The Application of Distributed Discrete Event Simulation Algorithms to Concatenated LOcal-area and Wide-area Network (CLOWN) Simulator

Mario Carreon
Department of Computer Science
University of the Philippines Diliman
Diliman, Quezon City
mario.carreon@gmail.com

ABSTRACT
The Concatenated LOcal-area and Wide-area Network or CLOWN simulator is a network simulator programmed using object-oriented C that runs on a single processor. This study aims to take advantage of the inherent parallelism of network models by implementing a distributed Java version of CLOWN (D-CLOWN) that can operate in two modes: one using the conservative Null Message protocol, the other using the optimistic Time Warp protocol. This paper pinpoints issues arising from this new implementation, describes the high-level system architecture for model programmers, and provides benchmarks of the new system (and its different modes) as compared to a non-distributed Java version of CLOWN.

Keywords: Distributed event simulation, Java

1. INTRODUCTION
Simulation is defined to be the study of a model of a system. This model is an approximation of the system, designed to more or less mimic the behavior of the actual system when exposed to the same scenario while being less expensive (in terms of cost or time) than the actual system.

Due to the nature of the system model or the target objective, simulation may take an exceedingly long time to run. Kuratti in [1] describes engineering simulations that involve over a billion events and take days of simulation time to execute. And that's just for a single simulation run; dozens simulation runs must be executed over a large set of scenarios to effectively judge the system being modeled.

Increasing simulation speed can be done by increasing the number of processors handling the simulation. There are two strategies for this. One is by running on each processor an instance of the simulation engine but running different test data (Single Instruction, Multiple Data set). The second strategy is by running a single simulation engine on multiple processors (Multiple Instruction, Multiple data set). Literature defines the former as "parallelizing" the simulation while the latter is called "distributing" the simulation. [2]

Distributing a machine into a set of logical processes [2] requires an additional level of complexity. First there is a need for message passing across a network, as data now resides in several computers rather than shared memory. In addition, synchronization protocols are now needed to keep the system running as a single unit. Although this may seem complicated as compared to parallel simulation, some models may take up too many resources to be run on a single machine: some simulations can only be executed on a distributed system.

This paper discusses the re-implementation of the Concatenated LOcal-area and Wide-area Network (CLOWN) simulator into the distributed paradigm (D-CLOWN). First is a discussion of the original CLOWN system. After that, this paper tackles two synchronization protocols: the conservative Null Message and the optimistic Time Warp. As D-CLOWN is implemented in Java, this paper surveys previous attempts of other researchers in creating their own Java-based distributed simulations.

After the review of related literature, this paper delves into D-CLOWN itself. In particular, this paper talks about the distributed system architecture, followed a summary of the D-CLOWN API, and finally some issues arising from the port from C to distributed Java.

The final part of this paper presents benchmarks for D-CLOWN. D-CLOWN is made to run on Null Message and Time Warp with a non-distributed Java version of CLOWN (as not to take programming language differences into account) over different network models to provide comparsion. These network models target performance issues like message passing delay vs shared memory and effects of changing the number of computers running the simulation. Also, a section is devoted on the theoretical aspect of D-CLOWN, how experimental data of a FIFO queue model matches those predicted in Queueing theory.

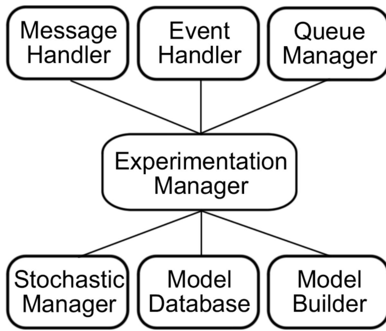

Figure 1: CLOWN System architecture

2. REVIEW OF RELATED LITERATURE

This section discusses the CLOWN system architecture and the process by which a network model is loaded for simulation. Converting CLOWN into the distributed paradigm involves an entirely new framework, so a review of distributed discrete event simulation is needed. Then this section delves into the two synchronization protocols to be implemented in D-CLOWN: Null Message protocol and the Time Warp algorithm.

2.1 CLOWN simulator

The Concatenated LOcal and Wide Area Network (CLOWN) [3, 4, 5] is an object-oriented simulation environment written in C. This section discusses the system architecture of CLOWN (shown in Figure 1) in relation to the process of loading and running a network model.

CLOWN simulation model creation has two parts. The first is though the creation of user-defined modules or by modifying built-in network object libraries.

A CLOWN module defines how a module executes events through function calls. It would follow that each simulation event type has a corresponding handler function in the module. Once defined, all these modules are then loaded into the Model Database.

Next, CLOWN accepts an input file describing how the network model is built up from the modules in the Model Database. CLOWN parses the file and creates the model in memory through the Model Builder. The Experimentation Manager then checks the file for validity using the entries in the Model Database. Once this has been verified, simulation can begin. CLOWN and D-CLOWN input files are discussed in greater detail in section 3.7.1.

Every simulation model is a specification of a physical system in terms of a set of states and events. Discrete event simulators like CLOWN have an Event List which chronologically stores states. Simulation progresses when the element at the front of the list is triggered, causing states to change and/or add new events to the event list. These are taken care of by the Event Handler subsystem.

Additional subsystems of CLOWN are as follows:

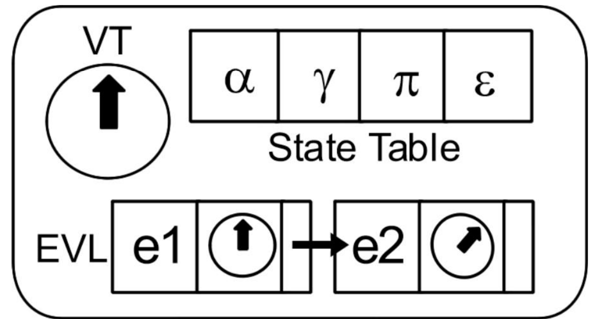

Figure 2: Parts of a Simulation Engine

- Message Handler - facilitates information exchange between network modules.
- Queue Manager - a library of different queue types (FIFO, Shortest Job First, etc...) for use by network modules
- Stochastic Manager - a library of different random number generators that return numbers according to a particular distribution. Clown provides a library for exponential, Weibull, uniform, and normal distribution.

Converting CLOWN into a distributed version necessitates some significant changes to its system architecture. For example, as network modules now reside on different computers, the Message Handler must be able to transmit messages not only locally, but also across a network. The Model Builder must be able to instantiate modules on remote computers. Finally, the Event List and Experimentation Manager must consider that some events are executed on remote computers.

2.2 Distributed Discrete Event Simulation

Discrete event simulation (DES) divides simulation into a set of events. These events are stored sequentially in an Event List (EVL). The simulation progress when the Simulation Engine (SE) chooses the next event from the EVL and acts on it, causing a change in the state of the model represented by State Table S and an advancement of Virtual Time (VT). Figure 2 shows an illustration of the simulation engine.

Distributed discrete event simulation divides the model into a set of logical processes (LP). Each LP represents a region of the simulation. This region could be a subset of the whole simulation in terms of its location in the system, or it could be a sub-epoch of the whole scenario. [2]

Partitioning the model requires that each region simulate a subset of the state variables. The model must be statistically divided into these regions taking into account that bad partitioning may cause the entire simulation to slow down due to message passing overhead (significantly slower over a network than with just shared memory). [6]

Each logical process has its own simulation engine, with a State Table representing the region the LP belongs to. The

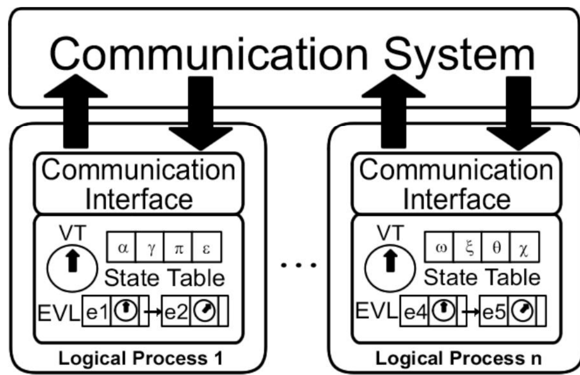

Figure 3: Distributed Discrete Simulation Engine architecture

Event List of the Simulation engine contains internal events, or locally generated events. And since events acting on the local state table may come from other LPs, a communication interface (CI) contains these external events. Synchronous LP simulation implements VT as a global clock while asynchronous LP simulation implements a local virtual time (LVT) which is the current time of that region of the simulation. Figure 3 shows a basic system architecture of a DES.

The simulation engine, as it does in a non-distributed DES, processes internal events in the event list or external events in the communication interface in chronological order, advancing its LVT and changing its state variables. However, as some events generated by the state change affect states in other LP's, the communication interface must be able to send these events via a Communication system.

Synchronization problems arise in asynchronous LP simulation when an event affects already executed events in other LP's, as these other LP's may have different local times. This is what is termed as a causality error. [2]

Conservative DES protocols avoid this problem by triggering the SE to only execute events that are considered "safe" to execute. Optimistic protocols, on the other hand, allow the SE to execute events regardless of local time in other LP's by allowing for a rollback of the simulation to a safe state if a causality error occurs. This feature is implemented in the CI for each protocol.

2.3 Conservative Logical Process Simulation

The Conservative Logical Process Simulation performs the simulation in a way that totally avoids causality errors by making sure that no events that cause this problem will be executed. This section discusses the initial work of Chandy, Misra and Bryant [7] and explores the Null Message Protocol which counters the problem of memory overflow and deadlock.

2.3.1 Overview

In a system of N local processes, a conservative LP uses the following data structures in its communication interface. Figure 4 illustrates this structure.

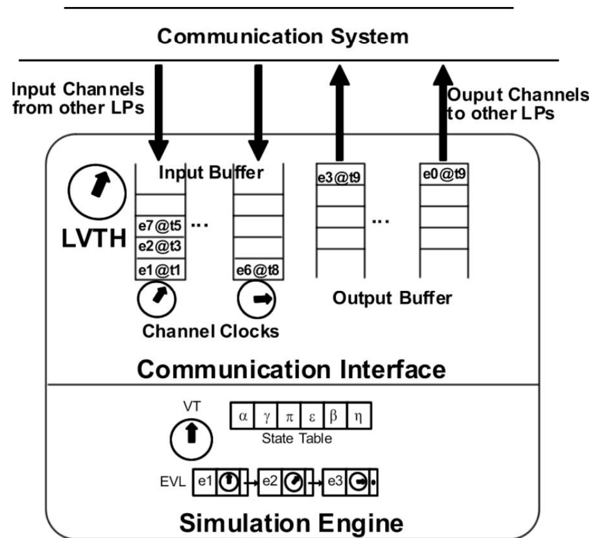

Figure 4: Conservative Local Process System Architecture

- Input Buffers IB[1..N] which queue all incoming event messages from LP[1] to LP[N]
- Channel Clock CC[1..N] which is set to the time of the element at the front of its respective input buffer.
- Output Buffers OB[1..N] which queue outgoing messages to LP[1] to LP[N]
- The Local Virtual Time Horizon (LVTH) which is set to the minimum of CC[ ].

Simulation still proceeds in the same manner as with non-distributed DES. The event with the lowest timestamp, whether from the input buffer or the event list, is chosen for execution, causing a change of state which generates either internal and/or external events. However, the LP must follow the following principles to guarantee that no causality events ever occur.

1. The LP processes events only up to an LVT for which it is guaranteed not to receive external event messages with timestamp in the future. This is called the local virtual time horizon (LVTH).
2. Events, both internal and external, are processed in chronological order. This guarantees that the sequence of external event messages produced by this LP will also be in chronological order.
3. The communication system preserves the order of messages sent, ensuring that all LP's receive messages in chronological order.

Given the LVTH, an LP processes local and external events until its LVT reaches LVTH. Upon reaching this state, the LP blocks until new external messages[1] arrive to extend the

[1]This paper will be using the terms "event" and "message" interchangeably

LVTH. As LVTH is the lowest timestamp of all incoming events, and that no out of order events arrive, it follows that an LP running with these principles does not execute events that have the potential to be out of order.

However, this blocking behavior is ripe for deadlock when LP's cyclically wait for each other. In addition, an LP that blocks may reach a point where local memory may not be able to buffer the backlogged events. The Null Message Protocol, discussed in the next section, discusses a systematic way to advance LVTH and avoid the blocking behavior of the basic protocol.

2.3.2 Null Message Protocol

In essence, an LP blocks when it has processed all events up to its LVTH. Only though external messages can its LVTH be extended. And external messages are only received by an LP when the simulation requires processing in that LP.

A relatively unused LP does not move past its LVTH, even though it has pending events, becuase it is not notified that other LP's have moved on with their processing until it gets an external message. The Null Message Protocol [7, 2] guarantees that all LP's are aware of the continuing simulation occuring in other LP's via the exchange of Null Messages.

All external messages exchanged by LP's come in the form `<ee@t>` where `ee` is a description of the event and `t` is the timestamp of the event based on the LVT of the LP where the event originated. A null message used in the null message protocol is a special external message in the form `<0@LVT>`. This null message has nothing to do with the simulation, it is used for the exchange of synchronization information.

If an LP does not have a transaction with some remote LP at a certain time T, then it sends this null message to the remote LP. This indicates to the remote LP that this LP will not be sending any other message smaller than the timestamp T, which can be used by the remote LP to advance its LVTH.

An optimization to this is to send the null message with timestamp T plus a lookahead value. The lookahead signals a time of concurrent events where two LP's can execute their events in parallel without having to consider causality errors at all. It can be seen that making the lookahead as far into the future as possible allows for more parallelism in the simulation execution and fewer null messages needed to be sent across the network. [8]

2.4 Optimistic Local Process Simulation

As described in the previous section, conservative local processes move in lock step to one another to ensure that no causality errors occur. This next section discusses the Time Warp algorithm [2, 9, 10], an optimistic process which simulate events regardless of the chance of causality errors by providing a rollback mechanism to recover from such events.

2.4.1 Time Warp Algorithm

An LP using the Time Warp Algorithm proceeds to advance its LVT, processing its internal and external events regardless of the local time of other LP's. When an external event occurs with time stamp less than LVT (affecting already executed events in the LP's local past), the LP rolls back simulation to the most recently saved state, and then proceeds a re-simulation, this time taking account of the straggler message.

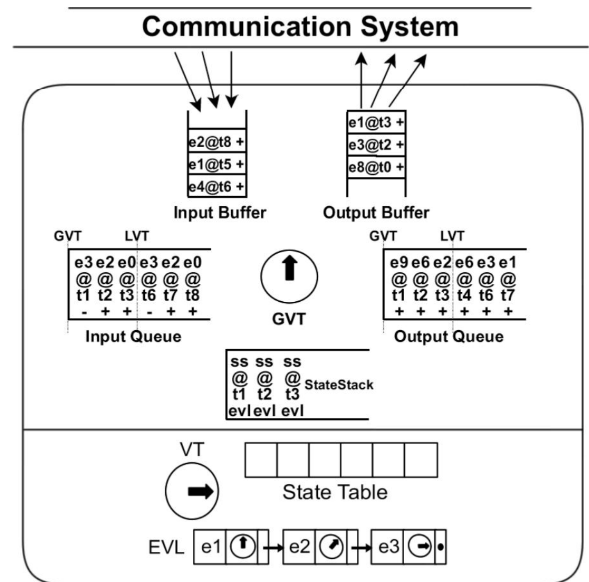

Figure 5: Time Warp System Architecture

As such, LP's using this protocol needs to be able to store state information. This includes not only the state variables and the event list at each LVT but both incoming and outgoing messages as well. The State of the system, consisting of the local state variables and event list at a given LVT, is stored in a State Stack (SS). In addition, incoming and outgoing messages are stored in an Input Queue (IQ) and Output Queue (OQ) respectively. Figure 5 illustrates this architecture.

It can be seen that a rollback not only involves a rollback in state, but also a rollback in the messages sent out by the LP, affecting other LP's. The next section discusses how Time Warp handles this scenario.

2.4.2 Messages and Anti-messages

Event exchange between Time Warp LP's still follow the same message form as in the Conservative LP. `<ee@t>` indicates an external event generated from some remote LP at a certain time T, which is the LVT where the event originated. Time Warp, however, adds an additional term to the event format to indicate whether it is a positive message `m+ = <ee@t,+>` or a negative or antimessage `m- = <ee@t,->`.

Positive messages m+ are the usual event messages being sent out by an LP to remote LP's. During rollbacks, antimessages m- are sent to cancel out the corresponding positive message (ala matter-antimatter reactions in the physical realm).

An LP responds to m+ and m- in the following ways:

- Messages m+ in the local future (timestamp > LVT) are processed the usual way: The message's event is placed in the input queue and is processed when the LVT reaches the timestamp of m+
- Messages m+ in the local past (timestamp < LVT) are straggler messages and would mean that a rollback is needed. State data from the LVT nearest m+'s timestamp is loaded into the state variables as well as the corresponding event list for that time.The IQ's LVT pointer is rolled back to take into account past external events, this time including the errant message. And for every message in the OQ that have been sent out (as it appears) erroneously, an antimessage is sent out in order to cancel the wrong message.
- If an antimessage m- whose corresponding m+ is in the local future, then m+ is annihilated from the input queue. As m+ hasn't been executed yet, no additional steps are required.
- If an antimessage m- whose corresponding m+ is in the local past, a rollback occurs as described above, and a re-simulation occurs with the m+ message cancelled.
- If an antimessage m- does not have a corresponding m+ in the input queue, it is placed in the input queue in order to annihilate the delayed m+.

Since antimessages can initiate rollbacks in other LP's, which in turn may send more antimessages in the system, then there is the chance of rollback chains and even recursive rollbacks. However, the protocol guarantees that all such events will eventually terminate, despite consuming a lot of time and communication resources. [2, 9, 10]

2.4.3 *Fossil Collection*

As the algorithm is described so far, the longer the simulation runs, the more memory resources are needed to store previous state information. A solution to this is to erase state information that is no longer needed by the simulation. Ferscha in [2] describes a strategy which defines a Global Virtual Time or GVT.

GVT is the minimum timestamp of the entire simulation. This comes from the least timestamp of LVTs of the system, as well as the timestamp of messages in transit. An LP only rollbacks if it receives a straggler message whose timestamp is less than that of its LVT. Since GVT is the timestamp of the simulation that is farthest back in time, it therefore follows that GVT is the farthest back in time that an LP can be rolled back.

It would follow then that events with timestamp less than that of GVT are irrevocably committed [2] and the LP can safely remove all state information from that time. This removal not only affects the State Stack, but also all outdated information in the Input and Output Queues.

Although there are many algorithms for GVT computation as described in [2, 11, 12] DCLOWN would only use a simple querying mechanism wherein an LP can ask other LP's for their least timestamp and computes its GVT given all the minimum timestamps. Although this is not an exact value for GVT, even suboptimal values of GVT provide fossil collection.

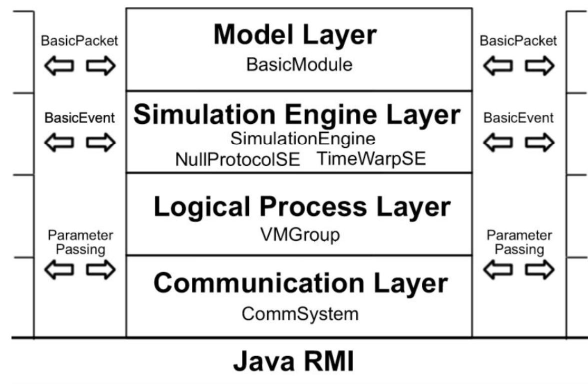

Figure 6: D-CLOWN system architecture

3. METHODOLOGY

This section is divided into three parts. First is the system architecture of D-CLOWN. Next is a description of how a simulation is run. The last part of this section is a discussion of D-CLOWN's performance on certain models.

3.1 D-CLOWN system architecture

D-CLOWN was implemented using a layered system architecture. Each layer abstracts the layer above it from implementation issues so that in the simulation layer the modules making up the simulation are not aware of the distribution of the model. Each layer is tackled in turn, discussing the interface methods it provides to the layer above it. This section concludes with a discussion of the classes in the topmost layer which are used by model programmers to create their simulation model.

3.2 Java RMI

D-CLOWN uses Java Remote Method Invocation (Java RMI). Similar to Remote Procedure Calls but in the Java application environment, Java RMI gives the programmer an illusion of a simple local method call to an object in reality existing in a separate system. Java RMI already handles socket creation, data exchange across the network, packaging parameters and returning values, eliminating the need for implementation by the application programmer.

The main remote object in DCLOWN is the Virtual Machine Groups (VMGroups), following the nomenclature of [13, 6] which will discussed in a later section.

3.3 Communication Layer

The communication layer attempts to abstract the upper layers from remote method invocation. All inter-VMGroup communication (and thus all D-CLOWN communication) are via methods of the CommSystem class which resides in this layer.

The CommSystem class maintains a table of references to VMGroups, both the local VMGroup, and remote references

to other VMGroups, all properly linked during simulation startup. In addition, this class has a PartitionTable, which stores the simulation model's distribution: which modules (indicated by module id) are on which hosts in the network.

3.4 Logical Process Layer

The VMGroup (borrowing from the naming convention of FATWa [13, 6]) is the core class of the Logical Process Layer. It is the backbone class of the D-CLOWN network. All hosts of the D-CLOWN network each run an instance of VMGroup. Simulation engine instantiation, message receiving, initialization of network modules are all handled by this group.

The Java RMI section talked about a server object that waits for clients remotely accessing its methods. In D-CLOWN, VMGroups act as these servers. To load VMGroup into memory, the user needs to run the executable class ClownServer, which registers VMGroup with a network-accessible name. Then this class waits for incoming method calls.

Although the CommSystem is on a layer below the Logical Process Layer, the CommSystem actually calls VMGroup methods, it mearly abstracts the upper layers from the RMI interface. The sendEvent method in CommSystem actually calls a particular VMGroup's receiveEvent method for example.

The VMGroup contains a RuntimeLibrary object. This object contains the simulation model itself: all the network modules that was loaded into this host are stored here. It also contains a reference to a SimulationEngine object. More on SimulationEngine objects in section 3.5

3.5 Simulation Layer

The Simulation layer holds the classes that are primarily responsible for running the simulation. In this layer, the SimulationEngine class runs the simulation without any synchronization protocols; it is a non-distributed Java version of the original Clown Simulation engine which is only used for single processor simulation runs. The TimeWarpSE class and the NullMessageSimulationEngine class are subclasses of SimulationEngine, which run a distributed simulation following the distributed discrete event protocols. The SimulationEngine that is loaded for this VMGroup is determined by the initializeSimulationEngine() method of VMGroup.

3.6 Model Layer

The topmost layer of D-CLOWN, the Model Layer holds the Network Model itself. Network modules are instances of BasicModule and all transmission between modules is through BasicPackets. Module interconnectivity is handled via IFEnv classes, following the original CLOWN implementation. State information is stored via ModuleData objects.

3.6.1 Model Layer Components

BasicModules represent modules in the model to be simulated. Model programmers would need to extend this class and implement certain abstract methods to determine the behavior of their model's BasicModules. One such method is the processPacket() method which will be discussed in a later section.

```
void processPacket(BasicPacket packet) {
  if (packet instanceof SendPktToServerPkt) {
    \\packet is a user-defined
    \\subclass of BasicPacket
    sendPacket(); \\direct method call
  } else if (packet.getHandler().equals("Handler")) {
    \\use BasicPacket's handlertext field
  } else if (in_interface.hasHandler(packet)) {
    \\ packet is processed by an IFEnv object
    in_interface.executeAction(packet);
  }
}
```

Figure 7: Different ways of handling packets

BasicModules also come with predefined methods to aid the model programmer. One such method is the sendPacket(double time, BasicPacket packet) method, which is called by subclasses when a module needs to transmit a packet at a given time.

BasicPackets are the main information exchange between modules in the Model Layer. BasicPackets store source and destination module ids, which allow for proper routing within the model. BasicPackets also contain methods that handle timestamping, which provide information for the model programmer on the behavior of the system.

Intermodule connection is handled via ClownInterfaces, or IFEnv classes. IFEnv classes in the original CLOWN implementation acted as an interface between two modules. IFEnv classes are discussed in detail in a later section.

State information in each module is stored via the ModuleData object. ModuleData objects store information via a `<key, value>` pair; key is a textual description of the field, value is that field's value. Module programmers can choose to use their own classes to store state data. However, using ModuleData or a subclass thereof allows for consistent information storage across the Model Layer and allows for retrieval and storage of data without knowing implementation specific information.

3.6.2 Handling packets

The BasicModule's processPacket(BasicPacket packet) method is called whenever a BasicModule needs to handle a packet. This is an abstract method; module programmers are required to implement behavoir for this method for their subclasses.

There are three ways for processPacket() to process a packet. First is via the use of BasicPacket subclasses and the Java `instanceof` operation. Second is via the handler string of a BasicPacket. Lastly, the module programmer can use the IFEnv EventPair class. See Figure 7 for an illustration of these three strategies.

The use of the IFEnv EventPair class warrants further discussion. In the original CLOWN implementation, every action is linked to a function which handles that action via function pointers[3, 5]. Function pointers allowed for a non-

```
:: Set global parameter ; Set SEngine ; Time Warp
:: Set global parameter ; Max message number ; 10000
:: Select a module ; ROUTINGAPPL ; 1
::     source id; 1
::     message arrival distribution ; exponential
::     mean time between messages ; 0.2
:: End of definition
:: Select a module ; ROUTINGRECEIVER ; 2001
:: End of definition
:: Connect two modules ; 1 2001
:: End of definition
```

Figure 8: D-CLOWN datafile highlighting the three main command types

```
1   ; 192.168.10.105
501 ; 192.168.10.106
1001; 192.168.10.110
2001; 192.168.10.111
```

Figure 9: D-CLOWN partition file

static function mapping: an action handler can change during runtime. Additionally, function pointers allowed modules to have a generic runAction command, which allowed for polymorphism despite being implemented in a non-object-oriented programming language. D-CLOWN improves on this by following the event handling paradigm of Java: each action has a corresponding actionListener which can be defined during runtime.

3.7 Model Building

This section discusses the sequence of steps that occur when D-CLOWN begins a simulation. In particular, this section talks on tackle the D-CLOWN input files, classes responsible for model creation, the process of setting up a distributed simulation, up to the beginning of simulation execution.

3.7.1 D-CLOWN input files

D-CLOWN accepts two input files, a data file describing the network model, and a partition file, which describes the distribution of the modules in the D-CLOWN network. Figure 8 shows an example of a CLOWN data file while Figure 9 shows an example partition file for use in D-CLOWN.

The datafile format has not changed from CLOWN to D-CLOWN. All commands in the datafile follow the format:
`::<commandtype>;<command name>;<additional params>`

The list of all possible commands is stored in the CommandList class. Model programmers seeking to create their own command sets must modify this class to include additional commands. The CommandList class converts a textual description of a command into a numeric value which is processed by the ModelBuilder. This allows for multiple textual descriptions of the same command. See Figure 10 for an example CommandList class definition.

3.7.2 Model Building

```
public static final int SOURCEID = 1048;
public static final int BANDWIDTHSHARE = 1049;
public static final int PROPDELAY = 1050;
...

private static Command commlist[ ] = {
   new Command("set source id"    , SOURCEID ),
   new Command("bandwidth share"  , BANDWIDTHSHARE),
   new Command("packet delay"     , PROPDELAY),
   new Command("propagation delay", PROPDELAY),
....
}
```

Figure 10: Excerpt of D-CLOWN's CommandList file

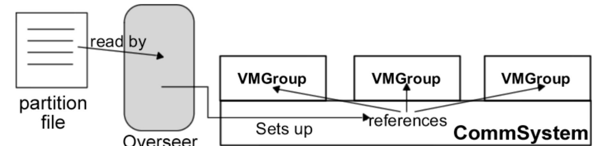

Figure 11: Setting up the D-CLOWN network

To begin simulation, the ClownCore class must first be executed. This would begin the process of parsing the input files, loading BasicModules into their respective VMGroups accross the network, and then starting the simulation.

When the ClownCore first starts, it creates an instance of the Overseer class, which is responsible for model building. First the Overseer class reads the partition file and determines which hosts in the network will take part in the simulation. The Overseer connects with these remote hosts and sets up the CommSystem. The CommSystem is responsible for the interchange of information across the simulation hosts. See Figure 11 for an illustration of this process.

After the Overseer sets up the network, the ClownCore creates a ModelBuilder object. The ModelBuilder class is a re-implementation of the old HPSIM_LoadModule() method of CLOWN, which is responsible for the creation of the simulation's Modules. However, unline CLOWN, D-CLOWN's ModelBuilder allows for the creation of Modules on remote hosts.

The ModelBuilder's primary task is to parse the data file. First it determines what BasicModules to load via a textual description in the data file. The conversion of a textual description of a BasicModule into an actual BasicModule object is handled by the BasicModuleFactory class. Its `BasicModule createModule(String moduletype)` accepts the string description, and then returns an appropriate BasicModule subclass. Model programmers must modify this class to allow their custom BasicModules to be loaded into the simulation model. Figure 12 shows an example of a BasicModuleFactory implementation.

Once an appropriate BasicModule has been initialized, ModelBuilder calls its editModule() method. This method is used in conjunction with the CommandList class in order to

```
class BasicModuleFactory {
  public BasicModule createModule(String moduletype) {
    if (moduletype.equals("ROUTINGAPPL")) {
      return new ROUTINGAPPL();
    } else if (moduletype.equals("ROUTINGSERVER")) {
      return new ROUTINGSERVER();
    } else {
      return null; \\ no such moduletype
}}}
```

Figure 12: BasicModuleFactory

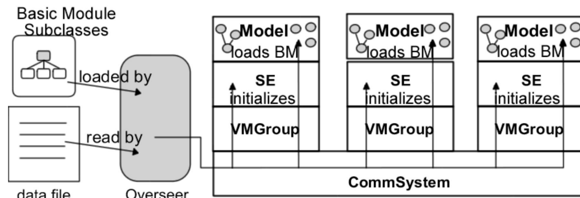

Figure 13: D-CLOWN simulation setup

properly initialize the BasicModule as defined by the data file.

In addition to BasicModule creation and initialization, the ModelBuilder also connects a BasicModule with other BasicModules in the simulation network. Also, the ModelBuilder is responsible for setting global parameters as defined in the data file.

Figure 13 shows a summary of this process of setting up the D-CLOWN system.

Once the datafile has been read and all the BasicModules have been loaded into their VMGroups, ClownCore gives a signal to the CommSystem to start simulation. Once a VMGroup receives this signal, it runs the initializeModule() method of each BasicModule in its RuntimeLibrary to set the initial state of the system. Then it flushes the output buffers to send the initial events caused by the initial state.

After waiting for a few seconds to finish initial event sending, the simulation begins in ernest, following the distributed DES protocols. Simualtion ends when a BasicModule calls the stopAllSimulation() method of the SimulationEngine. When this happens, VMGroups stop simulation as soon as possible, and call each BasicModule's finalizeModule() method for any last minute processing.

3.8 Implementation Issues

This section discusses issues that came up during the implementation of the distributed DES algorithms in D-CLOWN.

3.8.1 Starting Seed Problem

The original CLOWN implementation had as one of its global parameters an option to set the starting seed. This value was very important to the Stochastic Manager whose libraries had functions that returned random numbers following a particular distribution. Traffic generating modules are dependent on these functions to allow for a particular distribution of packets. Queueing theory defines theoretical metrics for a particular distribution of elements arriving at a queue, and the correctness of network models can be determined from these formulae.

Setting an initial seed allowed replication of what otherwise be an unreplicable simulation run. A traffic generating module will produce the same distribution of packets given the same initial seed.

This advantage came as a drawback in a distributed system. Two traffic generating modules produce different packet distributions on the same host because alternating calls to the Stochastic Manager produce different results. However, these same two traffic generating modules, when placed on two different hosts but with the same initial seed, call on the random number generating methods of ClownRandom in exactly the same order, resulting in *identical* packet distribution. When these two applications route packets to a single queue, the resulting distribution does not fall into the documented distribution categories: no theoretical metrics can be easily determined from such a system. Also, this would entail that a different simulation results would come out of different distribution schemes.

As a result, D-CLOWN does not support a single global simulation starting seed. A possible future implementation can allow for the specific assignment of starting seeds to each host. However, the purpose of assigning a starting seed to be able to exactly replicate an experiment does not occur in the system. The transmission of data over a network opens up variables such as transmission order, traffic collisions, physical data propagation delay, among others. Experiments on a distributed network become very hard to exactly replicate.

3.8.2 Simultaneous Events

Events that have the same timestamps are considered to be simultaneous events. Unlike regular events that are executed according to increasing timestamp, the order by which simultaneous events are executed can have an effect on the simulation. Consider a basic protocol where a node retransmits a packet if it does not receive an acknowledgement. If the retransmission event has the same timestamp as a just arrived acknowledgement event, then the order of execution of these two events will produce different results [14].

CLOWN simply uses an arbitrary [14] mechanism for simultaneous event tie-breaking, meaning it does not enforce any particular order in the execution of simultaneous events. Time Warp is also not affected adversely by simultaneous events and follows the same arbitrary mechanism as CLOWN.

However, simultaneous events have a great effect on the Null Message Protocol[2]. Thus, D-CLOWN using Null Message Protocol will first execute all simultaneous events before transmitting any resulting events to other parts of the simulation.

3.9 Test cases

This section tackles the different network modules used to test and give benchmarks for the performance of D-CLOWN. All tests were run on a 10Mbps network using four Pentium

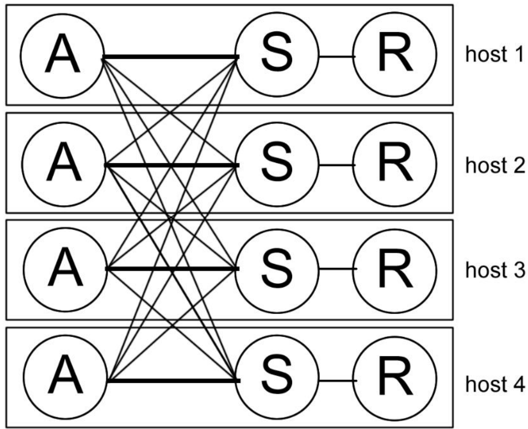

Figure 14: Test Model running on 4 hosts

4 computers running at 3.0GHz with 1GB of ram. These computers run Fedora Core 5 and run D-CLOWN on the Java Virtual Machine included in the Java v1.5.0_06 SDK.

The test network model is shown in Figure 14. This test network is made up of packet generating applications `A` that send packets at a defined rate to the servers `S`. Servers transmit data at a fixed rate to the Receivers `R`. If a packet arrives at a server, and the server is currently transmitting, that packet gets entered into a FIFO queue. The simulation model itself will be distributed among 4 computers.

Applications, however, have a preferred server (indicated by the solid line in Figure 14). At simulation start, all applications are given a weight value which indicates the percentage of packets sent to the preferred server. For example, an application whose preferred server weight is set to 30 sends 30 percent of its packets to the preferred server and the remaining 70 percent is distributed among the three other servers in the model.

4. RESULTS

This section discusses the metrics gathered from Java CLOWN and D-CLOWN runs using Null Message and Time Warp. The network load section shows the overall performance of D-CLOWN using the weighted server network model. Next metrics gathered from a degenerate version of the model will be discussed.

4.1 Network load test

This section describes the results of running the weighted server model on Java CLOWN and D-CLOWN. Applications send exponentially distributed packets (with a mean of 2 packets / second) to the server, which can send out packets to the receiver at the rate of 100 packets / second to the receiver. This test varies the preferred server weight from 0 to 100 percent with a model distribution shown in Figure 14. As can be derived from the illustration, the model running at 0 percent server weight sends all of its traffic to other hosts in the simulation, while the model running

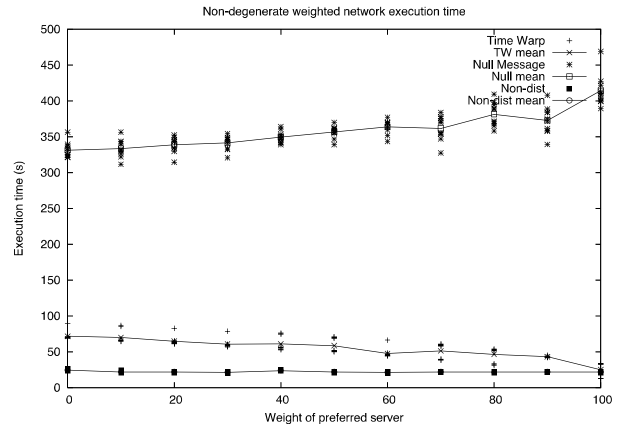

Figure 15: Non-degenerate weighted server execution times

at 100 percent server weight reduces to a completely parallelized simulation. Figure 15 shows execution time versus preferred server weight.

It can immediately be determined from Figure 15 that Null Message performs an order of magnitude slower than both Time Warp and non-distributed CLOWN. Time Warp itself also performs slower than the non-distributed version, as transmission of events over memory is significantly faster than over the network.

It is of note that the performance of Null Message decreases as the system becomes more parallelized. In a highly decoupled system, Null Message moves forward only at the rate of its lookahead. A null message with lookahead is sent to a target LP when no external events are to be sent to that LP. If the system is completely parallelized, then no external events are sent to any system except for null messages, which advance the LVT of all systems by lookahead each time the model is run. Contrast to that coupled execution, where frequent messages to remote LPs advance LVT with a timestamp greater than that of the lookahead.

4.2 Degenerate test

In the degenerate case, server input exceeds its output, thus queue lengths increase indefinitely. This puts an added load to the system in terms of memory allocation and additional processing for all the extra packets alive in the model. This section discusses the behavior of D-CLOWN under such a situation, which also gives an indication of how D-CLOWN performs in similar models with high processing load.

In this test, applications still release 2 packets per second on an exponential distribution. However, server bandwidth is now reduced to just 1 packet per second. The network modules are modified to print to terminal instead of to file to add processing load to the simulation execution. Figure 16 shows the results of the degenerate test.

With all twelve modules being processed by a single processor, and the backlog of printing to the terminal, simulation execution time of the non-distributed version is slowed down

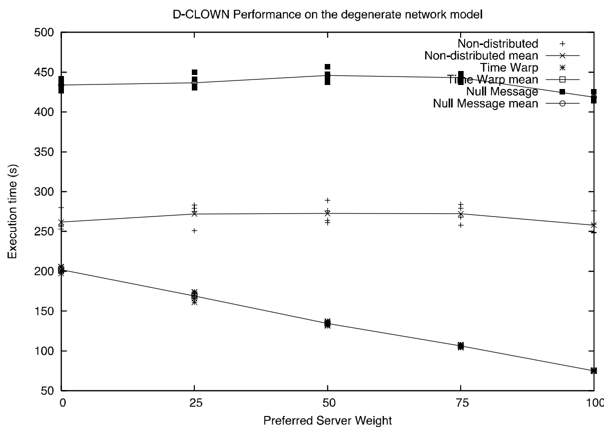

Figure 16: Execution times of D-CLOWN on the degenerate model

considerably as can be seen in Figure 16.

Null Message performs with a low variance among the different weights, taking advantage of parallel processing. While a single host must process all incoming packets, thus leading to simulation delays, Null Message, albeit slow, manages to use distribution to divide processing load among several hosts, thus avoiding the bottleneck of event processing and terminal screen printing.

On the other hand Time Warp performs spectacularly, running at a faster rate than that of the non-distributed runs at the high parallelization of the higher preferred server weights. Time Warp takes advantage of the model's parallelization just like Null Message but does not have the drawback of having to broadcast null events which delay simulation.

5. CONCLUSION

This paper has explored the implementation of distributed discrete event simulation algorithms to the non-distributed Concatenated LOcal and Wide-area Network simulator. This paper discussed the conservative Null Message protocol and the optimistic Time Warp algorithm and addressed issues regarding the implementation of CLOWN into the object-oriented Java programming language as D-CLOWN.

D-CLOWN provides a simple way of instantiating network modules on remote hosts and abstracts inter-host communication, providing for seamless view of the whole simulation model at the level of the model programmer, and also providing an API to assist model programming. In addition, D-CLOWN users can choose between running their simulation on a single host or to run their distributed simulation using either Time Warp or Null Message.

Distributing network simulation raises some issues. This paper has explored the need for the synchronization protocols to keep the distributed system running as a single unit. Also, a serious setback to simulation execution time is the significant delay of network message transmission to that of shared memory, topped off with a high variance in execution times due to network transmission uncertainties. The loss of a global initial seed is a severe hindrance, particularly for simulation debugging, as distribution precludes exactly replicable results.

However, D-CLOWN shows promise in certain simulation models, particularly those with high model parallelism and processing load. Time Warp was shown to perform better than the non-distributed version in such a case, overcoming the network transmission lag time and produce faster results.

6. RECOMMENDATIONS

Literature is full of research on optimizations to distributed discrete event simulation algorithms, and it is recommended to explore the implementation of these into the D-CLOWN framework. Another possible avenue of improvement is to port D-CLOWN into a version that runs directly on the underlying processor instead of a virtual machine. Although this implementation loses some of the portability that Java provides, increased simulation speed may come out of it. Graphical add-on packages to D-CLOWN, such as a way to visualize the currently executing simulation, or a click-and-drag system for model creation would be a welcome feature in future D-CLOWN implementations.

7. REFERENCES

[1] Anand Kuratti. Improved techniques for parallel discrete event simulation, 1993.

[2] Alois Ferscha and Satish K. Tripathi. Parallel and distributed simulation of discrete event systems. Technical Report CS-TR-3336, 1994.

[3] S. Sørensen. Clown: An object oriented simulation environment. *Proceedings International. ASME Conference Modelling, Simulation and Control*, pages 2018–2024, 1992.

[4] S. Sørensen and M.G.W. Jones. The clown simulator. *Computer and Telecommunications Performance Engineering*, 1991.

[5] Cedric Festin. *Utility-based Buffer Management Scheduling for Networks*. PhD thesis, Universtiy College London, 2002.

[6] Matthew C. Lowry, Peter J. Ashenden, and Ken A. Hawick. Distributed High Performance Simulation Using Time Warp and Java. Technical Report DHPC-084, 2000.

[7] Jayadev Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.

[8] Jorn W. Janneck. Generalizing lookahead - behavioral prediction in distributed simulation. In *Workshop on Parallel and Distributed Simulation*, pages 12–19, 1998.

[9] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[10] D. Jefferson and H. Sowizral. Fast concurrent simulation using the time warp mechanism. In *Proceedings of the SCS Multiconference on Distributed simulation*, pages 63–66, 1985.

[11] Behrokh Samadi. *Distributed simulation, algorithms and performance analysis (load balancing, distributed processing)*. PhD thesis, 1985.

[12] Global virtual time algorithms. In *Proceedings of the Multiconference on Distributed Simulation*, 1990.

[13] Matthew C. Lowry. FATWa: A testbed for time warp in java. In *Proceedings of the Seventh Integrated Data Environments - Australia (IDEA'07) Workshop*, pages 47–49, 2000.

[14] Vikas Jha and Rajive Bagrodia. Simultaneous events and lookahead in simulation protocols. *Modeling and Computer Simulation*, 10(3):241–267, 2000.

[15] David M. Nicol and Richard M. Fujimoto. Parallel simulation today. *Annals of Operations Research*, (53):249–285, 1994.

[16] Chandy and Lamport. Distributed snapshots: Determining goal states of distributed systems. *ACM Transactions on Computer Systems*, 3(7):63–75, 1985.

[17] P. Dickens and P. Reynolds. SRADS with Local Rollback. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 161 – 164, 1990.

[18] R. McNab and F. Howell. Using java for discrete event simulation, 1996.

[19] Jeff S. Steinman. Breathing time warp. In *PADS '93: Proceedings of the seventh workshop on Parallel and distributed simulation*, pages 109–118, New York, NY, USA, 1993. ACM Press.

[20] M. Gerla, K. Tang, and R. Bagrodia. Tcp performance in wireless multi-hop networks, 1999.