce a set of customers that will arrive at the bank at certain intervals to carry out their business. Once a customer enters the bank, the Bank Line module will assign the customer to a particular line, maintain the queue of customers in each line, and assign customers to tellers when tellers are available. The Tellers module will manage the servicing of each customer and determine how long each teller service will take. Monitoring all of this activity is the Bank Auditor module. It tracks the statistics regarding the bank and produces a report of items including average customer wait time, average line length, and utilization of tellers.

4.1.1 Customers

The Customers module is charged with the responsibility of determining when customers will arrive at the bank. The individual algorithm for determining this is internal to the module. In our specific example, the Customers module will be given a distribution range for the total number of customers in a given bank day, and a distribution range for the length of task each customer will need performed. These distributions may be specified as either uniformly distributed across the range, or normally distributed around a mean value.

In this way, the details of the customer's business is abstracted from the rest of the bank simulation, allowing the simulation to focus on its main goal of testing different scheduling systems. While the customer may have a deposit or withdrawal, the Customers module will abstract this by assigning a specific time to the individual customer.

Since the arrival of customers is not dependant on any events that occur within the bank, the Customers module will generate and submit all customers at the beginning of the simulation. The core will manage when the customers are passed to the Bank Line module. Even though it will appear that the Customers module is active throughout the simulation (since customers will appear at the bank throughout the simulated day), it is only needed at the start of the simulation and only needs to make one large communication to the core.

The only type of event generated by the Customers module is the Customer Arrival event. The Customers module does not respond to any events, so it does not need to listen for any events.

4.1.2 Bank line

The most variety is seen in the Bank Line module. This module has a large set of different algorithms that allows the user of the simulation to experiment with different scheduling schemes. In this specific simulation, the Bank Line module has three different algorithms: a single line algorithm, a multiple independent line algorithm, and a dynamic priority-based line algorithm.

When a Customer Arrival event indicates that a new customer has entered into the bank, the Bank Line will direct that customer to the appropriate queue. Which queue, and to which position in that queue, is determined by the scheduling algorithm.

The single-line algorithm is the simplest scheduling scheme. All customers enter the back of a single line, and all customers are dispatched to tellers from the front of the line. Customers may not change positions in the line, and must wait in a first-in, first-out fashion. This scheme is the one most often used in real banks.

In contrast to the single-line algorithm, the multiple-independent-line algorithm has one line for each active teller at the bank. Each individual line operates under the same rules as the single-line algorithm and thus acts as a FIFO queue. When a new customer enters the bank, the Bank Line module will place the customer in the shortest available line. If a teller becomes inactive and there are customers in that teller's line, the customers will be distributed to the remaining lines as through they were new customers. This scheme is not often seen in real banks, but is commonly used in supermarkets and fast food restaurants.

Like the single-line algorithm, the dynamic priority-based-line algorithm uses a single line, but does not operate in a first-in, first-out fashion. The algorithm determines when a customer will be serviced based upon the length of the transactions the customer wants to perform and can change customers' positions in the line at any time. Customers with the shortest transaction time will be placed at the front of the line. Further, when a customer has waited in the line for an extended period of time, the algorithm increases the customer's priority, such that that customer will preempt other customers with lower priority. This system would be unacceptable for a line at a bank, but is often used in telephone service systems (where the line is not visible to the customers).

When the Bank Line module is informed by the Tellers module, through a Teller Available event, that a particular teller is available to service a customer, the Bank Line responds by removing the appropriate customer from the line and sending that customer over to the teller, indicated by the Send To Teller event.

4.1.3 Tellers

The Tellers module manages the availability of tellers and the amount of time each teller will take to service a customer. All tellers are considered equal in their working efficiency and the number of tellers present during any day does not change.

The Tellers module will initially submit a Teller Available event for each teller that is able to service a customer. The Bank Line module assigns customers to tellers through Send To Teller events. When a customer is sent to a teller, the customer is serviced for an amount of time as specified by the customer, and then a Teller Available event is issued to indicate that the teller is ready for another customer.

4.1.4 Bank auditor

The Bank Auditor module collects statistics about the performance of the simulated bank. It listens to all events and tracks the actions of customers and tellers. Whenever an event occurs, the Bank Auditor module updates its statistics on the average customer wait time, average line length, and

utilization of tellers. The Bank Auditor module is, for all intents and purposes, invisible to all other modules. It does not generate any events and listens to all events.

4.2 Bank simulation events

The communication between each module is performed using a set of user-defined events. These events serve both functional and informational purposes. Functional events, such as Send To Teller perform a specific task upon the data, and trigger specific changes in modules other than their sender. Informational events, on the other hand, serve to notify another module of a specific occurrence, which can then be used for statistics tracking, such as the Customer Arrival event.

4.2.1 Bank-open Event

This event is sent at the start of a simulation by the Tellers module. It will inform the Bank Line of the current number of tellers, and serve to allow the simulation to be restarted.

4.2.2 Customer-arrival Event

Whenever a customer enters the bank, the Customers module sends a Customer Arrival event. This event contains a customer record that contains a customer identification number, the arrival time of the customer, and the duration of the customer's particular tasks. This record will be passed around the simulation, stored during the course of the customer's time at the bank, and used to compute statistics once that customer leaves.

4.2.3 Teller-available Event

Once customers arrive at the bank, they expect to be serviced by tellers. To indicate that a teller is ready, the Tellers module will submit a Teller Available message, which will indicate the number of the particular teller. The Bank Line will receive this message, and when the lines are non-empty, use this information to determine which customer will be sent to which teller.

4.2.4 Send-to-teller Event

After the Tellers module has sent a Teller Available event, the Bank Line module responds by assigning a customer to a particular teller, using the Send To Teller event. This event will contain the number of the teller to which the customer is being sent, as well as the customer record submitted in the Customer Arrival event (section 4.2.2)

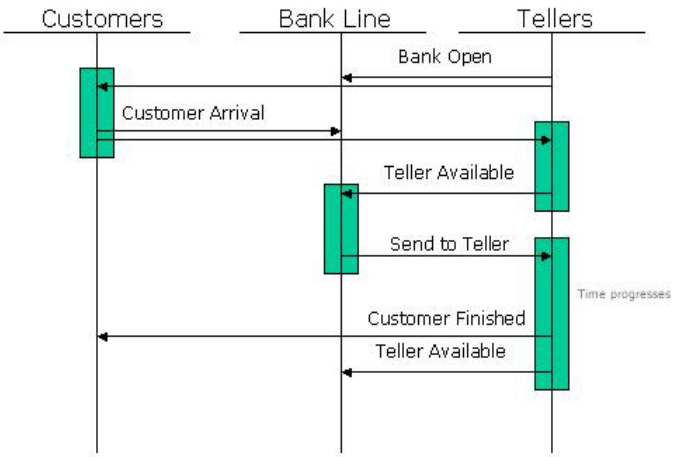

Figure 2. The flow of messages between modules

4.3 Bank simulation control flow

Step 1: Simulation starts when the Tellers module submits a Bank Open event, containing the number of tellers available, and indicating that the bank is open for business.

Step 2: Now that the bank is open, the customers begin to arrive, as the Customers module generates customer records, and submits a set of Customer Arrival events.

Step 3: Once the customers arrive at the bank, the Bank Line module receives the Customer Arrival events, and adds the customers to the line (or lines) according to the pre-assigned scheduling algorithm. Additionally, tellers indicate their availability, through Teller Available events.

Step 4: If tellers are currently available, the Bank Line module will use its scheduling algorithm to select an appropriate customer and teller, and send the customer over to that teller, indicated with a Send To Teller event.

Step 5: After the customer proceeds to the teller, the teller will help that customer with their business, and once that business is complete, the customer will leave the bank and the teller will again become available. The simulation indicates this when the Tellers module submits a Teller Available event.

Step 6: As long as more customers arrive at the bank, the simulation repeats Step 3 through Step 5. At the end of the banking day, once no more customers arrive at the bank, the bank closes, and the simulation terminates.

5 Distributed computing simulation

While the simulation of a bank is a common textbook example of discrete-event simulation, a more practical use of CS-DEVS, and indeed one of the main goals for the simulator, is the simulation of distributed computing algorithms, specifically for distributed and multi-computer operating systems. The simulation needs to accurately model the working of a computer network, either through an inter-processor network or inter-computer network, the scheduling and load balancing algorithms for each processor and the entire network, and the types of programs being run on the computer network.

5.1 Distributed computing modules

The simulation contains three distinct modules: the architecture module, which handles all details of the inner workings of processors and memory systems, the scheduler module, which contains the scheduling and load balancing algorithms, and the workload module, which generates models of programs to be run on the computer system.

By separating the modules into these three groups, the simulation can be used to test either the effectiveness of scheduling algorithms, the performance of computer processors and networks, or the execution of various types of programs on a multiprocessor system.

5.1.1 Architecture

The architecture module contains a representation of the number and type of processors used and the network connecting the processors. The specific details of the architecture implementation are abstracted by the module, and made irrelevant to other modules. An architecture may be specified as a shared-bus multiprocessor system or a cluster system, and still export the same interface to the rest of the simulation.

The Architecture module handles all details of tasks being sent to processors, tasks being processed, and tasks being removed from processors. When a task is assigned to a processor, a start-task event is sent to the Architecture module. The module then processes the task until either the task is completed or the end of a time slice occurs. A task is removed from a processor through either a task-complete event or a time-slice-end event.

Details of inter-processor communications are also abstracted through the Architecture module. Any transmission of data that does not involve moving a task is handled through the data-transfer events and data-transfer-complete events.

To describe the inter-connection networks and processor computing power, two different models can be used. One model is a high level description of an architecture with only four parameters and the other model is a low level description with many more details and settings.

The high level model is a description called LogP [3] [2]. The description contains four parameters. Latency is the time required to send data between any two nodes. Overhead is the processing time associated with setting up to send and receive data. Gap is the minimum wait time between sending messages. P is the relative processing speed of each processor. All connections and all processors are considered to be identical.

The low level model has descriptions for all network connections and all processors. The network is represented with an adjacency matrix. Each entry in the matrix is a value for the connection speed. Note that with an adjacency matrix, each node can have different connection speeds for each incoming and outgoing connection. Each processor has a value associated with it that indicates its processing speed.

5.1.2 Scheduler

It is the responsibility of the scheduler module to both schedule tasks to run on particular processors and to schedule data transfers between processors. When the scheduler receives a task-arrival event from the Workload module, the scheduler must determine which processors are available and how and where the task should be scheduled to run. Additionally, the scheduler must determine from the information in the task-arrival event if the task has any data dependencies from previous tasks, and if so, it must initiate a data transfer, and cannot schedule that process to run until that data transfer is complete.

5.1.3 Workload

Whereas the scheduler module is charged with the responsibility of scheduling tasks, it is the workload module itself that generates those tasks. The workload is also responsible for assigning tasks into separate jobs, if applicable. This may be useful for certain scheduling system, such as gang scheduling, which attempts to run all tasks from a particular job simultaneously.

To indicate the arrival of a job or a task to the system, the workload module uses the job-arrival and task-arrival messages. Each of these messages contains an identifier of the particular job or task. The task-arrival message also contains a Task object. This task object contains the length of the task, which is used in the simulation to determine how much time is remaining for that particular task to run. The Task object also indicates data dependencies between this task and other tasks, which are identified with their task identifiers. It is then the responsibility of the Scheduler, not

the Workload, to determine if a particular task will require a data transfer to obtain its required data.

Once the scheduler has indicated that a task is complete by sending the task-complete event, the workload will determine if another task is generated, and will send out another task-arrival message, if appropriate. The Scheduler itself is abstracted from the details of the workload specification - the Workload module itself handles the details of task creation and communication. The scheduler is only required to obtain the information from the workload and act appropriately.

For the current implementation, the workload is represented as a data-dependency graph, represented in the SHARPE format [5] for task graphs. However, the simulation framework is not specific to any particular representation. For the generation of SHARPE tasks graphs, we have developed a synthetic workload generator, which given a set of workload characteristics, will produce an appropriate graph.

5.2 Distributed computing events

5.2.1 System-start event

The system start event is sent by the architecture to signal the start of the simulation. It contains necessary information about the architecture, including the LogP characteristics. This event is also used to allow the system to restart.

5.2.2 Job-arrival event

A job-arrival event is sent by the Workload module to the Scheduler, and indicates the start of a job. A job is an entity that is made up of smaller tasks, and is used as a grouping for those tasks.

5.2.3 Task-arrival event

Once the Workload module has sent a job-arrival event, it will then send task-arrival events for each task within the job, as they are ready to execute. This event will contain a task object, which will be stored by the scheduler, and contains an identifier of the task, the length of the task, and an array of data dependencies. This array contains the identification number of each previous task on which the current task depends.

5.2.4 Start-task event

In order for the Scheduler to indicate to the Architecture that it wishes to start a task, the Scheduler sends the start-task message. This will provide a specific Task object, and will identify the processor on which to run.

5.2.5 Time-slice-end event

After the Architecture has received a start-tasks message, it will run that task on the indicated processor for a given time quantum, after which time the processor will be interrupted, and the Scheduler will be informed that the time slice has ended. This time-slice-end message will contain the interrupting processor and the interrupted task. The Scheduler then has the opportunity to either reschedule the interrupted task, or to run another task in its place.

The Time Slice End event is one of two events that advances time, along with the data-transfer-complete event. All other events occur in zero time. The Architecture module will schedule a time-slice-end event after the processor has directed it to start a task. Once all other events for the current simulation time have been processed, the simulator core will relay the time-slice-end event to all listening modules and increase the simulation time accordingly.

5.2.6 Task-complete event

Eventually, as tasks are run, interrupted, and rescheduled, all tasks will come to completion. When this happens, the Scheduler indicates this with the task-complete event, which contains the identification number of the completed task. This event will be received by the Workload, which will then determine which additional tasks, if any, are to be started. The Workload then sends additional task-arrival events, and the process repeats itself.

5.2.7 Data-transfer event

When a system has multiple processors, it is possible that the processors may not have a unified memory architecture. In this case, the distributed operating system (represented by the Scheduler module) must ensure that task data is properly transferred to newly arriving tasks, when required. When the Scheduler has received a task-arrival message, and it determines that the new task will be running on a different processor than the tasks on which it is dependant, the Scheduler must then transfer data from those processors to the current processor. This event is sent to the Architecture module, and will contain the source and destination processors of the data transfer operation.

5.2.8 Data-transfer-complete event

Once the Scheduler has requested a data-transfer operation, the Architecture must eventually respond with a data-transfer-complete event. The Architecture will use the LogP model to determine when the data transfer could be performed, and the time at which it would be complete, and schedules this event to occur at that time. As noted previously, this event and the time-slice-end event are the only

events that advance time. The data-transfer-complete event will contain the same information as the data-transfer event, including the source and destination processors of the data transfer.

5.3 Distributed computing simulation control flow

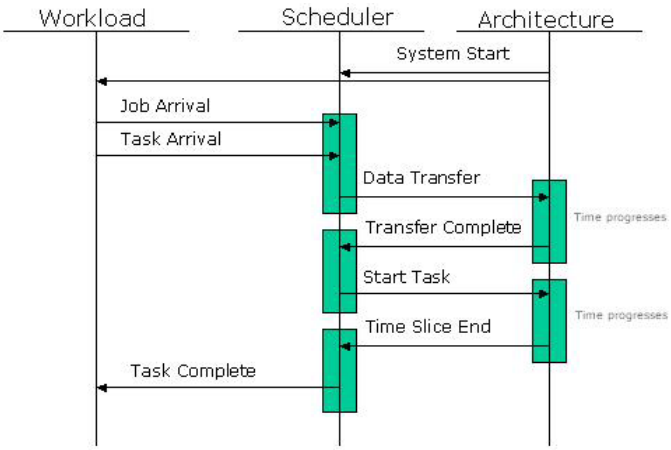

Figure 3. The flow of messages between modules

Step 1: The Architecture module sends a system-start event to all other modules

Step 2: The workload module sends the initial job-arrival and task-arrival events.

Step 3: After receiving the task arrivals, the Scheduler must then determine if the tasks have a data dependency, and schedule a data transfer, if necessary.

Step 4: If there are no dependencies, the system schedules the tasks to run on the first available processor, which is indicated by a time-slice-end message, by sending a start-task event to the Architecture module.

Step 5: If the scheduler was forced to resolve a data dependency, the tasks cannot be run until the particular data transfer is complete, in which case the task can be scheduled.

Step 6: After a task is scheduled, it will eventually be interrupted or terminated, which the architecture indicates to the scheduler via the time-slice-end event.

Step 7: If the task has completed, the scheduler indicates this by sending a task-completed event.

Step 8: Given the task-completed event, the workload will determine if other tasks are spawned, and if so, sends additional task-arrival events.

Step 9: The simulation repeats Step 3 through Step 8 until the entire workload has been completed.

6 Conclusions

The client-server paradigm is easily compatible with discrete event simulators, since all events can be categorized as either independent or reactive events. If a simulation contains a large number of independent events, a simulation can benefit from the client-server architecture, because there are few dependencies between modules, and thus modules can take advantage of the distributed nature of a client-server environment.

The simulation framework is applicable for both of the experimental simulations. The core does not need to be modified in either of the simulations, and the inner workings of the modules are developed independently of all other modules. The modules are able to communicate through the core and never need direct communication. In addition, all modules are able to execute on machines other than the machine running the core

All of the goals of the simulation framework are accomplished and proven through the examples. The examples are also distinct enough to demonstrate that the CS-DEVS framework can handle virtually any type of simulation.

References

[1] L. M. Campos et al. A General-Purpose Discrete Event Simulator. In *Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2001.

[2] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A Practical Model of Parallel Computation. *Communications of the ACM*, 39(11):78–85, 1996.

[3] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[4] B. R. Preiss and I. D. MacIntyre. YADDES - Yet Another Distributed Discrete Event Simulator. Technical report, Dept. of Electrial and Computer Engineering and Computer Communications Networks Group, University of Waterloo, 1990.

[5] K. Trivedi et al. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic Publishers, 1995.

[6] A. Varga. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, 2001.