The Distributed Open Network Emulator: Applying Relativistic Time

Craig Bergstrom

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
Computer Science And Applications

Srinidhi Varadarajan, Ph.D., Chairman

Godmar Back, Ph.D.

Calvin Ribbens, Ph.D.

May 15th, 2006
Blacksburg, Virginia

Keywords: Network Simulation, Network Emulation, Simulation Time, Scalable Simulation

The Distributed Open Network Emulator: Applying Relativistic Time

by
Craig Bergstrom
Committee Chairman: Srinidhi Varadarajan, Ph.D.
Computer Science

(ABSTRACT)

The increasing scale and complexity of network applications and protocols motivates the need for tools to aid in the understanding of network dynamics at similarly large scales. While current network simulation tools achieve large scale modeling, they do so by ignoring much of the intra-program state that plays an important role in the overall system's behavior. This work presents The Distributed Open Network Emulator, a scalable distributed network model that incorporates application program state to achieve high fidelity modeling.

The Distributed Open Network Emulator, or DONE for short, is a parallel and distributed network simulation-emulation hybrid that achieves both scalability and the capability to run existing application code with minimal modification. These goals are accomplished through the use of a protocol stack extracted from the Linux kernel, a new programming model based on C, and a scaled real-time method for distributed synchronization.

One of the primary challenges in the development of DONE was in reconciling the opposing requirements of emulation and simulation. Emulated code directly executes in real-time which progresses autonomously. In contrast, simulation models are forced ahead by the execution of events, an explicitly controlled mechanism. Relativistic time is used to integrate these two paradigms into a single model while providing efficient distributed synchronization.

To demonstrate that the model provides the desired traits, a series of experiments are described. They show that DONE can provide super-linear speedup on small clusters, nearly linear speedup on moderate sized clusters, and accurate results when tuned appropriately.

Dedication

This thesis is dedicated to all those who helped me along the way. In particular my mother and father whose love and support has always driven me to do my best. Also, Dr. Back and Dr. Varadarajan whose help in completing this work was invaluable.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Distributed applications are inherently difficult to develop. When a high degree of scalability is required, the difficult nature of distributed application development is exacerbated. Understanding the implications that a decision will have in a large scale deployment is not only important but also difficult. This thesis presents The Distributed Open Network Emulator, or DONE, a tool to aid in the understanding of network dynamics at a large scale. DONE is based on a distributed discrete event simulation core that drives an emulation layer. The emulation layer interacts with compiled application and protocol code. Although application code used in DONE requires no source-level modification and runs natively, even non-reentrant code modules can be run many times within a single process space without error. This provides an unprecedented level of scalability for a network emulator and rivals even the most scalable network simulators. The primary advantage that DONE has over other network simulators is that unmodified protocol code is incorporated as a part of the model. As a result, protocol code that has been validated within DONE can be moved to a production network with little or no modification. The net effect is a high degree of assurance that protocol code will perform as expected on real networks.

This project relies on several previous efforts. The Weaves framework [18] provides a new programming model compatible with C but tailored specifically for DONE's requirements. It allows the instantiation of multiple global name-spaces within a single virtual address space. This contributes significantly to the potential scalability of the solution. To allow application protocol code to interact with a network in a familiar fashion, the LUNAR protocol stack [15] is integrated into the simulation. LUNAR provides a user level implementation of the BSD sockets interface. It was isolated from Linux early in the 2.4 development cycle and introduced only a minimal set of modifications. This insures that the simulator's protocol stack acts much like a widely used implementation. An additional contribution to DONE was a non-distributed network simulator with similar goals of executing actual protocol code inside an simulation-emulation hybrid.

One of the primary hurdles in developing DONE is in the integration of simulation with emulation. This need, along with the requirement for scalability mandates a new model for distributed synchronization. This new model of simulation time, relativistic time, is a scaled real-time mechanism that coordinates the execution of many distributed entities. It relies on the uniform progression of real-time within a single frame of reference to coordinate the modeling of a vast number of simula-

tion entities and reduce the quantity of messages required to coordinate their execution. In addition to the use of real-time, relativistic time introduces a number of logical clocks designed to model the time-state relationships that exists internal to emulated entities. The result is that the quantity of messages required to coordinate simulation execution scales with the number of physical processors instead of the number of virtual hosts, an improvement of several orders of magnitude.

A number of tests support the claims that DONE scales well and provides accurate results. These tests have shown super-linear speedup on small clusters, and nearly linear speedup on moderately sized clusters. Additionally, TCP connections established using the LUNAR protocol stack provide expected throughput numbers on a number of simulated bandwidths and latencies.

1.1 Motivation

In recent years, there has been an explosive growth in the use of computer networks. This has allowed users to manipulate data and communicate regardless of their location or the location of the resource they are accessing. Paralleling the rapid growth in adoption of communication networks has been an equally rapid growth in the scale and complexity of networked applications and protocols that drive those applications. The world wide web is a perfect example of this. It was initially made up of a relatively small number of static web pages and was used as a way for a few scientists and engineers to share information. Today's complex web based applications are numerous and widely used. They are often run on clusters of computers, generate data on-the-fly, and incorporate multimedia content. As network protocols and applications continue to be applied to more diverse problem domains, they can only be expected to become more complex.

As the scale of a distributed application grows, its development can also be expected to become more difficult. To illustrate this, consider the methods used for their development. Interfaces for Message passing, such as MPI or BSD sockets, allow applications to pass data between distributed processes. Complex application level protocols use these libraries to exchange information. Encapsulation allows higher level protocols (ie. application level protocols) to be developed without knowledge of the inner workings of lower level protocols (ie. transport and network level protocols). This is beneficial as it allows developers to decompose large problems into more manageable ones. However, it does little to help them understand the implications that their decisions will have on the performance of the system. For small applications intended for isolated use, issues of scale are not critical. However, for protocols and applications intended for widespread use on large networks, scalability is a critical issue.

Developing protocols and applications that perform well requires a way for developers to evaluate the protocol's performance in the environment for which it is intended. There exist several options for this, but all have significant drawbacks. The most straightforward mechanism is to actually construct a system and conduct tests on it. The results of the tests can be used to motivate performance improving modifications. The obvious problem with this method is that an unproven system must be constructed before it can be evaluated. Though useful for small systems, for large scale systems it can be prohibitively slow and expensive.

Hardware emulation is a useful mechanism for evaluating the performance of a network application under various conditions, but is not useful for scale testing. To perform hardware emulation,

physical hosts are connected through an emulator which then models the desired network characteristics. Jitter, packet loss, and out-of-order delivery are among the characteristics that can be modeled. Applications are then run on the physical hosts to evaluate a protocol's performance under the various conditions. Though useful for testing application performance in realistic scenarios, this approach has several drawbacks. Primarily, it is impossible to model a network with capabilities exceeding that of the emulator, physical interconnection, or processor pool. For example, a network emulator can slow the delivery of packets on a 1Gbps physical link to model a 100Mbps link, but cannot accurately model a 10Gbps link without faster hardware. This is useful for application testing, but offers little help in the development of next-generation protocols for which hardware is not yet available. Additionally, hardware emulators do not aid in scale testing. To illustrate, consider scale testing an application for deployment on a 100,000 node network. First, a user would be need to purchase 100,000 nodes, wire them together through a number of emulators, configure the emulation hardware, and then run the application. Although it is useful for developers to test protocols on large networks, this method is infeasible.

For scale testing, simulation is preferable to emulation. A simulation begins with an abstract model of a system and performs a series of transformations on that model to determine the state of the system at a later time. Unlike emulation, it does not necessarily interact with real entities.

Many network simulations are available. Although each has its own benefits and drawbacks, some characteristics of the paradigm as a whole make it well suited for scale testing. In particular, the fact that simulation acts on abstract models allows it to be designed with the desired degree of fidelity to solve the desired problem. The implicit trade-off is that as the fidelity of the simulation is increased, the computational cost of the simulation grows. If some part of a system is expensive to model but does not contribute in a significant way to the result obtained, it can be abstracted away and replaced with a simpler and less expensive model. This allows a simulation to model a very large system at a much lower cost than an emulator.

Of particular interest here is the way that time and state relationships are modeled by simulation. In any physical system, the progression of time implicitly parallels the progression of the systems' state. Alternatively, a simulation forces the progression of its clock by processing events that affect simulation state. Time, therefore, is explicitly controlled by the processing of those state-altering events. This is beneficial because it allow a simulation to control how its resources are allocated with respect to the virtual time-line. Many resources (a 1Mbps link, for example) have an associated modeling cost per unit time (Requiring the transfer of 1 megabit of information per second virtual time). Although simulation cannot control the quantity of resources afforded to it by its environment (per unit real-time), it can control the mapping of real-time to virtual-time. Therefore, even though (per unit real-time) a simulator's resources are bounded (a distributed simulator may run over a 100Mbps link, for example), it can speed up or slow down the rate of event processing to obtain the required resources per unit virtual-time.

Although the fact that a simulation operates on an abstract model allows efficient allocation of real-time resources to virtual-time, it is not entirely beneficial. Often, models built for simulation must be developed in simulation-specific languages. These often differ significantly from the languages used to create production protocols. This requires costly and error prone re-implementations when moving protocols from model to production. In practice, this can result in a protocol whose

performance characteristics differ from the model that was validated. This, in turn, negates the benefit of using a simulation in the first place. Previously, simulations have been constructed that allow modeling of large networks, but they do not provide for the integration of actual application software into the model (direct code execution)[8]. Direct code execution models have been constructed, but they fail to scale well [11]. This work presents a network simulation-emulation hybrid intended to leverage the benefits of direct code execution (via software interface emulation) while taking advantage of the scalability and efficiency benefits of simulation. Specifically, the goal is to build a network modeling environment tailored specifically towards scale testing. The combination of a scalable network model with direct code execution is beneficial because it could provide a high degree of assurance that a protocol validated as scalable is implemented as such. Additionally, it could provide an environment for developers to construct scalable algorithms so they could easily be put into production.

Complicating the development of this protocol development environment is the fact that the current generation of simulation coordination mechanisms do not integrate well with emulation or scale sufficiently well for its purposes. The primary challenge in integrating simulation with emulation is in the reconciliation of the real-time nature of direct code execution with virtual-time nature of event based simulation. For this purpose, a new mechanism for scalable time-line management was developed. The relativistic time model coordinates distributed discrete-event simulation with network interface emulation in a scalable way. It takes advantage of the self synchronizing properties of real-time without negating the advantages of simulation's controllability. By leveraging this controllability, the model provides for an emulator that is able to gracefully handle virtual time intervals of low load by executing quickly as well as virtual time intervals of high load by executing slowly.

1.2 Related Work

1.2.1 Simulation History

The use of computers for simulation and modeling extends back almost as far back to the first computers themselves. Initially, scientists used simulation to quickly evaluate systems for which analytic models were too slow, inaccurate, or difficult to evaluate. These early systems were built in low level languages and sometimes without an underlying operating system. It quickly became obvious that constructing complex models in this way was unnecessarily difficult. Major advances in simulation came in the form of operating systems, high level programming languages, and eventually specialized simulation programming languages, or SPLs. These advances spawned the development of far more complex simulations and reduced development time.

Initially simulations were used for modeling manufacturing systems, performing weather prediction, and for military applications [24]. Soon after the advent of packet based computer networks, the advantages of using a simulation to evaluate the design of these systems became evident. The need to evaluate larger and more complex systems in less time has always motivated the need for ever faster computers and algorithms. More recently, the advent of vast worldwide networks has exacerbated this need, motivating very large network models that run very quickly.

In the past, sequential systems have relied upon faster hardware to improve their performance. Even with this crutch, applications demanding to be on the leading edge of the performance curve have relied on parallel architectures to achieve faster results to more complex problems than their sequential counterparts. More recently, heat dissipation has become a bottleneck in the production of faster processors. The effect is that developers can no longer rely on dramatic increases in clock speed to solve more complex problems faster. Instead, parallel architectures provide a means to improve performance. Parallel architectures cannot operate efficiently on serial algorithms, however, and instead require algorithms designed for multiple threads of execution. This motivates the need for parallel algorithms to increase application performance. This work is directed towards fulfilling this need for parallel algorithms by providing a new mechanism for event synchronization in a large parallel and distributed simulation. The evaluation section shows that the mechanism works as expected in the context of a distributed network simulator.

1.2.2 Logical Processes & Event Ordering

Widely cited in simulation literature and of primary concern here is logical process, or LP, simulation. The state of a logical process simulation is divided into N disjoint subsets, or objects, $O_1, O_2, ...O_n$. Similarly, transformations on the state of a LP simulation can be divided into N disjoint subsets, $T_0, T_1, ...T_n$. such that T_i represents the i^{th} object's set of transformations and $T_{i,j}$ represents the j^{th} transformation on the i^th object. LPs do not have direct access to each other's state, but instead communicate by sending messages which cause state transforms upon reception. When a message is received it causes some state change in the receiving process. Associated with each transformation, $T_{i,j}$ is a send time, $st(T_{i,j})$, and a reception time, $rt(T_{i,j})$ such that $st(T_{i,j}) < rt(T_{i,j})$. The state of an object is not affected by a transformation until it's reception time, and a message can be affected only by events whose reception times are less than its send time. Each object, O_i, has an associated time-line upon which all transformations are ordered. Events must be created and processed in time-stamp order to ensure the proper result. More specifically, for all i,j,k if $st(T_{i,j}) < st(T_{i,k})$, $T_{i,j}$ was created before $T_{i,k}$. If $rt(T_{i,j}) < rt(T_{i,k})$, then $T_{i,j}$ must be processed before $T_{i,k}$ is processed. If $rt(T_{i,j}) < st(T_{i,k})$, then $T_{i,j}$ must be processed before the creation of $T_{i,k}$ and $T_{i,j}$ can affect $T_{i,k}$. Furthermore, all events affected by $T_{i,k}$ can be transitively affected by $T_{i,j}$.

There is a directional distance, $D_{n,m}$, between any two objects, O_n and O_m, such that no transformation, i, caused by a message sent from object O_n to object O_m at time $st(T_{n,i})$ can have a reception time less than $st(T_{n,i}) + D_{n,m}$. Two transformations occurring on the state of O_n and O_m with time-stamps $rt(T_{n,i}) < rt(T_{m,j}) + D_{n,m}$ are said to be independent and can be processed in any order with respect to each other. Event independence is fundamental to parallel simulation, as it allows the processing of events simultaneously.

In order to ensure the desired outcome, every simulation must ensure that events are processed in the correct order. Processing two non-independent events in the incorrect order (a temporal inversion) must be avoided. For sequential implementations, ensuring proper event ordering is trivial. These algorithms typically consist of a series of iterations in which the lowest time-stamped event in the simulation in processed. Determining an ideal event ordering is complicated by parallelization. In addition to executing events in the correct order, parallel algorithms must identify a sufficient quantity of events as independent and process those events concurrently to justify the added com-

plexity of building a parallel implementation. To further complicate matters, the identification of event independence must be done without consuming excessive resources. Because of this, all parallel implementations rely upon some quickly computable heuristic to identify event independence. These algorithms can be broadly classified into two categories, optimistic and conservative.

1.2.3 Optimistic versus Conservative Models

Optimistic algorithms such as the time warp protocol [13] allow every simulation node to process events without bound until a causality error is detected. When this occurs, the simulation is 'rolled back' to the time of the error so that it can be corrected. This method has the advantage that it is able to fully utilize the computational resources of its simulation processors. However, unless the load is carefully balanced between simulation nodes, many compute cycles end up being used to roll back the state of entities that executed too optimistically. In certain problem domains where roll backs are inexpensive and load balancing is easy, optimistic protocols provide good results. However, in the case of directly executed code, rollback requires distributed check-pointing and recovery, which significantly increases rollback cost and complexity.

Conservative algorithms prevent causality errors from occurring by carefully controlling the speed at which events are processed. Algorithms such as Chandy-Misra-Bryant's [7, 17, 6] typically rely on explicitly passing messages about the state of nodes to decide when it is safe to process an event without causing an error. More specifically, to determine which events are safe to process, each node must have a lower bound on the time-stamp of messages that can arrive from all other nodes. This constraint is maintained by observing that all incoming messages from each LP arrive in time-stamp order with respect to other messages from the same LP. Therefore, the lower bound on all incoming messages from each LP is the time-stamp of the previously received message from that LP. Relying on previously received messages is prone to deadlock, so to avoid this, null messages are passed between LPs. This algorithm is limited by the fact that it requires that all communication paths in the LP topology to be known at the start of the simulation. Additionally, the number of null messages used for coordination increases as a function of the quantity of LP topology links crossing physical node boundaries. This results in an excessive number of coordination messages on large simulations and limits the ability of a simulator to decompose a simulation to produce faster results.

A number of mechanisms have been proposed to improve the performance of the conservative null message algorithm. These include attempts to decrease the number of messages passed between logical processes [21], exploiting temporal uncertainty in distributed systems [9], attempts to use a conservative lookahead window [2], and the recognition that for certain application domains (e.g. Virtual Environments) the precise ordering of events is less important than efficient and highly responsive execution [10]. Despite these improvements, conservative algorithms still incur high communication overhead and achieving good speedup on large systems is difficult.

1.2.4 Parallel Simulation of Networks

As in other simulation domains, parallel architectures are used for both speeding up the evaluation of and improving the potential scale of network simulations. Both space parallel and time parallel approaches are applicable, with space parallel approaches leading to more flexible solutions.

Time parallel approaches operate by dividing the simulation time-line into intervals, simulating each interval independently, and then 'patching' together intervals by modifying the places where the intervals intersect. If intervals are sufficiently long, independent, and patching is sufficiently fast, good speedup can be achieved. However, matching intervals can lead to inconsistent states and is difficult for direct execution simulation. Space parallelism, in contrast, is applicable to simulations where matching time intervals is not trivial, such as emulation. Space parallel simulation divides a large pool of simulation entities into a number of smaller 'chunks', and simulates those chunks in parallel. It allows scaling to large virtual processor counts because per-virtual-host memory can be distributed over, instead of replicated over, physical processors. The primary difficulty in space parallelism is in coordinating the execution of a large number of entities, especially in distributed simulation.

Parallel simulating of networks presents some unique challenges. Short lookahead windows plague physical layer simulations due to quick event propagation between entities. Wireless simulations that include mobile end hosts are one example that includes particularly poor lookahead windows [16]. The fine granularity of events (individual packet transmissions) and the frequency that events cross the physical node boundary additionally complicate space parallel approaches [22]. Additionally, the wide array of uses makes building any single general purpose network simulation tool difficult. Instead, most simulation tools attempt to fulfill some subset of the needs of the entire community. To understand the dynamics of particularly large networks at a high level of abstraction, fluid flow simulators have been developed. These fluid flow simulators typically capture only statistical characteristics like packet loss, jitter, and latency to aid in the comprehension of network dynamics at a high level. They do not accurately capture details at the packet-level because of the abstractions made. Packet based network simulations, in contrast, model the behavior of networks at the packet level, but do not typically scale as well as fluid simulations.

No single approach to parallelizing of network simulations has achieved widespread acceptance. Some efforts have started with successful commercial simulations and added parallelization and distribution into the already existing framework [25, 20]. Space-parallel approaches typically involve building 'ghost' or 'proxy' objects which take the place of virtual hosts whose computation is hosted elsewhere. The ghost object proxies all requests destined for a remote network object and ensures that the requests are delivered to the correct virtual host. The advantage of this approach is that users familiar with the sequential version will find transitioning to a parallel version easier than transitioning to a completely new environment. The disadvantage is that design choices that were made for sequential implementation may complicate parallelization or negatively affect performance.

Other network simulation tools are based on languages tailored towards parallel simulation. GloMoSim, for example, is a parallel network simulation library based on the parsec simulation language [26]. Using a parallel simulation language or library saves development time (and cost) but may not be fully optimized for the exercised cases.

Other simulations have been constructed 'from the ground up' in systems or general purpose programming languages. The advantage to this approach is that it is highly flexible and can achieve good performance. However, it comes at the cost of a comparatively slow and difficult implementation.

Although many high level design decisions have been explored in the development of network

simulation, many optimizations have been proposed to improve the performance of already existing techniques. The general observation to be made is that higher accuracy simulations are slower to evaluate [5]. However, many optimization techniques have been shown to provide good results. Examples include include packet aggregating techniques which amortize event processing time over multiple packets [1], using hybrid packet/fluid flow simulations [14], and exploiting the uncertainty of event timing [3].

Chapter 2

Relativistic Time

Although time management can be viewed as a just one component of a larger system, it often motivates the design for much of the rest of a simulation. This implies that it should be carefully designed in a way to allow the rest of the simulation to execute as efficiently as possible. Before considering the specifics of Relativistic Time, or any other distributed synchronization protocol, it would be useful to consider the nature of time in a distributed system. This will aid in the understanding of exactly what is and is not permissible for a synchronization algorithm.

2.1 Resources and Time

Although no single definition of time is universally agreed upon, for our purposes it is monotonically increasing and ordinal in nature. The passage of time is not itself measurable. Instead, observers measure the occurrence of events and make inferences on temporal relationships by the comparison of two (or more) events. For example, an entity can observe the movement of the hands on a clock and infer that a certain interval of time has passed. The clock (an entity itself) measures time by observing the oscillations of a crystal or other periodic waveform.

To achieve a high degree of configurability and efficiency, it is desirable to control the rate at which virtual time progresses. This will allow controlability over the rate of resource consumption for entities in a simulation. This, in turn, will allow a model to gracefully handle both resource under-subscription and over-subscription. To understand this, consider that many resources are provisioned at a certain rate (networks, for example, are described in terms of bits per second). Therefore, slowing the progression (in real-time) that entities execute provides additional resources per unit virtual time. Real-time is unobservable to entities in a simulation. Therefore, this change in execution rate is also unobservable, provided that all entities are affected in the same way. A relativistic time simulation utilizes this property by processing events faster when the simulation is undersubscribed, and slower when oversubscribed. The rest of this chapter discusses how the relativistic time model is put to use in a distributed multiprocessor simulation.

2.2 Introduction to Relativistic Time

Simulations run in virtual-time, which is forced ahead by event processing. In contrast, emulators run in real-time, which progresses autonomously. Constructing a simulation-emulation hybrid requires a reconciliation of the two sets of opposing requirements provided by simulation and emulation. Relativistic time seeks to integrate the requirements of these two paradigms in a scalable manner.

Like any distributed simulation time management scheme, relativistic time seeks to efficiently determine a correct ordering of events. It is directed towards high-fidelity models where rollback costs are prohibitively high. It, therefore, avoids processing an event until it can be ensured that the processing of that event will not violate causality. Specifically, causality is violated by the processing of any two events that act upon the same state in reverse time-stamp order.

An additional goal of relativistic time is to reduce the number of synchronization messages required to coordinate execution. It does so by taking advantage of the nature of emulation. In particular, the quantity of time required to execute a single instruction is constant. *Because the quantity of computation (and time) required to directly execute an emulated instruction (and hence, advance the emulated entity's clock) is bounded, we can determine a lower bound on the progression of all emulated components clocks.* The uniform progression of real-time across a distributed simulator allows us to define this lower bound as a scaled real-time function. The result is that all nodes in a distributed simulation have (and gain) knowledge of the temporal state of other nodes in the simulation without explicitly passing messages. By using this lower bound wherever possible, a simulation can reduce the quantity of messages required to synchronize with distributed nodes. To accomplish this goal, we introduce a hierarchy of clocks used to coordinate the execution of emulated and simulated components.

2.3 Hierarchy of Clocks

In order to directly execute a large number of emulated entities on a relatively small number of processors, a relativistic time simulation time-multiplexes the execution of entities' events. During intervals when an entity has control of the CPU, its state is modified. In reality, this state change occurs over some interval of time. In the model, therefore, its view of time should also advance when the state change occurs. The state of other entities in the model remains unchanged during this interval, and so their views of time should remain unchanged as well. In order to model this correctly, the time of each emulation entity is accounted for independently and advanced each time its state changes. Keeping a single clock per entity is sufficient to account for all the entities in the simulation, but it is unclear how to synchronize the entities' executions. In order to accomplish this goal, relativistic time uses global relativistic time, an artificially imposed clock that does not exactly depict any single entity's view of time.

2.3.1 Global Relativistic Time

When a simulation begins, The global relativistic clock is set to some predetermined initial value (usually zero), and progresses at some fraction of the rate of real-time. This fraction is defined such

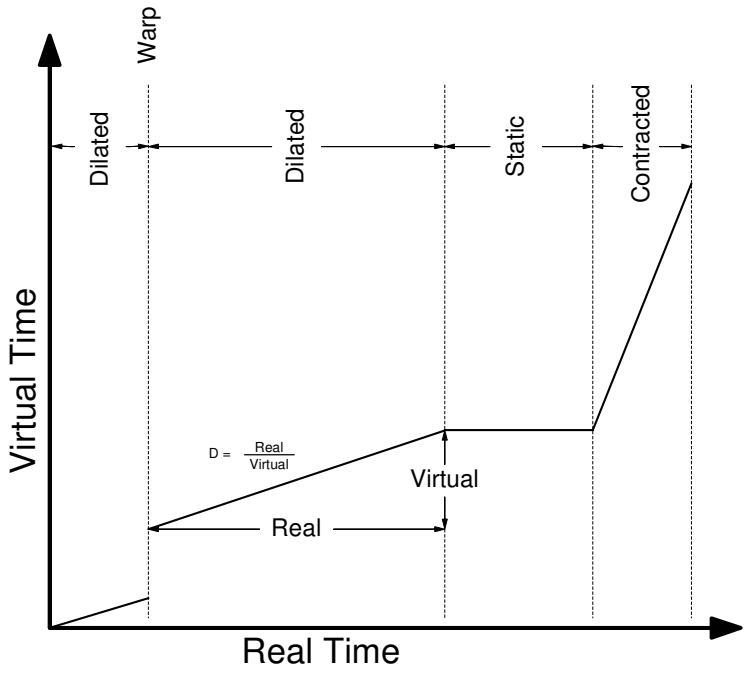

Figure 2.1: Virtual time (Global relativistic time) is shown here as a function of real-time in a relativistic time simulation. The virtual clock (GRT) goes through periods of dilation and contraction, which slow down and speed up the simulation respectively. Static intervals (the result of out-of-band, computationally expensive, or non-deterministic models) result in periods of real-time in which virtual clock time not progress. Inversely, warps consist of periods in which the global relativistic clock progresses instantaneously through some interval of simulation time. In practice, dilation and static periods are used to provide additional time to compute expensive models. To speed the evaluation of a simulation, contracted periods and warps are used.

that $GRT = (1/d)t + C$ where C is the initial value, d is the dilation factor given by the simulation input, and t is the length of real-time that the simulation has been running.

Defining the GRT in this way has several consequences. It allows the the simulation to define how fast the GRT will progress with respect to real-time. As a result, computationally inexpensive simulations can allow the GRT to run faster than computationally expensive simulations. In other words, the amount of computational resources per unit global relativistic time can be tailored to the specific needs of the simulation being performed.

To speed up the execution of relativistic simulations, relativistic time uses the warp operator. The warp operator instantaneously advances the global relativistic time by some constant value and is used during periods in which no state changes occur.

Figure 2.1 shows the relationship between real-time and global relativistic time (virtual time). A simulation progresses through intervals of dilated operation, contracted operation, static operation, and warps. The time-line is slowed as compared to real-time during dilated intervals, runs faster than real-time during contracted intervals, does not progress during static intervals, and skips intervals of virtual-time when warping. These states correspond to setting the dilation factor to values on

the intervals $[1, \infty)$, $(0, 1)$, ∞, and 0 respectively.

2.3.2 Local Relativistic Time

Each node in a distributed relativistic time simulation keeps track of its own *Local Relativistic Time*, or LRT. This LRT is simply a local copy of the global virtual time-line. Care should be taken by the designer of a simulation to ensure that all LRT's remain as closely synchronized as possible.

Due to both the skew in clock rates between two machines and the latency inherent in any network communication, the value of any two LRT clocks may diverge. To counteract this divergence, a relativistic time simulation dilates relativistic time sufficiently to achieve the desired degree of synchronization. Since further dilation of LRTs' slows their progression with respect to real-time, it mitigates the effect of LRT divergence.

To illustrate this point, consider the case of two physical hosts starting their LRTs' at the same point in virtual time but one second apart in real-time with dilation factor ten. The LRTs' will diverge by one tenth of a second (virtual-time), even though they started their clocks a full second apart (real-time). If, instead, a dilation factor of one-hundred is used, then the virtual times will diverge by just one hundredth of a second. This property can be used to provide more accurate results at the cost of requiring additional simulation time and overhead for model evaluation.

2.3.3 Local Time

Event processing is time multiplexed across many individual entities in a large simulation. Each time an entity's state is modified, its time is advanced to reflect the relationship between its changed state and the advancing of time during that change. Due to the fact that entities' executions are time multiplexed, changes to two different entities that occur at the same virtual-time may or may not occur at the same instant in real-time. A *local clock* or *local time* is kept for each entity in the simulation and is synchronized with the entities' state. The simulation engine is responsible for ensuring that entity state and local time is advanced in a safe way (does not violate causality).

2.3.4 Global Relativistic Time & Local Time

By advancing each entities' clock independently and according to to the rate at which its state changes, relativistic time ensures an accurate view of the time-state relationship internal to an entity. However, it must also provide a mechanism to support synchronization between simulation entities. As the number of emulated hosts grows, individually synchronizing each of these clocks with every other clock by direct comparison becomes an expensive proposition. This problem is aggravated by the fact that entities are distributed over any number of physical processors. Scalable synchronization is provided by comparing local clocks to the relativistic clock via the local relativistic time-lines. Before this comparison is useful, however, the GRT must be assigned a meaning.

To give the GRT meaning, we ensure that all local clocks remain greater than or equal to the GRT at all times during the simulation. If we enforce this assertion, then any entity (and the simulators' scheduler) can use an inexpensive comparison to GRT to discover information about every other entity in the simulation without querying each individually. The advantage that this strategy has over other conservative synchronization mechanisms, like Chandy-Misra-Bryant's, is

that global relativistic time advances without explicit message passing. The result is that fewer synchronization messages need to be passed to coordinate execution, thus reducing overhead.

Enforcing a system wide lower bound on local clocks results in a similar lower bound in event creation time. By making the assumption that there is a minimal latency between event creation time and event reception time, we can obtain an event horizon. The event horizon is the point in virtual time at which it can be guaranteed that no new events will be generated. Since we defined GRT as a minimum over all entity times, we can quickly obtain a conservative estimate of the event horizon by computing $GRT + MinimumLatency$. It is therefore safe to assume that by scheduling events with time-stamps less than $GRT + MinimumLatency$, we will produce the correct result.

2.3.5 Using Relativistic Time

Using real-time as a distributed shared frame of reference has the potential to reduce communication but introduces certain requirements. First, it requires that the advancing of entity's clock must have a bounded cost. Otherwise, relating the progression of entities times to real-time is meaningless because there is no way to guarantee that this relationship will remain valid. Additionally, it adds the complexity of additional clocks to the simulation. Developers must be aware of the multiple clocks, and when each should be used.

Associated with each entity in the simulation is its state, some process which acts upon that state, and its clock. These clocks, states, and processes must remain precisely synchronized with each other. This implies that the arrival and creation of events should be performed according to the clock of the entity receiving or sending the event respectively.

Event based simulation requires that events be processed in the correct order. To make this guarantee, a relativistic time simulation schedules event processing according to GRT. Because GRT is shared among all simulation processors, events processed by different processors at the same (or nearly the same) time must be independent and will not affect each-other. Some mechanism should be in place to ensure that all events are known by the time GRT reaches their time-stamp.

To summarize, a relativistic time simulation requires that all events are processed at their reception time according to *both* the GRT and the receiving entity's local clock. Enforcing this will ensure both that events act on the correct state and are synchronized with remote entities. The mechanism used to accomplish this should domain specific, and so it will not be discussed here. How this is accomplished for DONE will be discussed in the chapter that focuses upon its implementation.

Chapter 3

Foundations for The Distributed Open Network Emulator

This chapter introduces DONE and some of the work that formed a basis for its implementation. Although it does not directly discuss the novel aspects of this work, it does provide an introduction to the preliminary work that allowed DONE to be constructed. An introduction to the goals of DONE provides a basis to motivate the need for each of the aspects discussed here.

3.1 The Distributed Open Network Emulator: Goals

Many tools are available to parties interested in modeling both small and large scale networks, but all have drawbacks. DONE seeks to fill a gap in available network simulation tool kits. More specifically, this work attempts to develop a network simulation-emulation hybrid that scales to hundreds of thousands of virtual network hosts across a small to moderately sized cluster computer, is capable of modeling a variety of network types, directly executes application code, and emulates the same software interface as widely used operating systems.

To achieve a high degree of scalability, DONE utilizes a distributed architecture and a new approach to synchronization, relativistic time, which is explored in the previous chapter. Emulation is used to allow application protocol code to run without modification, and simulation is used to integrate an abstract network model. This work hopes to show that this hybrid approach achieves a degree of scalability that is unachievable with current generation emulators with an acceptable degree of accuracy.

3.2 Weaves : Virtual Host Namespace Isolation

It is advantageous for a direct code execution network model (emulator) to allow multiple virtual hosts to execute in isolation. Hardware emulators separate virtual hosts by isolating them on separate physical hosts. Software emulators, like X-Sim [5], have successfully isolated virtual hosts by running them in distinct process spaces. Neither approach is particularly scalable. Weaves [18]

provides the ability to instantiate multiple independent copies of a single code module within a single process space. Each independent code module is then assigned a distinct global namespace, even though they all share a single virtual address space. This provides the desired mechanism for virtual host isolation while maintaining the ability of virtual hosts to share data. This, in turn, provides a scalable mechanism for virtual host seperation.

Weaves patches a programs twice, once at compile time, and again at run time. The effect of these transformations is the creation of multiple global namespaces, each referring to its own copies of each application's global variables. The creation of a unique global namespace for each application instance effectively isolates each virtual host from every other virtual host. By ensuring that any references to global variables of two distinct virtual hosts are mapped to different locations in memory, we achieve re-entrancy from potentially non-reentrant application and protocol code. We can therefore run legacy applications multiple times within a single simulation without introducing the overhead associated with multiple process spaces. Most importantly, this re-entrancy does *not* come at the cost of requiring source code modification, a slow and potentially error-prone process.

3.2.1 Build Process and Implications

Although building a DONE simulation is not entirely different from the process as done for normal C applications, using Weaves does introduces some notable differences. This process is described here to aid the reader in understanding how DONE operates, differences between typical C applications and DONE, and how the weaves build process affects things for DONE (starting and managing MPI, for example). The reader should be aware that this discussion is specific only to the supported compiler, linker, and loader and may not apply to other compilers, linkers, or loaders.

During building, the simulation must provide information to Weaves on how to organize its data to model the desired network. The data organization takes place partially at compile time, partially at link time, and partially at run time.

During compilation, source files are transformed into object files. Next, those object files are linked into larger modules before the final linking against the simulation core creates an executable. These aggregate modules represent the finest level of granularity available weaves' namespace separation. the reasons are discussed later in this section.

The goal is to isolate all references to global variables from inside each virtual host and point the references to different locations in memory. This will require both copying global variables to unique memory locations, and ensuring references to those variables point to the copied memory.

During compilation, global data is separated into the .bss (uninitialized global data) and the .data (initialized global data) sections. As modules are linked together, all elements of .bss and .data are collected together and inserted into the output object or executable file *in the same order that they were specified on the command line.* This is important because it is relied upon while weaving our simulation.

During loading, a chunk of memory is allocated for the .bss section and (may be) zeroed. The .data section is similarly allocated, and initial values are copied from the executable ELF file. These transformations, performed during loading, preserve the order in which global data elements are linked. In other words, within each of these sections (.data and .bss), variables are loaded in the same order as their object files were specified to the linker's command line. Weaves takes advantage

of this by inserting dummy variables, inside both the .bss and .data sections, between each module for which we will provide namespace isolation. By taking the address of these dummy variables, Weaves is able to isolate each of the memory segments allocated for each modules' global memory segments. The modules' global data segments are copied after loading to provide multiple values for each per-host global variable.

In addition to copying all global memory to new locations, all references to global variables must be resolved to the new locations. Weaves takes advantage of the way that global variables are referenced by gcc-2.96 to do this. All code generated by gcc-2.96 refers to global variables by first looking up the location of that global variable in the Global Offset Table, or GOT. The global offset table is simply an array of pointers to the global variables. References to global variables look up an index into the GOT and then refer to the pointer located at a specified index. On the x86, the ebx register points to the start of the GOT. By creating a new GOT with modified values, and modifying the contents of the ebx register, Weaves executes native code generated by gcc-2.96 and modifies the ebx pointer to point to different global namespaces depending on the global namespace desired by the executing thread.

The actual implementation of this namespace replication first copies the entire GOT once for each virtual host and utility thread in the simulation. Since we have a pointer to the beginning and end of every module's global data (provided by the partition dummy modules linked against at link-time), it then iterates through each newly copied global offset table, testing each variable to determine which module it resides in. If the module to which the variable belongs has been copied, references to that variable should be isolated. So, the pointers in the new GOT are then modified to refer to the data copy. Finally, a thread is started for each virtual host utility thread, and its ebx register is modified to point to the new GOT that has been patched to refer to his global namespace. Since all registers (the ebx included) are automatically copied by the operating system's context switching code, the host is then ready to run, but within its own modified global namespace.

Users should be wary of dynamically linked libraries. They are loaded at run time, and are therefore unknown when the GOT is created (even for normal non-weaved applications). To resolve global variables in dynamically linked libraries, a different global offset table is created, and the ebx is pointed to that libraries' GOT each time a function is called for that library. In fact, each time that *any* function is called, the ebx is set to point to the GOT used by that function. To ensure that this ebx-setting code does not modify which GOT table is referenced by a weaved application, all code executed within a weave must be patched. This is accomplished by a small patching program which is run passes over assembly code output by every compilation but prior to assembly. The primary implication of this is that dynamically linked libraries are incompatible with Weaves. You can call into them, but they cannot be weaved like normal C programs. Also, if a dynamically linked library uses ('clobbers') the ebx pointer for some reason, but does not reset it to its previous value before returning, corruption will result.

Compilation of Weaves applications differs slightly from typical C programs. First, they must be compiled with gcc-2.96 using the -S flag. This stops the compilation before assembly so that the assembly code may be modified by the weaves patch. The patch is then run over the code and which modifies the way that the ebx register is used. Specifically, the ebx register is set to point to the beginning of the global offset table at the beginning of each function call. The patch removes

these instructions so that modifications to the ebx register at thread start up are persistent for the duration of each thread's execution. To reiterate an important point, the implication of this is that if the ebx register is modified due to some library call or signal handler call, it must be restored to its intended value before referencing any global value. Additionally, signal handlers must set the ebx to an appropriate value before referencing global variables. Failure to follow these rules may have unexpected results.

After complication and patching, the patched assembly file is then assembled to create an object module, and linked to create an executable. Once run it must be patched again to modify data structures created by the loader. Most of the details of copying global data and creating multiple global offset tables (discussed earlier) are handled by the Weaves API. Starting MPI(which is required for DONE), however, is not handled by weaves and needs to be done in a compatible manner. Specifically, MPI must be started *before* global data copying occurs so that any copied data reflects changes made by MPI's start up procedures. Additionally, the MPI specification states that MPI can make use of signals, so signal handlers must be modified to configure the ebx to point to a sane location for MPI.

3.3 LUNAR : BSD Socket Emulation Layer

The network simulation core emulates a network interface for virtual hosts. This virtual network interface (or OS personality layer) calls into the application each time that a message is received, and every 10 ms (virtual-time) to replace the OS timer interrupt. Additionally, it provides a mechanism for virtual hosts to put messages 'on the wire'. Application programmers do not typically interface with the network at such a low level. To provide an interface familiar to application programmers, DONE includes LUNAR, a TCP/IP network protocol stack based on the Linux 2.4 kernel but configured to run in user space [15].

LUNAR links against the simulation engines' virtual network interface and provides a BSD sockets interface to virtual applications. Because the interface of LUNAR closely resembles the interface of the in-kernel Linux protocol stack, applications require minimal modification to port from Linux to LUNAR. For example, just like Linux's in kernel IP protocol stack, LUNAR provides socket operations and is configured through sysctl()'s. Applications interfacing with LUNAR's "top half" link against a family of entry points defined in lunar_socket.c. Unlike the majority of the protocol stack, these entry points were rewritten because the existing definitions interact with the kernel running on the host operating system.

LUNAR's philosophy is to use existing source code instead of writing new code wherever possible. The result is that the idiosyncrasies of Linux's protocol stack are reproduced in LUNAR. This is important because protocol stacks sometimes diverge from protocol specifications. One example of this is the length of time that a protocol stack remains in the CLOSE-WAIT state. RFC 793[19] requires four minutes, but the Linux stack waits for only sixty seconds[15]. Applications that rely on this property in Linux can therefore take advantage of it in LUNAR as well.

One commonly used RPI (underlying communication) layer for MPI applications is the TCP/IP stack of the host operating system. MPI interfaces with the host operating system's protocol stack through the same BSD sockets interface that LUNAR presents to virtual applications. It is desirable

to leave this interface exposed to MPI without interfering with host applications interfacing with LUNAR. To prevent namespace collisions between LUNAR's socket calls, and the c-library's socket calls, DONE uses namespace mangling techniques. Through a set of macros, all socket calls from virtual applications are re-mapped to names which will not collide with libc's standard definitions. Similarly, LUNAR's functions are defined with the altered names. To link against LUNAR's sockets interface, an application includes Linux/lunar_socket.h, which defines these macros and automatically remaps its socket calls. Alternatively, internal components (like the message passing interface) can link against the operating system's socket interface by omitting this header.

3.4 Kernel and User space Time-line Accounting

To ensure that virtual hosts' clocks are kept consistent with the quantity of time spent executing (and hence, their state), high precision CPU resource usage tracking has been added to the Linux kernel. The idea is to keep track of the quantity of time that each host has spent using the CPU. During periods that a host is running, its clock is defined in terms of its CPU resource usage.

To keep track of a virtual hosts' CPU resource usage, each time a virtual host thread begins using the CPU, the time that it started using the CPU is stored in the kernel. Each time the thread yields the CPU, the time that it yields the CPU is subtracted from the time that it started to use the CPU. The difference between these two values is added to a counter stored in a per-thread data structure. This counter, therefore, always contains the total quantity of CPU time that each thread has utilized.

To maintain this data, two fields are added to each task structure in the kernel, one for the running interval start time and another for the tasks' total elapsed time (over all previous intervals). These fields are updated by code segments added to Linux's schedule() function and use the get_cycles() function as their real-time clock.

The use of the get_cycles() interface to return the current time has several implications. It provides the highest resolution clock available to software, but it is hardware dependent. get_cycles() relies on the processors tsc register, whose resolution is the same as its clock speed. For example, on a one gigahertz processor, this interface provides a nanosecond resolution clock. To achieve a uniform interface across processors with varying clock speeds, DONE converts the high resolution values returned by the kernel's resource usage interface to nanosecond resolution values at the kernel space-user space boundary. The value used for this conversion is set by the user at compile time through the SIMULATOR_CYCLES_TO_NS macro. This value should be set to the clock speed of the CPU core. An automatic tuning routine is provided to isolate the user from the need to tune the simulator for their machine.

Application code does not need to concern itself with the internal workings of kernel time accounting. Instead, an interface, the get_time() function (not to be confused with the in-kernel get_cycles() function), is provided for applications to query the simulator regarding their current simulation time. This interface is kept consistent by intercepting the virtual host each time it becomes runnable and keeping track of its running time. When the application queries the simulator for its current time, the simulator determines the correct simulation time and returns that value to the application code. Similarly, the rest of the simulation engine is isolated from the internals of kernel time accounting

through encapsulated interfaces. The current global relativistic time, for example, is returned with calls to get_rel_time() and is updated with update_rel_time().

Chapter 4

The Distributed Open Network Emulator

The Distributed Open Network Emulator is built on top of MPI, a message passing library that targets high performance parallel applications. It provides both bootstrapping and communication for DONE. Interacting with the MPI layer is a connection component which manages message receipt for distributed event processing and simulation time management. Linking and loading is jointly handled by gcc-2.96, Weaves, and ld (the Linux loader). Simulation event processing components rely on the communication layer and the linker/loader components. Event processing supports the emulation layer by modeling the virtual network. The emulation layer provides an interface for protocol code which, in turn, interacts with applications directly. The overall system design is shown in figure 4.1.

4.1 Distributed Event Processing Core

At the core of the simulation engine is a distributed discrete event processor. This event processing engine runs in dilated global relativistic time and provides a mechanism to distribute the processing of events across a distributed memory multiprocessor.

At start up, DONE statically assigns sets of virtual network hosts to physical simulation nodes. A virtual host consists of state (memory) and a set of transformations on that state. Both of these are hosted by the processor assigned to that host. To facilitate communication between virtual hosts, the distributed event processor provides an encapsulated mechanism to schedule events destined for other virtual hosts. This event layer is further isolated from virtual host applications by the emulation and protocol layers. The protocol layer, LUNAR [15], interacts with the simulator through the emulation (or OS personality) layer. The emulation layer plays the part of a virtual network driver and interrupt subsystem by delivering message receive and timer interrupt events to the protocol stack. When the emulation layer requests event creation, the event processing layer delivers the event to the destination virtual host by passing it across MPI and storing it on the destination processor. An Event is stored in a heap until it can be scheduled. Once schedulable, it is removed from the

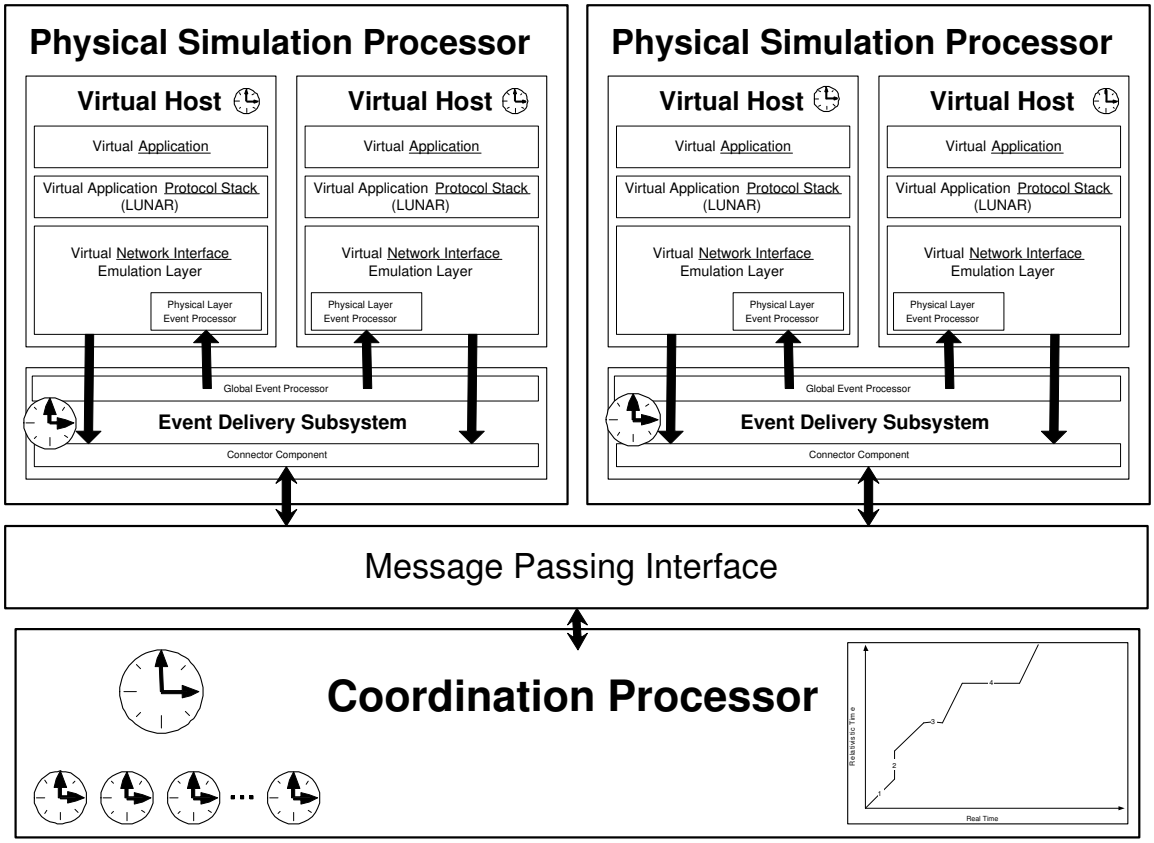

Figure 4.1: High Level Design Of DONE showing two simulation processors (above) and one coordination processor on the bottom. The message passing interface (MPI) interacts primarily with the event delivery subsystem and coordination processor to ensure correct event delivery. Virtual applications, shown in more detail in figure 4.4 consist of the virtual application, protocol stack, and emulation layer (which contains a physical layer event processor). The emulation layer interacts with the event delivery subsystem to ensure proper event delivery. Each virtual host has its own clock, as well as a global event delivery clock for each processor's event delivery subsystem. The coordination processor is responsible for collecting information regarding each of these clocks and coordinating global warps.

event heap and passed to the virtual host for processing. Each virtual host has a per-host event processor which then performs the necessary state transformations dictated by the newly arrived event.

4.1.1 Event Core Timing & Lookahead

Event scheduling is based on the assertion that a simulation need not schedule events at the absolute time of their time-stamp. Instead a simulation will provide the desired result if (a) events are processed in the correct order and (b) virtual time is correctly presented to the virtual host. This assertion is supported by the fact that an event will produce the correct result if the proper state exists when it is processed. The proper state is provided by ensuring that all events are scheduled only when it can be ensured that all previous events have been processed. By induction, and because the initial event is scheduled first, all previous events must therefore have been scheduled in time-stamp order. Using this rule, the event scheduler initiates event processing once it can be guaranteed (by the safe lookahead window) that the scheduling of a candidate event will not be followed by the discovery of another event with a lower time-stamp.

As discussed in detail in the relativistic time chapter, a safe lookahead window is provided by ensuring that (a) global relativistic time remains a system-wide lower bound over all clocks and (b) that there exists a virtual time latency between event creation and delivery. In terms of network simulation, this lookahead window is provided by the latency inherent in all network communication. In this discussion we will refer to the minimal latency between the time any host sends a message (schedules a remote receive event) and when that message should be delivered (the event's reception time-stamp) as λ. Therefore, and because GRT is greater than all other time-stamps in the system, no receive message event will be created with a delivery time less than $GRT + \lambda$. Events whose source and destination reside on different physical processors may need to be passed over a physical network. If the physical network is classified by a maximal latency, δ (specified in real-time) and this is factored into the lookahead window, no event will be delivered to a destination event processor with a delivery time less than $GRT + \lambda - \frac{\delta}{d}$, where d is the relativistic dilation factor.

By increasing the dilation factor, d, we widen the simulation lookahead window, at the cost of slowing simulation time. This becomes more clear if $GRT + \lambda - \frac{\delta}{d}$ (which is specified in virtual time) is rewritten in terms of real-time: $d(GRT + \lambda) - \delta$. Leaving λ constant (using the same virtual network), the duration of real-time allocated to processing a particular lookahead window increases linearly as the dilation factor is increased. A larger scaling factor, therefore, results in a more accurate simulation but comes at the cost of increased synchronization overhead. This overhead manifests itself as a greater number of warp events.

Once an event is deemed schedulable by the global event processor, it is passed to a per-host event processor. Implicit in this delivery is the knowledge that events delivered to the physical event processors are correctly ordered. Since this per-host event processor need only concern itself with the events of a single virtual host, it can perform more costly operations without fear of interfering with other virtual hosts. For this reason, it is assigned the task of running application protocol code assigned to event delivery. More specifically, it runs the receive message, and timer interrupts each time a message is delivered, and every 10ms relativistic time.

4.1.2 Local Time-line Management

Simulation entities(virtual hosts) are always either running or blocked. When running, their associated application thread resides in the simulator's operating system scheduler's runnable queue and may be scheduled for execution at any time. Virtual hosts are permitted to continue running uncontrolled and in real-time until some emulation event causes them to become blocked. For example, if a virtual host calls a blocking receive on its network stack and no data exists, then the host is taken off the runnable queue and remains blocked until data arrives. During runnable intervals, the virtual host's clock can progress arbitrarily far ahead of the rest of the simulation (and GRT). When it becomes blocked, it is taken off the operating system's scheduler's runnable queue. Its state can be modified by the simulation scheduler (which differs from the operating system's scheduler) during these blocked periods (to account for a message reception, for example). Events are delivered according to the global relativistic time, so once the GRT arrives at the time of the event responsible for unblocking the virtual host, it is restarted.

In addition to blocking receive calls, a virtual host is artificially blocked whenever it attempts to access the protocol stack. This is required because the state of the protocol stack is subject to modification from events initiated by any virtual host with which it shares a network link. Since a virtual host's clock can progress ahead of GRT (and nodes with which it shares a network link), accesses to the protocol stack must be explicitly ordered with respect to all potential externally initiated events. This is accomplished by blocking the virtual host each time it accesses the protocol stack, and not restarting it again until the global event processor (GRT clock) 'catches up' with the virtual host's local clock. This occurs when the safe lookahead window (based on global relativistic time) reaches the value of the virtual host's clock at the time of the attempted access. In the meantime, the event processing engine transparently (to the virtual host) updates the state of the protocol stack. It is then guaranteed to be in the correct state for the virtual host to access it.

Management of the local clock of a virtual host is accomplished by keeping track of when (in virtual-time) virtual hosts begin running and how much CPU time they utilized during their runnable intervals. A virtual host progresses through alternating periods of runnable and blocked and may only become runnable due to the processing of an event (with an associated time stamp). When a virtual host becomes runnable, the virtual time of its unblocking is taken from the time stamp of the event and noted. It then keeps track of its CPU utilization (in nanoseconds) during the running interval. To obtain its current virtual-time at any point that it is running, the simulation adds the amount of CPU time utilized during that running interval to the interval's start time. Negating overhead and processor sharing, this should be identical to the quantity of real-time that the same program would take on an unshared processor.

To understand the relationship between local time, global relativistic time, and real-time more clearly, see figure 4.2. Global relativistic time (shown as the long bold arrow) progresses at a constant rate at all times (except when warping). Once all nodes become blocked, the relativistic time-line is updated to the time of the next unblock event (as shown by the dashed vertical lines). Virtual hosts (shown as shorter arrows, one host as bold and one not) are chosen to run by Linux's operating system scheduler. Ideally, they would be chosen in round robin fashion. An incorrect ordering of events is not possible until one falls behind global relativistic time. The progression of virtual host time-lines is defined in terms of the quantity of computational time allocated to the host's

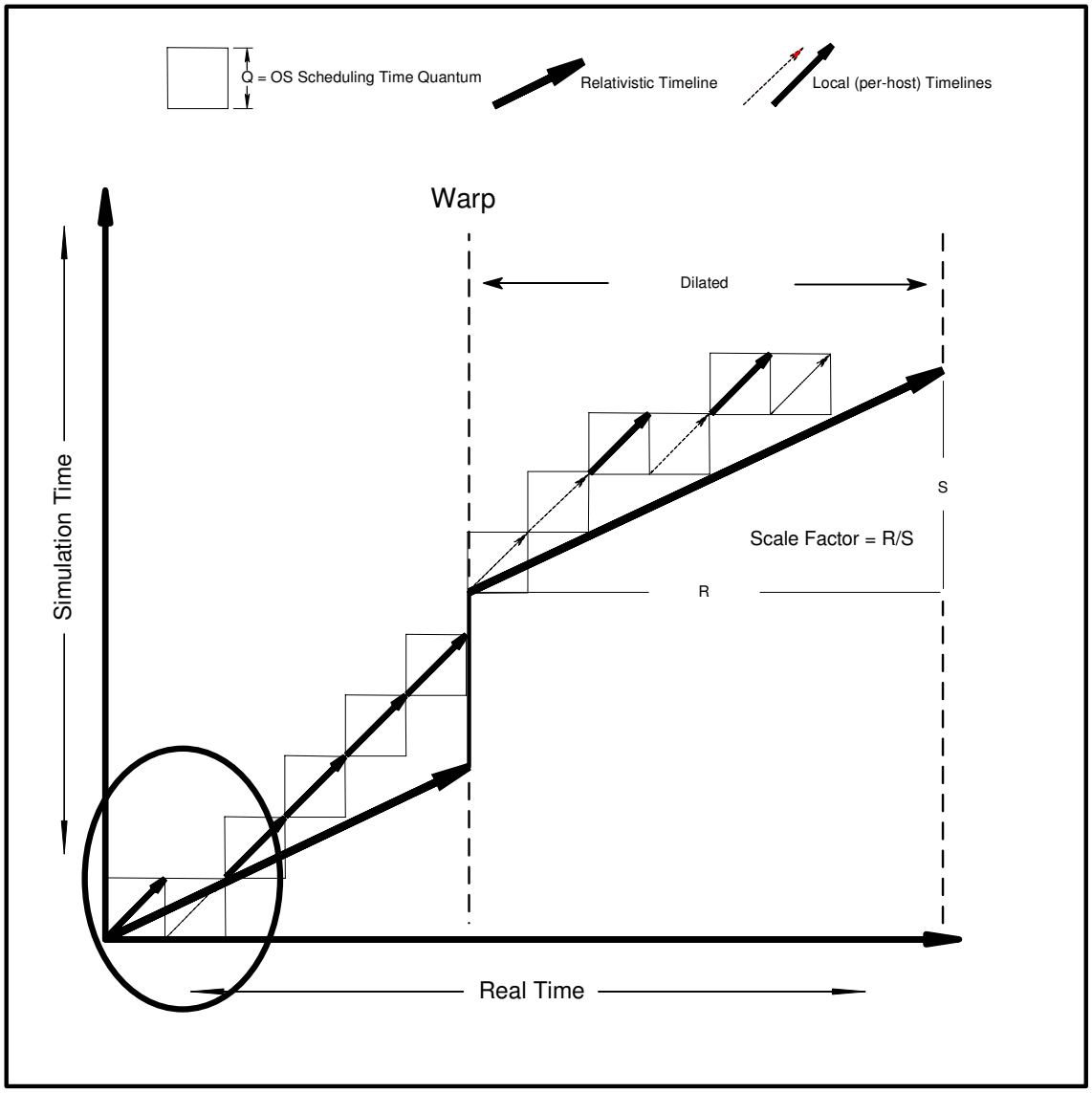

Figure 4.2: The relationship between relativistic time and virtual hosts' local times. The relativistic clock progress at a fraction of the rate of real-time. The fraction, the dilation factor, is set by the user at the beginning of a simulation. The local time of each virtual host (shown as smaller arrows) progresses at the same rate as real-time while executing on the simulators' processor and are stopped otherwise. When one host becomes blocked, another host can use its time quantums to progress ahead of the rest of the simulation, as shown by the small bold arrow progressing ahead after the host represented by a small thin arrow becomes blocked (within the circle shown at the bottom left of the image). A problem arises if virtual host times are started at the same instant in virtual-time as the global relativistic time. As shown in the circle at the bottom left, the thin arrow falls behind the global relativistic time. This violates one of our core assumptions, possibly causing events to be scheduled in the wrong order. To avoid this, the relativistic clock should be started slightly earlier (in virtual time) than local clocks at the beginning of a simulation and after warps.

thread. The time-lines shown demonstrate this behavior. If one host becomes blocked, its quanta are allocated to other hosts (This is shown by the short thin arrow only be present for one quantum at the bottom left of the image, and then the short bold host is allowed to execute repeatedly). This behavior allows the simulator to make use of all CPU cycles to advance a virtual hosts' clocks as long as any of its hosts are unblocked (negating time spent processing events and other overhead). Finally, when all virtual hosts become blocked, the simulation warps to the next system wide unblock event. This process continues until all virtual hosts terminate and the simulation ends.

As shown in the circled region of figure 4.2, if GRT is initialized to the same value as an unblocked host (or the same time as virtual hosts are initialized to at the beginning of the simulation), it is possible for virtual hosts clocks to fall behind global relativistic time even if scheduled in the ideal round-robin fashion. To prevent this *relativistic inversion*, each time a warp occurs (and on simulation initialization), relativistic time can be set at least Q time units less than the earliest unblocking event. Q should be tuned to the length of an OS timer quantum or greater if ideal scheduling is not guaranteed. This updated solution is shown in figure 4.3. An alternative, but equivalent approach is to include this time quantum size as a factor in the scheduling decisions of the global event processor. The implementation of DONE includes this second solution.

4.1.3 Global Time Management

During the execution of a simulation, global relativistic time progresses through alternating intervals of dilation and instantaneous warps. The dilation factor is set by the user at the beginning of the simulation and remains constant for all dilated periods. In contrast, warps are automatically detected and require no application or user intervention. It should be noted that this differs slightly from the discussion of global relativistic time-line in the section on relativistic time. DONE has no need for static or contracted periods.

In order for a warp to occur, all virtual nodes must be blocked, all inter-node event transport must be complete, and the next event (system-wide) must be greater than some constant threshold representing the cost of a warp. The following discussion describes how such warp opportunities are detected and distributed. First, the coordinator node's algorithm is discussed, and then the algorithm that the simulation nodes use is discussed.

One processor, the warp coordinator, is dedicated to identifying warp intervals by analyzing information collected from simulation nodes. The coordinator executes in a loop querying each simulation nodes to determine its *local warp bound*. The local warp bound represents the next event (in the virtual time-line) that the responding simulation node is aware of. Once all nodes in the simulation have replied with their local warp bound, the coordinator determines the minimum over all local warp bounds, the *global warp bound*. If and only if the global warp bound is greater than the warp cost threshold constant, the coordinator distributes knowledge of the warp opportunity to all simulation nodes. All nodes then participate in a barrier to ensure their clocks are ideally synchronized. Following this, they immediately set their relativistic clock to the agreed upon global warp bound and continue processing. The algorithm then begins another iteration.

To improve the scalability of the simulation, decrease the network load, and distribute computational cost of the warp protocol, simulating nodes do not reply to a warp query message (sent by the coordinator) until it is clear that it would be advantageous for that node to warp. To ensure

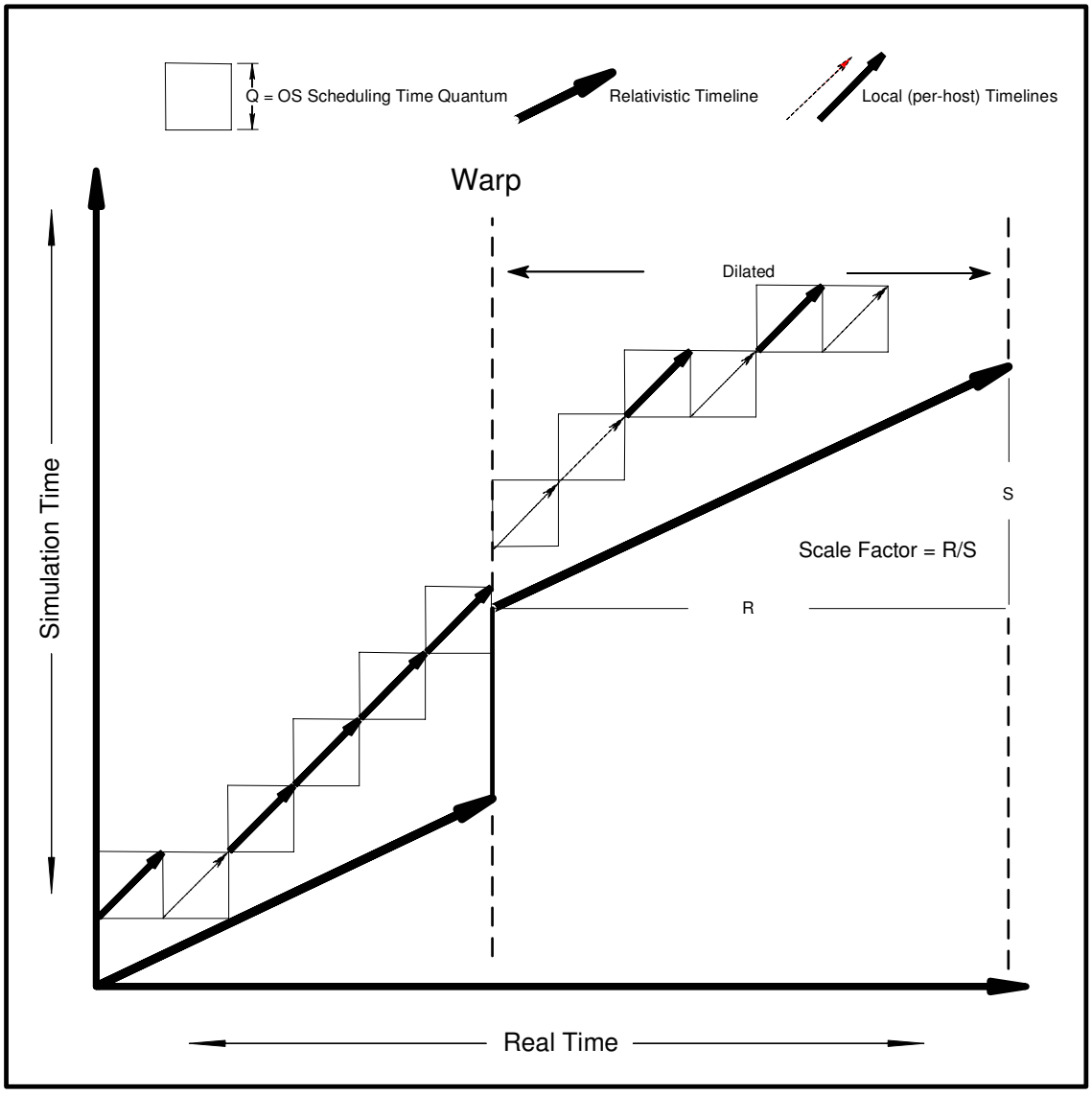

Figure 4.3: The relationship between virtual host time (local time) and global relativistic time is shown in this graph as it is implemented in DONE. Notice how local per host time-lines are initialized (at the beginning of the simulation and after warps) to have values greater than the value that which relativistic time takes. This ensures that relativistic time can be maintained as a system wide lower bound during the entire simulation.

this, first a node confirms that all of its virtual hosts are blocked. It then determines which event is the lowest over all locally known events (including outgoing, but unconfirmed events). If and only if (a) the lowest time-stamped event's time is greater than the current lookahead window plus the warp cost coefficient, and (b) that event's destination virtual host resides locally (the message does not need to be processed elsewhere), then the local node replies to the warp query. Outbound events are removed from local data structures when a confirmation of their delivery is received. Therefore, if the lowest time-stamped event is to be delivered to a remote virtual host, it has not yet been confirmed and may not have been received yet. Disallowing warps until a message has been confirmed prevents messages that are 'in-flight' from being warped past. If any of these criteria fail, then the node aborts the reply and sets a flag to ensure the pending reply is eventually answered. Pending reply flags are checked at regular intervals, when the final virtual host on a node becomes blocked, and when an event delivery confirmation message is received. Once all criteria are met, the simulation node returns a reply to the coordination node informing it of a warp opportunity.

4.2 The Dilation Factor

The user must specify the dilation factor at the start of the simulation. Increasing the dilation factor slows the progression of the global clock. Decreasing the dilation factor speeds the progression of the global clock. The advantage of using a higher dilation factor is that the simulator has more time (and hence more resources) to simulate each unit of global time. To counteract the effect of an over-dilated clock, warping skips intervals that the simulator would otherwise be idle.

Setting the dilation factor incorrectly can result in poor performance or an incorrect result. A dilation factor set excessively high will result in each simulation node repeatedly executing all events inside its safe lookahead window and becoming blocked before the lookahead window allows any additional events to be processed. If a dilation factor is set too low, the clock will progress too quickly for simulation processors to keep up with event processing. Events will enter the safe lookahead window and will be processed before it is truly safe to schedule those events. In this case, it is possible for an event with a lower time-stamp to be received after an event with a higher time-stamp was processed. This is a temporal inversion and may cause incorrect results. To allow efficient execution while preventing temporal inversions, the dilation factor should be set sufficiently low to allow efficient execution but not too low as to cause temporal inversion.

4.2.1 Determining the Ideal Dilation Factor

This section considers the theoretical minimum dilation factor required to ensure the simulation has sufficient resources to produce correct results. Later, in the evaluation section, consideration will be given to setting the dilation factor in the actual implementation of DONE.

Although the simulation may experience periods of under-subscription, it is desirable to use a configuration such that the simulator is provided with sufficient resources to handle the period of highest possible load without fault. All resources that are time shared between virtual hosts must be considered when setting the dilation factor. This includes the simulator's CPUs, and network, and memory bandwidth. Although DONE does not properly model the cost of disk I/O, the model could be extended similarly to what is shown here.

Since computation and network I/O occur in parallel, the derivation of their bounds on the scaling factor should be considered separately. Data copying (in memory) and computation do not occur simultaneously (a CPU is essentially blocked during the copying of large chunks of data). As a result of this, the effect of memory bandwidth should be considered in conjunction with CPU virtualization. The maximum over the dilation factor bound provided by CPU/memory centric factors and network-centric factors should be taken and used as the true system-wide bound. The overall minimum system-wide minimum bound on dilation is given below. Each of the contributing factors in this equation will now be considered independently.

$$MinimumDilation = max\{dilation_{network}, dilation_{cpu} + dilation_{memory}\} \tag{4.1}$$

To compute $dilation_{cpu}$, consider the case where all virtual hosts on all simulation processors are running. This case (which occurs at least once at the beginning of every simulation) is the worst case for processor usage per unit simulation time. We must ensure that even in this case the processor to which the most virtual hosts are assigned has sufficient computational cycles to keep all of its virtual host time-lines ahead of GRT. Since each virtual host's clock progresses in real-time when executing on the CPU, and each CPU is (ideally) divided equally between its virtual hosts, we must ensure that the GRT progresses no faster than $1/n$ (where n is the number of virtual hosts on the most highly loaded processor). This implies a minimum dilation factor of n.

$$dilation_{cpu} = n \tag{4.2}$$

On most systems, memory bandwidth and latency is significantly better than on the network. This motivates the strategy of placing all endpoints of as many virtual network links as possible on the same physical node. Although this is ideal, if the number of virtual network links on a single node is sufficiently large and all links are fully utilized, replicating data once for each receiver of every message could create a performance bottleneck. DONE includes real network protocol stacks which expect exclusive access to all application data buffers, all data sent by any virtual network client must be copied at least once for each receiver.

Consider the more general case where all network data must be copied at least C times while being transferred between virtual hosts. This is possible for an inefficient implementation of event delivery. Consider a system with an achievable memory bandwidth of $M_j$bps modeling a virtual network of speed $N_{i,j}$bps (on the ith virtual link of the jth physical processor). In this case, we must set the dilation factor to the maximum over all physical processors, as shown below.

$$dilation_{memory} = Max_j\{\Sigma_{i=links}\frac{C \cdot N_{i,j}}{M_j}\}. \tag{4.3}$$

The current implementation of DONE implements only point to point virtual network links. Using a broadcast network would significantly increase the overhead of sending messages over network links by adding the factor E_i for the number of endpoints on each link to the computation of $dilation_{memory}$.

$$dilation_{memory\_with\_broadcast} = Max_j\{\Sigma_{links}\frac{(E_i - 1) \cdot N_{i,j} \cdot C}{M_j}\}. \tag{4.4}$$

Independent of these factors is the ratio of physical network bandwidth to virtual network bandwidth. For network parameters, we must consider the worst case interface or switch; each interface and switch represents a potential bottleneck. For each interface in the simulator, divide the sum of the bandwidths of all virtual network links crossing that interface by the achievable throughput of that physical interface. This ratio represents the minimum dilation factor allowed by that interface. Next, consider each switch by dividing the aggregate bandwidth of all virtual links utilizing that switch by the throughput of its switching fabric. This ratio represents the minimum dilation factor allowed by that switch. The maximum ratio taken over all switches and interfaces in the system represents a system wide minimum dilation factor, denoted $dilation_{network}$ in equation 4.5.

$$dilation_{network} = Max\{\max_{interface} \frac{\Sigma_{vlinks\_across\_interface} BW}{interface\_bw}, \max_{switch} \frac{\Sigma_{vlinks\_across\_switch} BW}{switching\_fabric\_bw}\} \quad (4.5)$$

Apart from being a required input for a simulation run of DONE, the derivation of appropriate dilation factor is a good metric for the worst case cost of simulation (per unit time). It is an accurate description of how much real-time is required per unit virtual time if all network hosts are fully utilizing their virtualized resources. For example, we know that the dilation bound can be greatly affected by modeling a broadcast network with many virtual hosts. This indicates that the cost of simulating a network with many broadcast links will be significantly higher than a network with mostly point to point links.

4.3 Simulation Interface

The simulation engine provides several notable APIs which enforce a layered design. These APIs, discussed in this section, are of interest to users and developers of the simulation, as they allow protocol and application modules to interact with the core simulation engine at different levels. Additionally, understanding the division of state provided by the simulation and emulation APIs is critical to understanding how optimistic execution is safely permitted in DONE's conservative simulation model without violating consistency.

The interfaces and logical divisions between virtual host modules are shown in figure 4.4. The interface between virtual applications and their protocol stacks is the BSD sockets interface. Similarly, an interface exists between the protocol stack and the virtual network interface. This virtual network interface emulates the operating systems' network driver. A logical division exists between private application state and public protocol and network interface state. This division is significant because state transformations that occur purely inside a virtual application (private state) need not be ordered with respect to public state transformations. This allows application that execute transformations inside their private application state to executed ahead of the rest of the simulation. In effect, this provides for a optimistic execution in an otherwise conservative simulation.

4.3.1 Virtual Host Configuration Interface

After bootstrapping and weaving the simulation, internal data structures are constructed, and control is passed to application code to configure the IP stack. Configuration code is permitted to

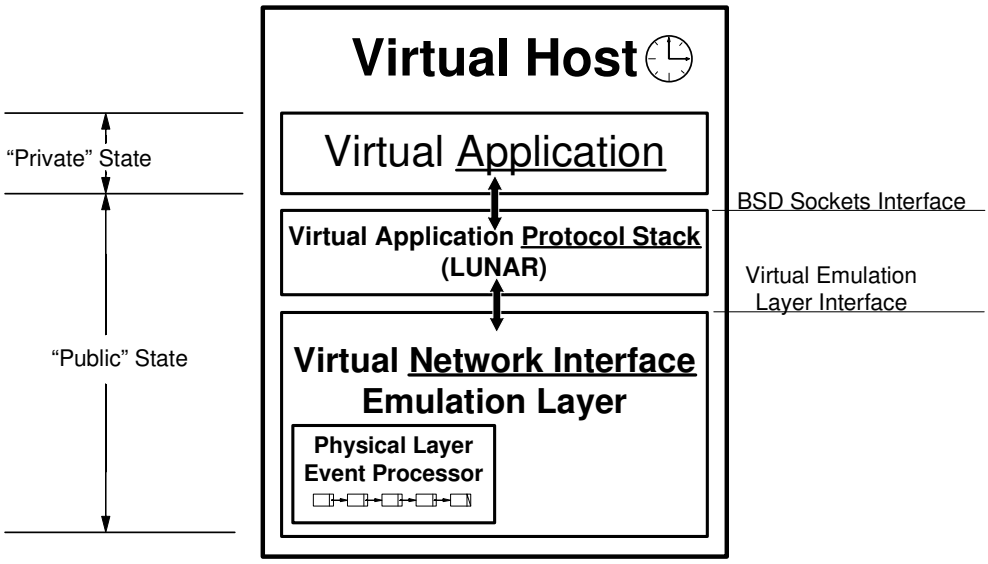

Figure 4.4: Detailed View of Per-Virtual-Host interfaces. The application (the virtual host's private state) interacts with the LUNAR stack through the BSD socket layer. The LUNAR stack interacts with the emulator through a virtual emulation layer interface, which mimics the relevant interfaces of the Linux operating system. The virtual hosts' public state consists of the protocol stack, the network interface emulation layer, and it's physical layer event processor. Accesses to the public state are carefully synchronized to ensure that they occur in the correct order. Alternatively, the application can execute in (modify and access) its private state without synchronization with the rest of the simulator. It should be noted that (although not shown here) the application can call into the network emulator layer to determine its environment or block for periods. A physical layer event processor delivers events to the LUNAR protocol stack through the network interface emulation layer.

run at this point (before the virtual time-line starts) to allow virtual hosts to self-configure without introducing timing constraints.

The user is provided an interface for configuring their IP stack at this stage via calls provided by LUNAR. LUNAR provides functions for configuring interfaces, ifconfig(int argc, char* arg[]), and for routing table manipulation, route(int argc, char* argv[]). An interface to net-filtering operations, iptables(int argc, char* argv[]) is also provided, but has not been extensively tested. Configuration code is placed in the function ip_config_node(int node_id) and is executed at the beginning of the simulation.

4.3.2 Virtual Application Interface

After the simulation has begun, control is passed to application_main(int argc, char* argv[]), the main application routine written by the simulation user or application developer. Applications are then free to run as they would on an actual BSD sockets system, using the sockets interface. In addition to the BSD sockets interface, applications also have access to several functions to return their current state and link characteristics. These functions, get_time(), phy_my_bps(int port), phy_my_latency(int port), phy_my_pcost(int port), and phy_my_neighbor(int port) evaluate to useful values describing the virtual hosts' links and state in terms of bits per second and nanosecond resolution clock values.

To provide execution monitoring capability, the simulation provides a macro, APP_STAT(char*), allowing user code to define a character string buffer into which it can place a descriptive string. When a simulation process receives a signal, it prints out all of its virtual hosts' application status strings. This can be used, for example, to trace a large number of applications and monitor for the occurrence of bugs during execution. Monitoring for bugs using the standard IO libraries is difficult in all except the smallest simulations due to the way that MPI reorders output. To alleviate this problem, standard out redirection has also been built into the emulation layer and is discussed in the appendix. Both the APP_STAT() and standard out redirection mechanisms provides a way for virtual hosts to produce output for examination during and after the simulation.

4.3.3 Virtual Application Interface & blocking

Like any user-space application running on a UNIX system, virtual hosts have access to any function provided by the system libraries. There has been no attempt to ensure that every library call works as expected. Many library calls will produce correct results, but users must carefully consider the implications of external calls. Simple state manipulation calls that modify memory or perform some computation should work as expected. These include memcpy() and math library calls, for example. However, I/O routines outside the BSD sockets library may produce unexpected results. In particular, virtual host clocks may be adversely affected by blocking calls to the standard libraries.

Disk I/O routines are particularly troublesome, not because they fail to read from the disk or return incorrect values, but because virtual host time-line virtualization does not consider blocking caused by standard libraries. For I/O calls that block virtual host threads while waiting for data, serious problems may arise. When a virtual host issues a blocking call to read data from the disk, the quantity of real-time that it spends waiting for the data from disk is non-deterministic and is related the number of pending requests to read from the disk. Additionally, no mechanism has

been employed to ensure that a virtual host's clock is advanced to account for the time that it spends reading from the disk. The result is that virtual host's local clocks can fall behind the global relativistic clock, violating a primary assumption of the relativistic time model. This, in turn, can cause events to be scheduled in reverse time-stamp order (a temporal inversion and an error).

More difficult still is the non-determinism of disk I/O, and interferences that occur when multiple reads are issued by different virtual hosts sharing the same disk. As more virtual hosts issue more reads to the disk, for example, each request takes longer to complete. Therefore, we cannot simply measure the amount of time a disk read takes to complete and advance the virtual host's clocks in accordance with that quantity. This would result in interferences between virtual hosts and incorrect results. Instead, a more complex model is required.

A model that performs correctly for disk I/O could be constructed by providing an estimation of time required to retrieve data. Then, interfaces could be provided to user applications returning the pre-fetched data and advancing the virtual hosts' clocks. Due to the nature of disk I/O, data should be pre-fetched to ensure that it can be retrieved by the deadline specified on the relativistic time-line. Disk I/O calls should be intercepted and made to modify virtual host time-lines in a manner consistent with the model developed.

4.3.4 Emulation Layer / Simulation Engine Binding

In addition to the interface intended for applications to bind themselves to the protocol stack, the simulation core provides a virtual emulation layer interface, shown in table 4.3.4. This interface binds the protocol stack to a virtual network interface, allows applications to learn about their environment, and provides a means for safe non-blocking output. It provides a way for the protocol stack to pass data to the virtual network hardware, and a means for the application protocol stack to register callbacks for message receive and timer events.

Table 4.1: The emulation layer interface is summarized here. These calls support LUNAR by providing a virtualized operating system and network interface. Calls are divided by category and are provided for network message insertion and delivery, virtual host scheduling, and environment inspection.

Name	Description
write_msg()	Allows protocol stacks to put messages 'on-the-wire'
phy_init_read()	Allows protocol stacks to initialize their read callback
one_init_timer()	Allows protocol stacks to initialize their timer callback
one_start_timer()	Starts the OS personality layer 10ms timer
one_block_until()	Blocks a virtual host's string for until a specified time
one_block_for()	Blocks a virtual host's string for a specified number of nanoseconds (virtual time)
one_block_costring()	Low-level interface to costring blocking accounting. Marks one costring blocked
one_unblock_costring()	Low-level interface to costring blocking accounting. Marks one costring unblocked
enq_event()	Low-level interface to insert events for delivery
phy_my_bps()	Macro that evaluates to the bandwidth of link N for any virtual host
phy_my_latency()	Macro that evaluates to the latency of link N for any virtual host
phy_my_pcost()	Macro that evaluates to the processing cost of link N for any virtual host
phy_my_neighbor()	Macro that evaluates to the switch id of neighbor N for any virtual host
APP_STAT()	Macro allowing applications to pass debugging and safe output to simulator
printf(...)	Standard output redirection interface (optional)
fprintf(...)	Standard output/error redirection interface (optional)

Chapter 5

Evaluation

This section contains an evaluation of DONE. It shows that DONE scales well and provides a reasonable degree of accuracy. It is organized as follows: First a discussion of the experimental setup used, then a discussion of a set of experiments intended to show that the simulator provides correct results, and finally a discussion of simulation scalability.

5.0.5 Limitations

The tests shown in this chapter are not exhaustive and are not intended to present all possible input scenarios for the simulator. Likewise, the simulation is not optimized to perform well in all scenarios. Instead, the simulation is directed towards very large simulations consisting of many virtual hosts with a reasonable quantity of virtual network latency.

Scenarios where DONE will provide the best speedup are situations in which all application programs are computationally expensive, and computational tasks are spread evenly across all physical processors during similar intervals of virtual-time. Two computational tasks hosted by two distinct physical processors that take place at the same time on the virtual time-line can be performed concurrently. However, two computational tasks that take place at two distinct time intervals on the relativistic time-line (separated by at least one lookahead window), are not identified as independent. Since relativistic time does not provide a means to prove that these two computations can be performed concurrently, they will be performed serially and poor speedup will result.

The application used in most test scenarios is network bound. The network bound case represents the worst case for parallel speedup, but the best case for accuracy. All applications start at the same time, and perform similar deterministic tasks. As a result of this, the vast majority of computation (and event processing) takes place at the same virtual time on all physical processors. In this respect, the scenarios tested are DONE's best case scenario for parallel speedup. A more realistic scenario would consist of a mix of network and CPU bound tasks applications taking place at randomly distributed times. Even with these limitations, the tests shown in this section demonstrate a number of useful characteristics.

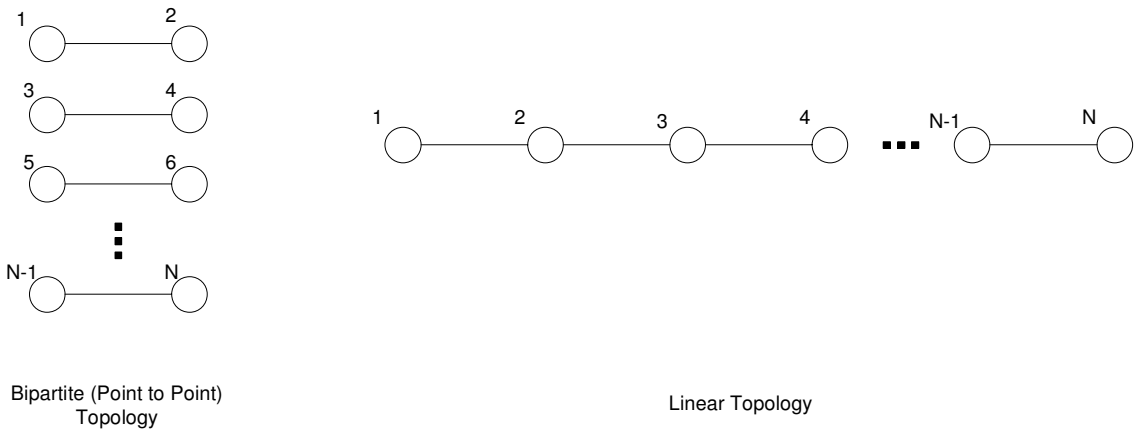

Figure 5.1: Linear and bipartite topologies (shown above) have IP auto-configuration built into DONE. The bipartite topology (shown on the left) consists of an even number of virtual hosts connected by point to point links, each virtual host is connected to exactly one other node. The linear topology consists of a daisy chain of virtual hosts connected in a linear fashion.

5.1 Experimental Setup

All simulations were performed on a Redhat Fedora Core 3 based system in a disk-less configuration. Two separate hardware configurations were used. In both configurations, compute nodes were PXE booted off a head node that also served their root file systems over NFS. Smaller tests (consisting of up to 16 processors) were run on eight dual-2Ghz 246 opterons. All nodes in this configuration had 2GB main memory and were configured without swap space. Larger scale tests were done a 64 processor opteron 240 cluster clocked at 1.4Ghz. Each node (32 in all) had two processors and 1GB of main memory also configured without swap space.

Two simple TCP applications, a server and a client, were used for all tests. The server side opens a listening socket, accepts a single connection, and issues one read call. Once the first read returns, it saves the time of the call's return and continues reading data in one kilobyte chunks until the expected quantity of data has been received. It then prints out statistics on the duration, length, and rate of the transfer and sends a one kilobyte response to the client confirming the end of the transfer. Finally, it shuts down its half of the socket with a call to shutdown(socket_fd, SHUT_WR) and returns. The client application connects to the server and sends a pre-determined amount of data across the socket. It then reads the server's confirmation message, shuts down its half of the socket, and returns. In compute bound tests, additional code was added to perform a configurable quantity of computation between each socket call in the main loop.

To ensure that all transfers finish and no data is lost in the internal buffers of any instance of the LUNAR stack, all protocol stacks continue processing events (timer tick events, receive message events, and unblock events) even after their application has shutdown. Once every application on all virtual hosts has shutdown, all protocol stacks stop processing events and the simulation is terminated.

Future plans include integrating network interface configuration and routing tables into a config-

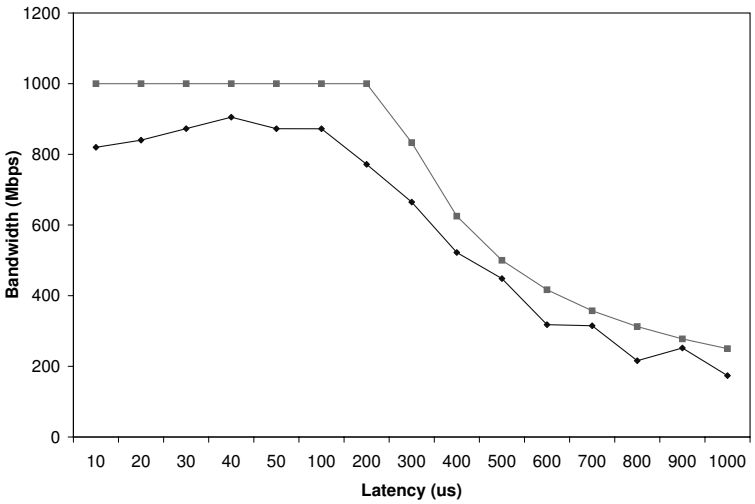

Figure 5.2: Measured bandwidth for different simulated latencies. The top curve shows the theoretical maximum, the bottom curve shows the measured throughput. The simulation verifies the expected behavior of a sliding window protocol.

uration file; however, to test the simulation, automatic configuration has been provided for the two topology types shown in figure 5.1. A set of point to point links is the simpler of the two. Each node participates in just one link. In this case, the server is configured as having IP address 192.168.0.1 and the client is configured as having IP address 192.168.0.2.

The linear topology, a slightly more complicated case, consists of up to 63 nodes. Each node participates in two point to point links. This results in a daisy chain of nodes, strung together by a line of links. Each node is assigned two IP addresses, one for each virtual interface. IP addresses are assigned starting at 192.168.0.1 for the first virtual host, 192.168.0.2 and 192.168.0.5 for the second virtual host's interfaces, 192.168.0.6 and 192.168.0.9 for the third virtual host's interfaces, and so on. To allow routing of packets across an entire string of hosts, routing entries are set up on each virtual host for all other virtual hosts. Although one would not expect this topology to be used in any realistic network, it provides a useful means to evaluate the emulator's packet forwarding capabilities.

5.2 Correctness Tests

5.2.1 TCP Window Verification

TCP is unable to fully utilize the throughput of high-bandwidth high-latency links [12, 4]. To prevent a receiver from being overwhelmed with data, and for historical reasons, TCP's default flow control mechanisms allows only 64 kilobytes of data to be unacknowledged by a receiver. In order to be acknowledged, data must first be received, and then an acknowledgment must be returned across the link. This process occurs over an interval twice the length of the latency of the link connecting the two hosts, or one Round Trip Time (RTT). Let RTT equal this Round Trip Time in seconds,

Table 5.1: Observed TCP Throughput for transmitting 5MB of data at different simulated bandwidths.

Link B/W	Throughput	Run Time	Simulated Time	Speedup	No of Warps
56kbps	54.842kbps	16.01s	764.637s	47.637	83,731
100kbps	95.638kbps	9.98s	438.474s	43.935	51,101
500kbps	478.182kbps	3.32s	87.696s	26.412	15,083
1Mbps	0.956Mbps	2.53s	43.849s	17.331	11,734
10Mbps	9.427Mbps	1.74s	4.448s	2.556	6,727
100Mbps	78.182Mbps	1.66s	0.536s	0.323	6,332

W be the maximum window size in bits, and link_bps be the link's achievable bandwidth in bits per second. The link throughput is bounded by $throughput <= min\{\frac{W}{RTT}, link\_bps\}$ (where link_bps is the link bandwidth).

To show that DONE accurately models network characteristics of simulated networks and therefore exhibits this behavior, different latencies were modeled on a single point to point link at 1Gbps. The application's achieved throughput was measured and the results are shown in figure 5.2. The higher line shows the theoretical throughput that a 64 kilobyte window should allow (negating protocol and implementation overhead). A second line shows the achieved throughput over the link. As expected, for low latency links, the link bandwidth is the bottleneck to achieved performance, and roughly 80% efficiency is achieved. When $RTT > \frac{W}{link\_bps}$, latency becomes a bottleneck to throughput. For a 1Gbps link with a 64KB TCP congestion window, this occurs at $\frac{64KB}{1Gbps} = 524us$. Dividing the RTT by two to get latency, we arrive at the conclusion that performance should begin to degrade around $262us$. As shown in the figure, performance degrades as expected when latency exceeds this theoretic number.

The reduction in link efficiency as latency increases is not explicitly coded into the logic of the simulator. Instead it is the result of the interaction of the simulator and multiple instances of the protocol stack. This behavior demonstrates that the simulator properly models both simulation latency and bandwidth parameters on a 1Gbps link and that emulated components and simulated components interact as expected.

5.2.2 Bandwidth Variation Test

To verify that DONE correctly models a wide variety of virtual link speeds, a series of tests was performed with a variety of link bandwidths. For each of the tests in this section, a single point to point link is simulated on a single physical processor with two virtual hosts transferring five megabytes of data (a second processor on the same memory bus was used for warp coordination). Table 5.1 shows performance in this non-distributed case. It includes achieved throughput, the amount of (real) time spent processing events, the speedup versus a real-time emulation, and the number of global warp events during the course of the simulation.

Timer tick events add a constant amount of overhead per unit virtual time. Slower virtual link take more virtual time than faster virtual link to send the same amount of data. The conclusion is

Table 5.2: Observed average and standard deviation throughput for best case and worst case node assignment scenarios.

Scenario	Average Bandwidth	Standard Deviation
Least Distributed Case	0.956271Mbps	0.000018 Mbps
Most Distributed Case	0.956359Mbps	0.000338 Mbps

that simulating the same size data transfer over a slower link is more expensive due to an increased number of timer events. Event density per unit virtual time is lower for slower links, so there is a greater speedup versus the real-time emulation case in this case. These conclusions are supported by the experiments run in this section.

To show that the simulation provides accurate results when virtual links cross physical links, two simulations were run over 10 simulation processors (with one coordinator processor). One megabyte was sent over 50 1Mbps point to point links with 50us latency. The first run distributed every virtual link across a physical link (most distributed case). The second placed both ends of every virtual link on the same physical processor (least distributed case). Average and standard deviations of the achieved throughput for the 50 connections are shown for both runs in Table 5.2. Although there was a slight increase in deviations and throughput on the most distributed case, the difference is minimal.

5.2.3 IP Forwarding

To verify that LUNAR and DONE interact to correctly implement IP forwarding, a configuration was setup to model a linear topology. This linear topology, shown in figure 5.1, exercises IP forwarding on each intermediate host in the topology when a connection is established from one end of the topology to the other. In these tests, the end-most hosts run the same application server and clients as in the point to point case. Each intermediate node runs no application at all. The client on one end of the linear chain of nodes connects to the server on the other end, forwarding packets across every node in the chain. Link speeds of 1Mbps with 50us latency were used for all virtual links.

This experiment was repeated twice, once placing each virtual host on its own physical processor and a second time placing all virtual hosts on the same physical processor. In the non-distributed case the achieved throughput was 0.906711 Mbps and in the distributed case the achieved throughput was 0.903209 Mbps. The total difference between these two runs was 0.003502 Mbps or less than 0.4%. This experiment demonstrates that IP forwarding functions as expected over virtual links in both the distributed and non-distributed cases.

5.3 Scalability Tests

The tests in this section are intended to show that DONE scales well on moderate to large input data sets. Several tests are shown that demonstrate different aspects of scalability and characteristics of the simulation engine.

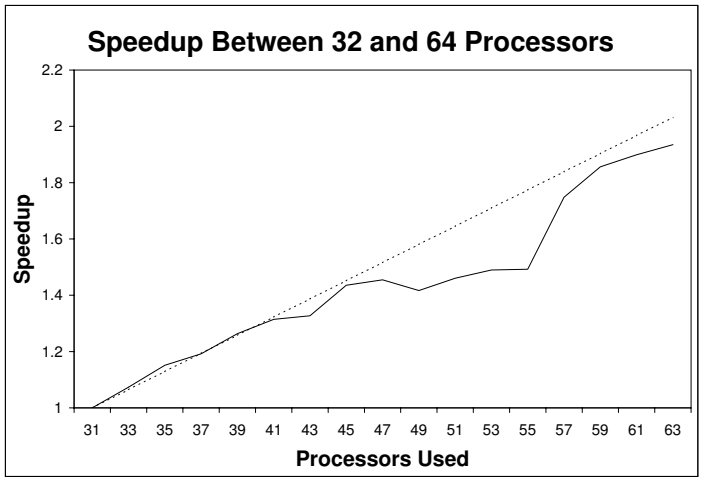

Figure 5.3: Parallel speedup for physical processor counts between 31 and 64 processors on the Anantham cluster. 70,000 virtual hosts are distributed over between 31 and 63 simulation processors in a peer to peer fashion.

5.3.1 Large Scale Test

To show that DONE provides good speedup on a large problem and moderately large physical processor counts, a scale test was performed on the Anantham cluster. For this test 70,000 virtual hosts were simulated using between 32 and 64 processors. Like many distributed algorithms, performance on larger processor counts is better with larger input data sets. In effect, this reduces the ratio of communication (and coordination) to computation. When well-tuned, 70,000 virtual hosts is the largest simulation that fit into the 32GB of memory across these 32 processors; therefore this is the size of the data set used for this set of tests. Smaller topologies are not likely to produce speedup as good as this test. However, with more memory (and hence, a larger problem size), the same processors are likely to provide better speedup.

Figure 5.3 shows parallel speedup for between 33 and 63 physical processors running 35,000 point to point links connecting 70,000 virtual hosts as compared to the same problem running on 31 physical processors. All simulations consist of one additional coordination processor (not included in the processor count or speedup calculations), so actual processor counts are between 32 and 64. Achieved speedup for these tests is nearly linear.

5.3.2 Moderate Scale Test

To show the performance of the simulation on more modest processor counts, a series of tests was performed on a 16 processor opteron system, ClusterONE. For these tests, one megabyte of data was transferred between each server/client pair with a total of 4,000 total virtual hosts. Speedup for this test is shown in table 5.4 and is super-linear for processor counts between 3 and 13. It is believed that the super-linear speedup is due both to the greater cache capacity of a system with larger processor counts and to less overhead per event processed when the ratio of virtual hosts to simulator processors significantly large. The lesser speedup on the 15 processor configuration can

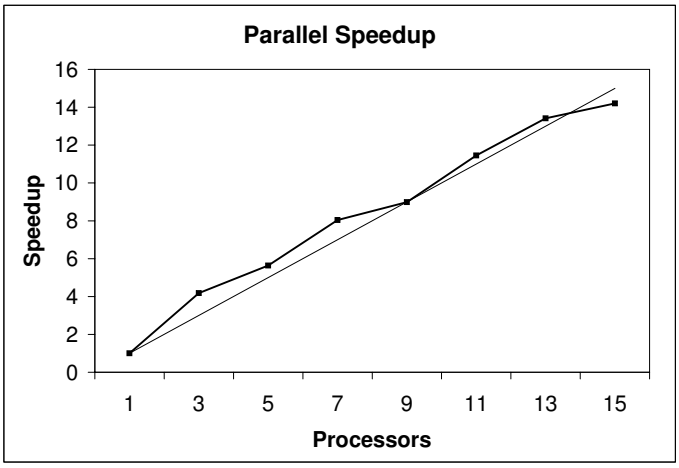

Figure 5.4: Parallel speedup for different physical processor counts, using an optimal load balancing that minimizes communication. The bottom curve shows linear speedup. DONE achieves a super-linear speedup on processor counts up to 13 processors, as shown in the upper curve. For this test, 4000 virtual hosts were used in a point to point configuration. Each host transferred one megabyte to it's immediate neighbor.

be accounted to the addition of the NFS server to the processor pool.

5.4 Performance and Accuracy : Setting the Scale Factor

To show the impact of the relativistic scaling factors, a series of experiments was run with a wide range of scale factors. Two different applications were run in all cases. One application is the same TCP application that was used for the other experiments, and the second was a cpu bound variant. The cpu bound variation does approximately one-tenth of a second of computation between each send and receive socket call. This application was used for two reasons: first to show that cpu bound applications will perform as expected on DONE, and second to show how using a cpu intensive application will affect the performance and accuracy of the simulation.

It should be noted that the range of scale factors chosen is not uniform over the entire range. All tests were run with 100 virtual hosts over 12 physical processors (11 of which were simulation nodes). All virtual links had a bandwidth of 1Mbps, and nodes were placed in a way that minimizes the number of virtual links crossing across physical links. There are between 9 and 10 virtual hosts per physical processor, leading us to believe that a good choice of a scaling factors is around 10 (negating overhead). Increasing the scale factor to account for overhead would lead us to believe that a scale factor of somewhere between 20 and 100 would be ideal. To provide a detail view of this region of interest and gain insight on overly conservative scale factors, a linear scale was chosen over the range 20 to 200. Additional data points were added at 5 and 10 to provide for the cases where the simulation is knowingly oversubscribed. To show how the simulation performs over a wide range of values, a log_{10} scale was chosen for the range 200 to 200,000.

Figures 5.5 and 5.6 show run times for the two cases over a variety of scaling factors. As expected,

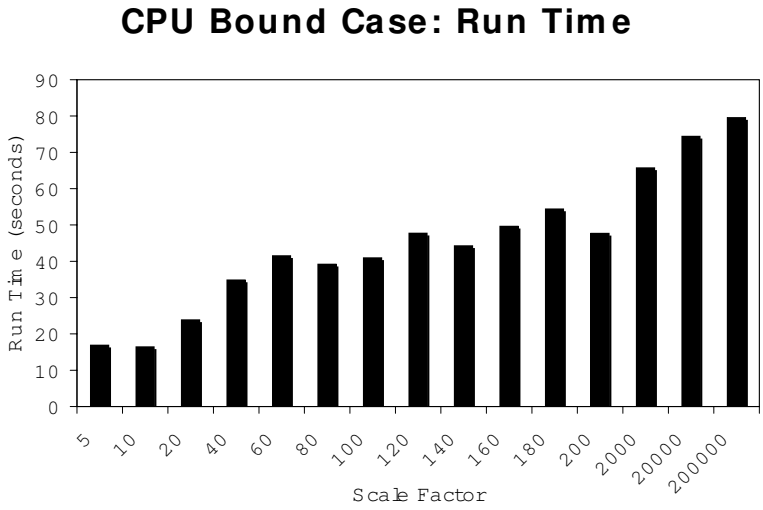

Figure 5.5: Run time for the CPU bound problem over various relativistic scaling factors. For the CPU bound problem, increasing the scale factor results in additional global warp events, more overhead, and a slower simulation.

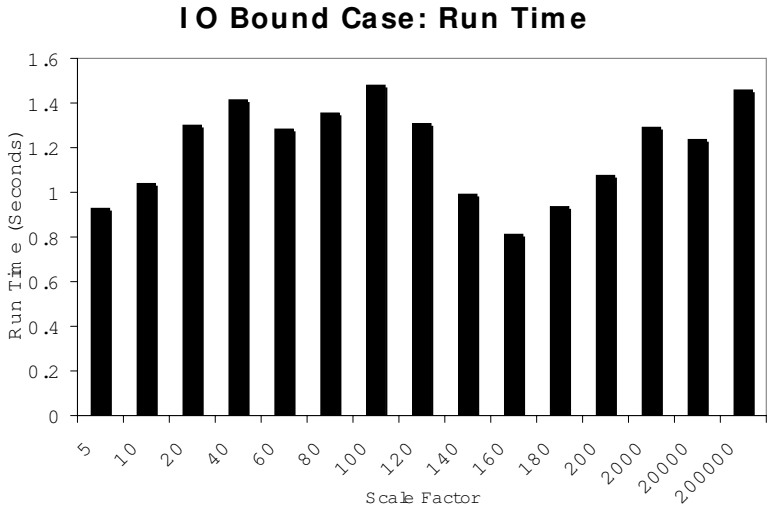

Figure 5.6: Run time for the IO bound problem over various relativistic scaling factors. Run time seems independent of scaling factor in the io bound case. This is due to the tendency for the I/O bound problem to have alternating periods of extreme under-subscription and over-subscription. This results in frequent warps, even when the dilation factor is set quite low.

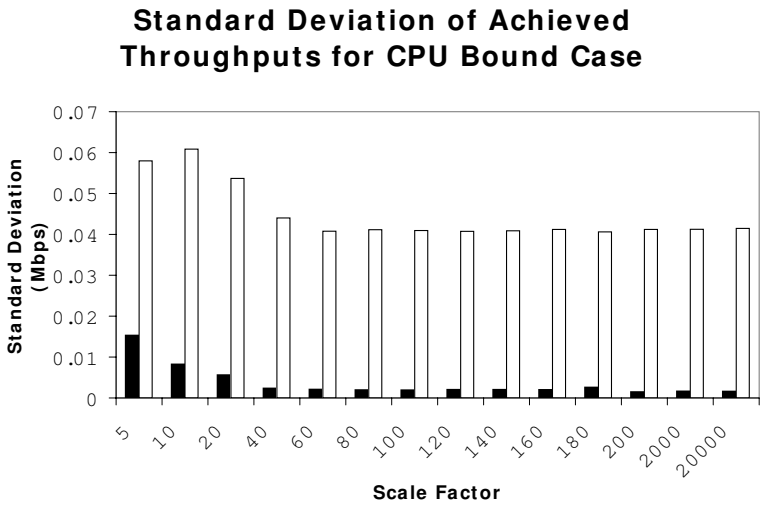

Figure 5.7: The standard deviations (filled) and averages (unfilled) of the 50 throughput measurements provided by a single simulation run are shown. Each simulation run consists of 100 CPU bound hosts in a point to point configuration. Although results tend to be very consistent for high scaling factors, they vary when the scaling factor is set too low. This is a result of insufficient resources per unit virtual time to accurately model the configuration provided.

the cpu bound case shows a tendency to run faster with lower scale factors. On the other hand, run times for the I/O bound case seem unrelated to the scale factor setting. This is because the system's cpu is under-subscribed for periods between message receipts on I/O bound problems. As a result, the simulation warps between each message reception regardless of the scaling factor used.

Figures 5.7 and 5.8 show the standard deviation of the throughput as reported by the applications run on the simulation. This is believed to be a good metric for the accuracy of the simulation, as all application instances represent identical cases and *should* produce the same throughput. Any deviation in values returned can be attributed to error. As expected, with scale factors set very low (thus oversubscribing the simulation) the standard deviation of throughput increases. This indicates a greater tendency for error. To reinforce this point, the average throughput reported by the applications are shown in figures 5.9 and 5.10. Notice that for scale factors set inappropriately low, the values deviate from those shown for high scale factors. For scale factors that are set high, the values stabilize. This indicates that although for scale factors set too low the simulation is unable to provide accurate results, it becomes more consistent when the scale factor is set appropriately.

An important distinction exists between results for the cpu bound and io bound cases. Although all metrics of error discussed here tend to decrease as the scale factor is increased, the cpu bound tasks do not achieve the same degree of accuracy as the I/O bound tasks. There are several reasons for this. First, variations due to micro-architectural influences (such as the effects of cache) are not accounted for in the model and are a source for some of the errors seen in both cases. Cache issues are magnified by exercising the CPU more, which causes a virtual host to spend more time executing. Additionally, every processor is responsible for both event handling and virtual host thread execution. As the virtual host threads use more of the cpu, the event scheduler receives less

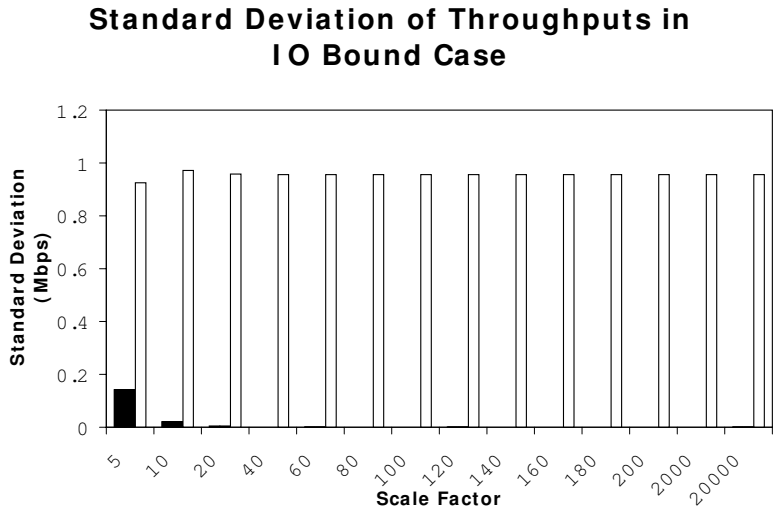

Figure 5.8: The standard deviations (filled) and averages (unfilled) of the 50 throughput measurements provided by a single simulation run are shown. Each simulation run consists of 100 IO bound hosts in a point to point configuration. Although results tend to be consistent for high scaling factors, they vary when the scaling factor is set too low. This is a result of insufficient resources per unit virtual time to accurately model the configuration provided.

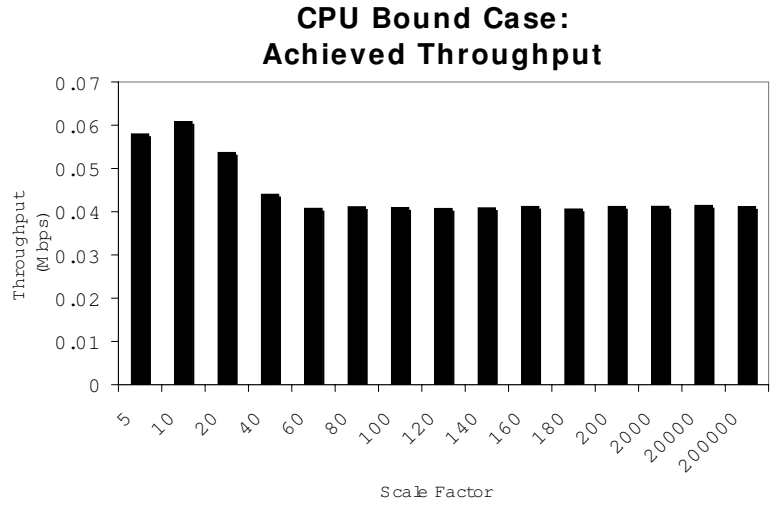

Figure 5.9: A single simulation run of 100 CPU bound virtual hosts in the point to point configuration results in 50 measurements of throughput being output. This experiment was run several times, on various scaling factors, and the average throughput is shown in this graph. Although results are consistent when scale factors are set sufficiently high, they vary when the scale factor is set too low. This indicates that low scaling factors result in inaccurate results.

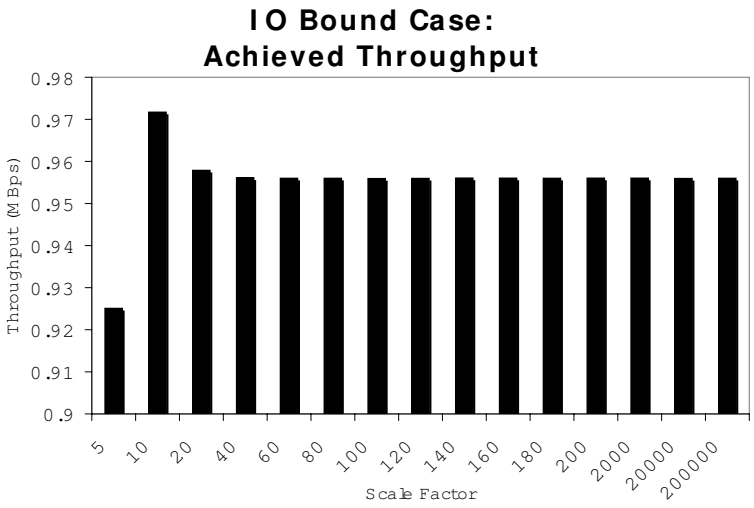

Figure 5.10: A single simulation run of 100 IO bound virtual hosts in the point to point configuration results in 50 measurements of throughput being output. This experiment was run several times, on various scaling factors, and the average throughput is shown in this graph. Although results are consistent when scale factors are set sufficiently high, they vary when the scale factor is set too low. This indicates that low scaling factors produces inaccurate results.

of a share of the cpu and accuracy suffers. With this said, the standard deviations of throughput in the cpu bound case are less than 5% of the achieved throughput (for high scale factors) and are less than 0.03% in the I/O bound case (see figures 5.8 and 5.7).

Chapter 6

Conclusion and Future Work

This work extends several previous efforts in network simulation. It integrates the development of a user level network protocol stack, LUNAR [15], with an implementation of the relativistic time [24] on a distributed memory architecture. It makes use of the Weaves Run Time Framework [23] and previous work on shared memory network simulation. This integration manifests itself as a functional network application and protocol simulation engine. The simulation engine was tested to ensure both that it provides correct results and scalable performance. It has been shown to scale to tens of thousands of virtual hosts on a modest 16 processor cluster. Additionally, it has provided super-linear speedup on a small cluster and nearly linear speedup on a large cluster.

In addition to work directly in network emulation, several contributions have been made to relativistic time. It has been implemented and shown to provide good performance in a distributed memory environment. Additionally, lookahead has been incorporated into the model by assigning specific meaning to the global time-line. Possible consistency problems have been identified when application state and protocol state is not separated. These problems have been addressed by dividing virtual host state into portions that are globally accessible (via message delivery) and portions that are private. Accesses to globally accessible state are serialized to ensure that time and state remain consistent. This solution still relies on the Linux scheduler to provide nearly round-robin scheduling, however, which is not ideal.

Although initial results are promising, more work remains to be done. A new configuration file that includes the capability to integrate protocol stack configuration with physical topology description and virtual host configuration would yield a much more usable environment. Additionally, the reliance on Linux's scheduler to provide near round-robin scheduling is not ideal and requires virtual host state to be explicitly separated into internal private state and external public state. Modifications to the operating system scheduler would provide consistency guarantees by making scheduling decisions that are relativistic-time-aware. This would negate the need for protocol stack state separation, and may introduce performance benefits. The result would be a simulation more tailored towards protocol stack development and would not require users to understand the inner workings of relativistic time to modify the protocol stack.

REFERENCES

[1] Jong Suk Ahn and Peter B. Danzig. Packet network simulation speedup and accuracy versus timing granularity. *IEEE/ACM Trans. Netw.*, 4(5):743–757, 1996.

[2] Rassul Ayani and Hassan Rajaei. Parallel simulation using conservative time windows. In *Proceedings of the 1992 Winter Simulation Conference*, pages 709–717, 1992.

[3] Roberto Beraldi and Libero Nigro. Exploiting temporal uncertainty in time warp simulations. In *DS-RT*, pages 39–46, 2000.

[4] Craig Bergstrom, Srinidhi Varadarajan, and Godmar Back. The distributed open network emulator: Using relativistic time for distributed scalable simulation. *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, 0:19–28, 2006.

[5] Lawrence S. Brakmo and Larry L. Peterson. Experiences with network simulation. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 80–90, New York, NY, USA, 1996. ACM Press.

[6] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977.

[7] K. Mani Chandy and Jayadev Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–205, April 1981.

[8] Xinjie Chang. Network simulations with opnet. In *WSC '99: Proceedings of the 31st conference on Winter simulation*, pages 307–314, New York, NY, USA, 1999. ACM Press.

[9] Richard M. Fujimoto. Exploiting temporal uncertainty in parallel and distributed simulations. In *PADS '99: Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 46–53, Washington, DC, USA, 1999. IEEE Computer Society.

[10] Richard M. Fujimoto. Parallel simulation: parallel and distributed simulation systems. In *WSC '01: Proceedings of the 33nd conference on Winter simulation*, pages 147–157, Washington, DC, USA, 2001. IEEE Computer Society.

[11] N. C. Hutchinson and L. L. Peterson. The x-kernel: An ar5chitecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[12] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992.

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

[14] Cameron Kiddle, rob Simmonds, Carey Williamson, and Brian Unger. Hybrid packet/fluid flow network simulation. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, 2003.

[15] Christopher C Knestrick. Lunar: A user-level stack library for network emulation. Master's thesis, Virginia Polytechnic Institute and State University, 2004.

[16] Jason Liu and David M. Nicol. Lookahead revisited in wireless network simulations. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, 2002.

[17] Jayadev Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[18] Joy Muhkerjee and Srinidhi Varadarajan. Weaves a framework for reconfigurable programming. *International Journal of Parallel Programming*, 33:279–305, 200.

[19] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[20] George F. Riley, Talal M. Jaafar, Richard M. Fujimoto, and Mostafa H. Ammar. Space-parallel network simulation using ghosts. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, 2004.

[21] Wen-King Su and Charles L Seitz. Variants of the chandy-mistra-bryant distributed discrete-event simulation algorithm. In *Proceedings of the 1989 Distributed Simulation Conference*, December 1989.

[22] B. Szymanski, Y. Liu, and R. Gupta. Parallel network simulation under distributed genesis. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, 2003.

[23] Srinidhi Varadarajan. The Weaves runtime framework. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop*, pages 197–, October 2004.

[24] Srinidhi Varadarajan and Richard Nance. Relativistic time: The evolution of temporal models for multi-domain simulation. In *Proceedings of the 1999 Winter Conference on Simulation*, December 1999.

[25] H. Wu, R. Fujimoto, and G. Riley. Experiences parallelizing a commercial network simulator. In *Proceedings of the 2001 Winter Conference on Simulation*, 2001.

[26] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. Glomosim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.