# HydraNetSim

A Parallel Discrete Event Simulator

MUHAMMAD FAHD AZEEMI

KTH Information and
Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

# HydraNetSim: a Parallel Discrete Event Simulator

Muhammad Fahd Azeemi
mfahad@kth.se

Master's Thesis
7/4/2012

Examiner : Professor G. Q. Maguire Jr.
Supervisors: Professor G. Q. Maguire Jr.
and
Christof Leng

School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

# Abstract

Discrete event simulation is the most suitable type of simulation for analyzing a complex system where changes happen at discrete time instants. Discrete event simulation is a major experimental methodology in several scientific and engineering domains. Unfortunately, a conventional discrete event simulator cannot meet with increasing demands of computational or the structural complexities of modern systems such as peer-to-peer (P2P) systems; therefore parallel discrete event simulation has been a focus of researchers for several decades.

Unfortunately, no simulator is regarded as a standard which can satisfy the demands of all kinds of applications. Thus while given a simulator yields good performance for a specific kind of applications, it may failed to be efficient for other kinds of applications. Furthermore, although technological advancements have been made in the multi-core computing hardware, none of the mainstream P2P discrete event simulators is designed to support parallel simulation that exploits multi-core architectures. The proposed HydraNetSim parallel discrete event simulator (PDES) is a step toward addressing these issues.

Developing a simulator which can support very large numbers of nodes to realize a massive P2P system, and can also execute in parallel is a non-trivial task. The literature review in this thesis gives a broad overview of prevailing approaches to dealing with the tricky problems of simulating a massive, large, and rapidly changing system, and provides a foundation for adopting a suitable architecture for developing a PDES.

HydraNetSim is a discrete event simulator which allows parallel simulation and exploits the capabilities of parallelization of modern computing hardware. It is based on a novel master/slave paradigm. It divides the simulation model into a number of specific slaves (a cluster of processes) considering the number of cores provided by the underlying computing hardware. Each slave can be assigned to a specific CPU on a different core. Synchronization of the slaves is achieved by proposing a variant of the classic Null-Message Algorithm (NMA) with a focus on keeping the synchronization overhead as low as possible. Furthermore, HydraNetSim provides log information for debugging purposes and introduces a new mechanism of gathering and writing simulation results to a database.

The experimental results show that the sequential counterpart of HydraNetSim (SDES) takes 41.6% more time than HydraNetSim-2Slave and 23.6% than HydraNetSim-3Slave. HydraNetSim-2Slave is 1.42 times faster, consumes 1.18 times more memory, and supports 2.02 times more nodes than a sequential discrete event simulator (SDES). Whereas, HydraNetSim-3Slave executes 1.24 times faster, consumes 2.08 times more memory, and supports 3.04 times more nodes than SDES. The scaling factor of HydraNetSim is [($\beta$-1)*102.04%] of the maximum numbered of nodes supported by SDES; where $\beta$ is the number of slaves.

# Sammanfattning

Diskret händelsesimulering är den mest passande typen av simulering för att analysera ett komplext system där förändringar sker i diskreta tidpunkter. Diskret händelsesimulering är en stor experimentell metod i flera vetenskapliga och tekniska områden. Tyvärr kan en konventionell diskret händelse simulator uppfyller inte med ökande krav på beräkningsprogram eller strukturella komplexiteten av moderna system som peer-to-peer (P2P) system, och därför parallellt diskret händelse simulering har varit ett fokus för forskare under flera årtionde.

Tyvärr ingen simulator ansåg som en standard som kan uppfylla kraven på alla typer av applikationer. Så samtidigt få en simulator ger bra prestanda för en specifik typ av applikationer kan det inte vara effektivt för andra typer av applikationer. Även om tekniska framsteget har gjorts i multi-core datorhårdvara, är ingen av de vanliga P2P händelsestyrd simulatorer för att stödja parallella simulering som utnyttjar flera kärnor arkitekturer. Den föreslagna HydraNetSim parallella diskret händelse simulator (PDES) är ett steg mot att fokusera på dessa frågor.

Utveckla en simulator som kan stödja ett mycket stort antal noder för att realisera en massiv P2P-system, och kan även utföra parallellt är en icke-trivial uppdrag. Litteraturstudien i denna tesen ger en bred översikt över aktuell metoder för att hantera de svåra problem som simulerar en massiv, stor och snabbt ändra system och ger en grund för att adoptera en passande struktur för att utveckla ett PDES.

HydraNetSim är en diskret händelse simulator som gör det möjligt parallellt simulering och utnyttjar funktionerna i parallellisering av modern datorhårdvara. Det är baserat på en ny master / slav paradigm. Den delar simuleringsmodellen i ett antal specifika slavar (ett kluster av processer) med tanke på antalet kärnor som tillhandahålls av den underliggande datorhårdvara. Varje slav kan tilldelas en specifik CPU på en annan kärna. Synkronisering av slavarna uppnås genom att föreslå en variant av det klassiska Null-Message Algorithm (NMA) med fokus på att hålla simuleringen overhead så lågt som möjligt. Dessutom ger HydraNetSim log information för felsökning ändamål och inför en ny mekanism för att samla in och skriva simuleringar resultat till en databas.

De experimentella resultaten visar att den sekventiella motsvarigheten till HydraNetSim (SDES) tar 41,6% mer tid än HydraNetSim-2Slave och 23,6% mindre än HydraNetSim-3Slave. HydraNetSim-2Slave är 1,42 gånger snabbare, förbrukar 1,18 gånger mer minne, och stöder 2.02 gånger fler noder än en sekventiell händelsestyrd simulator (SDES). I HydraNetSim-3Slave kör 1.24 gånger snabbare, förbrukar 2,08 gånger mer minne, och stöder 3,04 gånger fler noder än SDES. Skalfaktorn av HydraNetSim är [($\beta$-1)*102.04%] av den maximala numrerade noder som stöds av SDES; där $\beta$ är antalet slavar.

*Dedicated to…*

*my guiding light*

# Acknowledgements

All praises and thanks to Allah the Almighty. This thesis would not have been possible without His blessings.

I would like to acknowledge the support, wisdom, and encouragement given by many respected and loving people around me. I would like to start by expressing my deepest gratitude to my immediate supervisor Mr. Christof Leng, currently a PhD candidate at the Databases and Distributed Systems Group (DVS) at TU-Dramstadt, for his invaluable assistance, guidance, and mentorship from the preliminary to the finale phase of this work. His willingness to give his time so generously has been very much appreciated. I am grateful for having the opportunity to learn from him and work with him.

I owe my deepest gratitude to Prof. Gerald Q. Maguire Jr., my supervisor at KTH, for guiding me all the way through my thesis. His valuable comments helped me a lot in improving and polishing the whole thesis and the overall format of the report in general and some steps such as the analysis, in particular.

I would like to extend my thanks to DVS for providing me the opportunity and necessary facilities to conduct my thesis project.

My grateful thanks are also extended to all the people that I got to know during my stay in Stockholm. Among these people, I wish to express my special thanks to Abd-ur-Raheem, Ahmad Kamal Mirza, Waqas Liaqat, Shaqriq Mobeen and Muhammad Muaz for their moral support and guidance. I also wish to express my gratitude to Muhammad Rashid Idress for his valuable recommendations to improve this thesis.

Special thanks to my family, in particular to my parents for their support and unconditional love. I would not be the same without their support and encouragement.

Thank you, really.

Stockholm, Sweden Muhammad Fah'd Azeemi
July 2012

# Table of Contents

# Table of Figures

# Table of Tables

# List of Acronyms and Abbreviations

| | |
|---|---|
| **ATM** | Automatic Teller Machine |
| **CAD** | Computer Aided Design |
| **CMB** | Chandy-Misra-Bryant (protocol) |
| **CPU** | Central Processing Unit |
| **CUSP** | Channel-based Unidirectional Stream Protocol |
| **DB** | Database |
| **DVS** | Databases and Distributed Systems Group |
| **EIT** | Estimated Input Time |
| **EOT** | Estimated Output Time |
| **FEL** | Future Event List |
| **GB** | Gigabytes |
| **GVT** | Global Virtual Time |
| **IP** | Internet Protocol |
| **LBTS** | Lower Bound on Time Stamp |
| **LCM** | LP Configuration Message |
| **LH** | Lookahead |
| **LP** | Logical Process |
| **MTW** | Moving Time Window |
| **NMA** | Null-Message Algorithm |
| **PDES** | Parallel Discrete Event Simulator |
| **RAM** | Random Access Memory |
| **RNG** | Random Number Generator |
| **RPC** | Remote Procedure Calls |
| **SCTP** | Stream Control Transmission Protocol |
| **SDES** | Sequential Discrete Event Simulator |
| **SST** | Structured Stream Transport |
| **TCP** | Transmission Control Protocol |
| **TU D** | Technische Universität Darmstadt |
| **UDP** | User Datagram Protocol |

# Chapter 1.

# Introduction

Distributed systems and computer networks have sparked the zeal of a huge community, and today applications of distributed systems range from telecommunication to the air traffic control systems [1]. This diverse range of applications exhibits the growing popularity of distributed systems. This ever growing popularity has been a continuous source of attraction for computer scientists to explore the new dimensions of this field. Different reasons can be cited for the need of such distributed and networked systems, some of these are:

- The nature of the application may require it to be run on a communication network.
- To avoid a single point of failure.
- Economics: a collection of several multiprocessors offers better price-performance ratio than a large mainframe [3].
- However, the sharing of resources is considered the main motivation behind the construction of distributed systems [2].

Unmitigated access and a complete control over a real network or distributed system is presently only a dream for researchers [22], due to the very turbulent and complex nature of such systems. Evaluation and analysis of the operational behavior of a system has always been an essential part of gaining an ample grasp of such a system, both in industry and academia. Managers and analysts need to estimate the expected operational behavior of the system in order to make correct and timely decisions, and to make changes in the architecture of existing systems or a system under development. Three different methodologies are usually applied for this purpose [4]:

1. Conducting experiments with prototypes or real systems,
2. applying mathematical analysis, and
3. simulation.

There are financial and technical constraints involved with the first approach, as it is not an easy task to prototype a heterogeneous system which is highly dynamic in its nature. It is also not convincing to disturb the individual components of a real system as the risk of failure of the experiment for one reason or the other may be high, and the modifications may seriously disrupt the system altogether. Further, there may be some situations which cannot be explored by experimental means, such as the study of evolution of galaxies as this cannot be done pragmatically by experiments [23].

The other two techniques (i.e., analytical modeling or analytical analysis and simulation model) are the specialized forms of mathematical modeling which enable an experiment with *the model* of the system. Figure 1 shows several different choices of how to analyze a system.

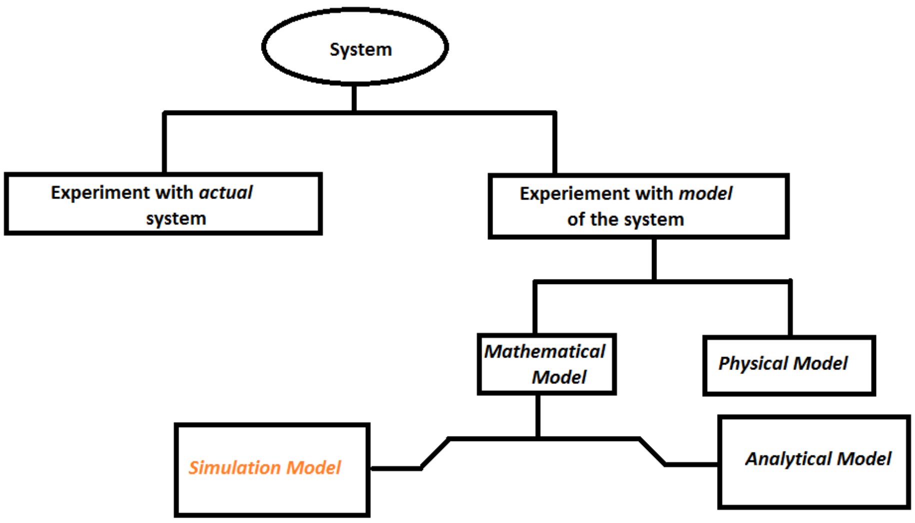

**Figure 1: Ways to Analyze a System [adopted from 13]**

An analytical solution refers to as analyzing the system by finding solutions of equations and evaluating functions. However, some systems may be too complex to represent by mathematical formulas and accordingly it may be difficult to find analytic solutions [24].

Real-time experiments can also cause safety risks in some situations. For example, testing of new equipment and procedures on avionic systems are not possible without imperiling safety. Thus, this evaluation is usually done by simulation. Simulation is a technique of imitating a system or process by another system or process [23].

For these reasons, simulation is a preferred choice for analysis particularly for the systems whose characteristics cannot be easily captured in a mathematical way [4]. Additionally, (computer) simulation is thought by many to more helpful even if analytical methods are available [23], as computer simulation can give more comprehensive insight into the system's behavior than complicated formulas written down on a paper.

Simulation plays an indispensable part in research as it offers a controlled environment for researchers to perform their experiments, enabling them to gain deeper insight into a complex system under investigation. Another important characteristic of computer simulation is its repeatability, as simulation offers the ability to test a range of designs under *exactly* the same environment conditions or to make random parameter changes to the simulation or the environment.

## 1.1 Computer Simulation

Computer simulation is a specialized form of simulation derived from the generic approach of using a *simulation model* to analyze a system. Computer simulation refers to the imitation by a computer of a real-time system or process over time [5, 6]. Computer simulation has become an essential technique in various fields over the years, including engineering, architecture, production management, entertainment, business, military, government, logistics / transportation [7], etc. Computer simulation supports simulation of large scale events which are not possible or even imaginable to be simulated by using traditional pen-and-paper mathematical modeling. This power and support for scalability has made computer simulation an essential part of ongoing research in almost every field of life. A biology project simulating 2.64 million atoms in motion to model the complex and intricate structure of ribsomes [44] is just one example of the support for large scale events that currently can be simulated by using computer simulation.

### 1.1.1 Types of Computer Simulation

Computer simulation has a truly wide range of simulation types varying from a computer program that may run for a few minutes, to a network-based group of computers or cluster of computers running for hours, to simulations which may run for days or even years. There are several types of computer simulations which are used for academic as well as for industrial purposes. Continuous simulation, Monte Carlo simulation, discrete event simulation are a few types from a vast range of types of simulations currently being employed in different fields.

#### 1.1.1.1 Continuous Simulation

Continuous simulation represents a system over time. Wouter Duivesteijn states: “Continuous simulation concerns the modeling over time of a system by a representation in which state variables change continuously with respect to time” [8]. This type of simulation may utilize game theoretic models, algebraic systems, statistical models, or differential equations. A continuous simulator applies these equations in the context of the system’s environment and generates a continuous output (often represented as a graph) reflecting the changes in the state of the system with respect to time (see the section on Continuous Simulation of C. Craig’s master’s thesis [9]) and thus, this type of simulations exhibits how the system would behave if realized. Typically, differential equations are used for describing the rate of change of state variables of the system over time.

A popular example of a continuous simulation is the predator-prey model: a biological model of competition between two populations called predators and prey, in which both populations interact with each other. The population size of predators depends upon the population size of prey and vice versa. The number of prey will decrease if number of predators increase, and will increase if the number of predators decreases. This relationship is usually examined with continuous simulation by employing partial derivatives [8].

Some other suitable candidates for continuous simulation include urban growth, population growth, weather forecasting, fermentation models, disease spread, and hurricane prediction. Continuous simulations are usually used within mathematical modeling software packages such as MATLAB® [54] or in conjunction with a computer aided design (CAD) system.

Continuous simulation is often computationally intensive, particularly when there are thousands of interconnected elements. In such a setting, continuous simulation yields slow performance and is, therefore, only useful for simulating a relatively small number of components.

#### 1.1.1.2 Monte Carlo Simulation

In Monte Carlo simulation, repeated random sampling is employed to compute the results of the simulation. This type of simulation is defined as, “a scheme employing random numbers, which is used for solving certain stochastic or deterministic problems where the passage of time plays no role” [10]. This approach is useful to simulate scenarios with considerable uncertainty in their inputs, such as calculating the risks in a business to support decision making. This approach is widely used in a diverse range of fields including mathematics, applied sciences, project management, finance, research and development, engineering, transportation, oil and gas exploration, environment, insurance, etc.

The use of random sampling numbers makes it possible to yield results which were not possible by using continuous simulation. Such a simulation generates possible results by substituting a range of possible random values derived from a random distribution over the estimated domain of input values, for a factor having a degree of uncertainty. Each simulation run uses a different set of random values from one or more random distributions to calculate a result. A typical Monte Carlo simulation run will utilize thousands or tens of thousands of recalculations, depending upon the number of uncertainties and the specified ranges for each them [11].

A disadvantage of Monte Carlo simulation is that it is computational intensive and uses a great amount of computational resources which is undesirable as a day-to-day routine practice [12].

#### 1.1.1.3 Discrete Event Simulation (DES)

Discrete event simulation is a dominant simulation technique in the field of computer networks [4]. This approach is employed when significant changes occur in the system at discrete time instances; these distinct changes at specific time points are referred to as events. Discrete event simulation represents the operations of a system as a chronological chain of events, thus every event represents some change in the state of the system.

A classic example of discrete event simulation is a queuing model of a bank’s customer’s use of the services of an Automatic Teller Machine (ATM). In this example, we consider the customers’ **arrival** and **departure** as *events*, the **customers** and ATM **machine** as *entities* of the system, and **the number of customers in a queue** and the ATM **machine's status** (idle or busy) as *states* of the system which are changed by the *events*.

Although both Monte Carlo and discrete event simulation techniques conceptually overlap as shown in figure 2, Monte Carlo simulation does not care about time and removes time from the model whereas discrete event simulation is based on the passage of time, although the rate of passage of time need not be uniform as in continuous simulation [7].

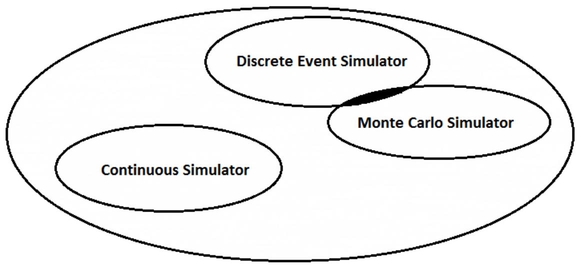

Figure 2: Models of Computer simulation (adopted from figure 1.5 in [9])

The fact that an event in a discrete-event simulation can only occur at a distinct unit of time during the simulation distinguishes it from continuous simulation in which events can also occur between the time units (see the section on Discrete-Event Simulation in C. Craig's master's thesis [9]).

Discrete event simulation is more popular than continuous simulation not only because it provides faster results, but also because it imitates the system's behavior in a generally convincingly accurate manner (see the section on Discrete-Event Simulation in C. Craig's master's thesis [9]). That is why, today, discrete event simulation is employed for research on any layer of computer networks, including signal processing in the physical layers, medium access in the data-link layer, routing in the network layer, protocol concerns in the transport layer, and design questions in the application layer [4]. This popularity is due to the fact that the discrete event simulation paradigm fits very well with nearly any system under investigation and it is also relatively easy to implement in comparison with the other techniques.

#### 1.1.1.4 Terminologies and Components of Discrete Event Simulator

Unfortunately there is no standardized set of terms and thus the naming of the components in a discrete simulator may vary in the literature [4]. The definitions employed in [5, 6] are loosely adapted in this thesis, in the paragraphs below we give a definition of the terms and components that will be used for the rest of this thesis.

An ***entity*** is an abstraction in the *system* of a particular interest, and can be described by its attributes. For example, the entity '*packet'* can be described by its length, source address, destination address, etc.

A ***system*** is composed of a set of entities and their relationships which fulfils a certain purpose (i.e., in order to achieve some desired goal of the system). A network, for example, may be considered as a *system* having routers, hosts, and links as entities and a common goal to provide end-to-end connectivity.

A ***discrete system*** is a system whose states only change at discrete points of time. These changes are triggered by the occurrence of some ***event***.

All discrete-event simulators, in general, share the following components [4]:

1. **System state** is a set of variables that describe the state of the system.
2. **Clock** represents the current time during the simulation.

3. **Future event list (FEL)** is a data structure which is used to manage the events (which will occur in the future.). ***Events*** are recorded as an *event notice* in the FEL. Each such event is composed of a *Time* and *Type* of event.

These three components are realized as data structures and formulate the core of any discrete event simulator. A discrete event simulator may also need some other data structures to perform simulation:

4. **Time-stamp:** The simulated time of the occurrence of the *event* is called a *time-stamp* of that *event*.

5. **Statistical counters:** A set of variables which contain statistical information about the performance of the system.

6. **Initialization routine:** This routine is used to initialize the simulation model and to set the clock to 0.

7. **Timing routine:** This routine retrieves the next event from the FEL and advances the clock to the time of occurrence of this event.

There are, usually, three flavors of time when dealing with the simulation:

1. **Physical Time:** This is the real (physical) time of the simulated real system.
2. **Wall clock Time:** This is the execution time during simulation.
3. **Simulation Time:** This is an abstraction of time used within the simulation for performing its operations, such as processing events.

The simulation clock keeps the current time during simulation. The value of this clock is advanced when an event is retrieved from the FEL. The system states change over time during the simulation. All the events are maintained in the FEL sorted by their time of occurrence. During the execution of the simulation, the scheduler of the simulator removes the event(s) with the smallest time-stamp from the event list and updates the simulation clock to this time.

The core algorithm of a discrete event simulator has three phases [4]:

- Initialization phase,
- Event processing phase, and
- Output phase.

During **initialization** the state variables, entities, and the clock are initialized. The simulator, then, enters the next phase based upon executing an **event processing loop**. The events are retrieved from the FEL, for processing, and an appropriate event handler is called to process each specific event on the basis of its *type*. The event handler may generate new events, can change the state variables or entities, and update the statistics as the result of processing an event.

During this event processing loop, the oldest events (the events with the smallest time-stamp) are removed from the event list. Choosing the event with the smallest time stamp (i.e., $E_{min}$) from the event list is crucial, as the event must be processed in the context (in terms of the system's state variables) of its time. Processing the events out of time order would simulate a system in which the future could affect the past, which is generally unacceptable [17], thus leading to a ***causality violation***.

According to Van Hoai Tran the algorithm for the event processing loop is (slide 15 from [19]):

```
while simulation_in_progress do
        Remove smallest time stamp event from event_list;
        Set simulation_time_clock to the time-stamp of this event;
        Execute event_handler to process this event;
end
```

In the last phase of the algorithm, statistics are computed, updated, and stored (if necessary). After doing the operations required in this phase, the simulator terminates. Figure 3 shows the flow chart of a simple sequential discrete event simulator (SDES).

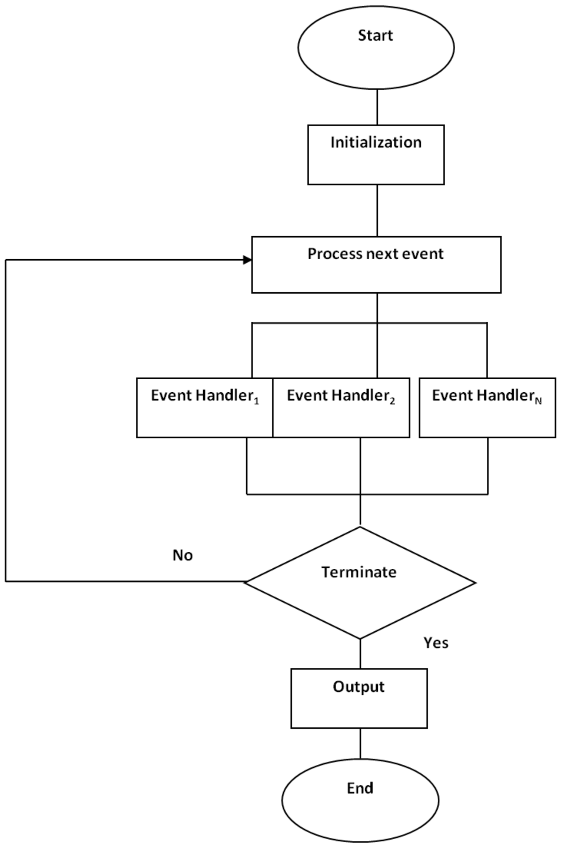

Figure 3: Flow Chart of Serial Discrete Event Simulation [adopted from 15]

This generic description of the core algorithm of discrete event simulator is not sufficient to deal with the increasing complexity of the modern (communication) systems and accordingly dealing with their models. In the next section, we will discuss the need to devise an approach to deal with the complexity of structure and increasing computational requirements of the simulation of communication systems.

#### 1.1.1.5 Case study

The Databases and Distributed Systems (DVS) group of Technischen Universität Darmstadt, Germany has developed a testbed for network (mainly peer-to-peer) applications which offers some simple interfaces for event scheduling and user datagram protocol (UDP) messaging. The implementation of these interfaces is done in two ways:

i. For real networks, i.e., for building actual applications or
ii. To support a large number of nodes running in a virtual network, i.e., running in a discrete event simulator.

These implementation schemes are used to test the same application code in:

1. a real network for getting maximum realism, and
2. a virtual environment to allow maximum configurability.

DVS has already implemented several applications, such as the BubbleStorm P2P overlay [50], CUSP: a TCP-like transport protocol [46], Kademlia peer-to-peer (P2P) overlay [51], and even a P2P multiplayer game called Planet PI4 [52] on the top of these interfaces. All of these applications run both in the simulator as well as on real networks. The simulator allows testing the applications on large networks with a high degree of realism [53].

This simple testbed can be used to simulate an application developed by using the interfaces provided by this testbed. Furthermore, the application must be written in an asynchronous, event-driven fashion. This requirement for asynchronous, event-driven operation of the application allows the simulator to run many nodes in parallel without having any time conflicts. For further details about this testbed, please read [53].

#### 1.1.1.6 Limitations of Sequential Discrete Event Simulation

Modern communication systems are becoming increasingly complex which is accordingly increasing the complexity of the evaluation tools used to study them. This complexity can either be computational complexity or structural complexity. Structural complexity of a simulation model is due to the growing size of a simulated network.

Distributed systems and massive peer-to-peer systems have caused a gigantic boost in the size of communication systems that researchers wish to simulate. The complex behavioral characteristics of such large systems are impossible to observe by using a smaller sized network as a test-bed or when the researcher lacks an accurate analytical model. This complexity requires a network simulation model and approach which can support a huge number of simulated network nodes. Here a problem arises: all of these network nodes need to be represented in memory and they will trigger events in the network simulation model. Both of these factors will significantly increase memory consumption and makes the network simulation model computationally intensive. Furthermore, the simulation of complex real-time systems, such as air traffic control systems may require hours to complete, whereas the decisions need to be made within minutes.

The shortcomings of classic or sequential discrete event simulation can be classified under the following two perspectives [17]:

- Academic and
- Industrial or pragmatic

However, both settings (academic and industrial) have their own reasons for developing a **P**arallel **D**iscrete **E**vent **S**imulators (PDESs). From an academic point of view, a PDES represents a problem domain which may offer substantial parallelism [17]. From an industrial point of view, to evaluate the large and complex models in sectors such as engineering, military, economics, defense, government, etc. by employing sequential simulation tools may take hours (see slide # 2 from [19]) which is unacceptable as the insight from the simulation is too late to be relevant. Therefore, industrial users want to have an efficient solution which can address the growing computational and structural complexities of the systems that they wish to consider in time to enable them to make critical decisions.

All these issues are addressed by enabling the simulation to be executed in parallel or on multiple processing units. We can only meet the demand for extremely large amounts of memory either by employing a cluster of processors or by combining the memory and computational resources of a number of processing units. Additionally, we can utilize these many processors to provide the computational resources required for processing simulations of specific applications. Luckily, modern hardware developments and technological advancements have greatly reduced the cost for parallel computing hardware, thus making such hardware available to a larger research community. These different aspects have all contributed to the development of PDES.

## 1.2 Problem Statement

A case study was explained in detail in section 1.1.1.5. Although this testbed works well for small applications, it does not perform sufficiently to allow simulation of larger applications. This poor performance is due to the fact that currently the simulator is single-threaded, making it very slow when simulating larger networks. Furthermore, the simulator can only scale up to a couple of thousands nodes because of the relatively low-level interfaces that have made few simplifications from an actual program.

These limitations limit the ability to examine the behavior of extremely large networks, such as massively multiplayer online games (MMOG) or peer-to-peer (P2P) systems. Therefore, there is a need for a means to evaluate large applications running on very large distributed systems.

The increasing complexity of P2P applications demands a parallel version of the simulator to run on many processors or even on clusters of computers. This parallel simulator should support hundreds of thousands of nodes in the simulator. This parallelization will allow exploring large-scale scenarios with hundreds of thousands of nodes with a relatively high degree of realism, and will enable researchers to realistically evaluate the behavior of larger P2P networks.

Furthermore, technology advances have made relatively powerful multi-core computing hardware available to a larger research community. According to recent surveys [65 - 68], none of the main stream P2P simulators has been designed to exploit the multi-core capabilities of modern computing hardware [64]. Therefore, a PDES should be designed to exploit such multi-core architectures.

The research question for this thesis is to design, develop, and evaluate a PDES for examining the behavior of larger network applications and P2P systems, which can exploit the multi-core architecture of modern computing hardware while keeping all the Logical Processes (LPs) in synchronization.

### 1.2.1 Hypotheses

Studies [4, 33, and 64] suggest that the parallelization of a (sequential) discrete event simulator boosts the simulation speed and enable simulations on a larger scale (in terms of the number of simulated nodes). Therefore, our hypotheses for this research will be:

1. HydraNetSim (a PDES) speeds up simulation in comparison with the SDES given that the experiment settings are same, and
2. HydraNetSim scales up the simulation by enabling the simulation of more nodes than SDES for a given hardware platform.

### 1.2.2 Goals

The overall goals of this master's thesis project will, therefore, be to design, develop, and evaluate a parallel version of existing testbed so it can be run on many processors utilizing different cores or computing clusters.

This PDES will preserve the correct simulation semantics by ensuring that all the processors are fully synchronized. The simulator's output will be collected and aggregated from all the distributed processes. This output will then be written to a database so that these results will be available for future verification and so that the behavior of the system can be examined thoroughly.

We can categorize the goals of this master's thesis project as:

- The designed simulator must be capable of exploiting the multi-core architecture of modern computing hardware,
- To increase the scalability of the existing simulator in such a way that it can support extremely large numbers of nodes (where extremely large is defined as hundreds of thousands of nodes),
- The nodes should be distributed in the simulator over a cluster of simulation processes,
- The random number generators must generate the same sequence of numbers, for each simulated node, independent of the number of processes used for the simulation (This is necessary for getting same simulation results independent of the partitioning over one or more machines), and

- The output of the simulation run must be collected and aggregated from the worker processes.

We will examine of all of these goals and elaborate the challenges in Chapter 3 of this thesis.

### 1.2.3 Blueprint

The starting point for developing the proposed PDES will be a literature review and related work done by other researchers. This review will not only provide a comprehensive illustration of general concepts of the field, but will also provide an ample description of the architecture and necessary components of a PDES.

We will explore the underlying approaches proposed and adopted by different researchers for developing a PDES in general. Next, we will analyze and compare all of these state-of-the-art approaches while considering all of their design aspects in the context of our thesis goals. In this way, the literature study will provide a foundation for implementing and evaluating our proposed architecture.

This literature review and study of contemporary design approaches will assist us in formulating a systematic approach to design the architecture of our PDES, along with procedures for keeping all partitions synchronized, and for gathering and writing the simulation results into a database (DB). Consequently, we will implement our proposed PDES (to be called HydraNetSim) in order to achieve our goals.

Finally, we will evaluate HydraNetSim in order to examine the behavior and performance of our adopted approach. We will examine the performance improvements after parallelization and validate our hypotheses as defined in section 1.2.1.

The proposed plan for this thesis project consists of:

- A literature review,
- Analytic study of contemporary approaches,
- A study of related work,
- Design our proposed PDES architecture,
- Implement this proposed architecture,
- Conduct a performance evaluation of our proposed PDES, and
- Analysis of this evaluation will be performed in order to draw some conclusions and to suggest future work.

### 1.2.4 Scope

This thesis will provide a comprehensive understanding of how a SDES actually works, what its key components are, what are the core challenges, and most importantly what the core challenges are in developing a PDES to be run on a multi-core computing platform, and how to deal with the relevant challenges.

This thesis provides an analytic study of popular approaches for developing a PDES. This study examines the advantages and disadvantages of all these approaches. All of these approaches belong one of two broader categories:

1. Conservative, and
2. Optimistic.

For this reason, a comprehensive comparison of these two categories is provided for researchers to understand the advantages and disadvantages of each approach. This study will assist researchers for preferring one on the other category of PDES.

Furthermore, a brief overview of some recent surveys [65-68] of existing mainstream P2P simulators has also been included in terms of their scalability, architecture, programming language, and PDES capabilities.

### 1.2.5 Target Audience

This thesis, primarily, will assist researchers from the area of PDES in particular and computer simulation in general. The developers of PDESs in general might utilize the results presented in this thesis due to its novel architecture, synchronization technique (which minimizes the synchronization overhead), its new idea of gathering simulation results over a specific log period before writing them directly to the DB, and more importantly its capabilities of exploiting the parallelization capabilities of the underlying computing hardware.

### 1.2.6 Contribution

The main contribution of this thesis is its novel and innovative overall design and architecture. This thesis should spark the interest of the research community because of its novelty regarding:

- Capability of exploiting the multi-core architecture of modern computing hardware,
- Architecture,
- Synchronization technique,
- Approach to rely on null messages as little as possible,
- Effort to minimize synchronization overhead, and
- Mechanism of gathering simulation results over a specific log period before directly outputting them.

### 1.2.7 Outline

The structure of this thesis work is as follows: Chapter 2 includes comprehensive details of a PDES (its core components and concepts, its working), core challenges to developing a PDES, popular approaches for coping with these challenges, and comparison of these approaches. Chapter 3 provides the overview of some recent surveys on existing mainstream P2P simulators. Chapter 4 provides the objectives and goals of this thesis project, proposed model and design solutions, along with implementation details (of its architecture, core components and concepts, and proposed approaches for completing the established objectives and goals such as strategies for message routing, synchronization, determinism, etc.). The experiment details, performance and scalability evaluation, and analysis of results are presented in chapter 5. Finally, Chapter 6 offers some conclusions and suggests some future work.

# Chapter 2.

# Literature Review

The benefits promised by computer simulation and the shortcomings of SDES have caused a lot of research attempting to devise techniques to meet the challenges posed by the need to simulate complex and large communication systems. PDES emerged from this research and has been the focus of researchers for several decades [20]. The main motivations behind the development of a PDES can be summarized as:

- To simulate systems which are highly complex in their structure and turbulent in their nature, such as the internet [18],
- To use the simulator as a forecasting tool for making time critical decisions, such as in the case of air traffic control systems,
- The high computational demands posed by simulation of complex, large, and high-resolution applications [16],
- Enabling interoperability, i.e., in order to connect a number of autonomous simulators running on geographically distributed machines, with each of these distributed machines simulating a different and distinct component of a large and complex system,
- To achieve a better resource sharing, by connecting and running multiple of simulators in parallel in order to execute a single over-all model,
- To reduce the execution time of a complex model. The time required for a simulation run can be reduced in proportion to the number of nodes or processors it is allocated [16],
- Larger models can be simulated by combining the memory capacity of all processors, and
- To exploit the great resources of some special nodes, for example to meet high graphics requirements or handle very large amounts of data.

We will first discuss the basic concepts and architecture of a PDES and then will give an overview of some contemporary approaches to deal with the challenges of developing a PDES.

## 2.1 Parallel Discrete Event Simulator

A parallel version of a sequential discrete event based simulator can be viewed as a collection of a number of sequential discrete event based simulators, executing on different processors which communicate with each other only through *time-stamped* messages (called events). Such a PDES can be defined as a PDES "which divide[s] a simulation model into multiple parts which execute on independent processing units in parallel." [4]

### 2.1.1 Architecture

The model of the system is decomposed into sub-models. These sub-models are usually called partitions, which are finite in number and they are created by applying some specific partitioning scheme. These partitioning schemes are generally categorized into the following three classes [4]:

1. **Channel Parallel Partitioning:** The channel parallel partitioning scheme is based on the assumption that the data which is transmitted on different radio channels or media does not interfere, and hence remain independent. Thus, the events occurring on non-interfering nodes are considered independent. On the basis of this assumption, the simulation model is decomposed into finite groups of non-interfering nodes. The problem, nonetheless, involved with this scheme is that we cannot generally apply this scheme to every simulation model, as its paradigm suits only some special models [4].
2. **Time Parallel Partitioning Scheme:** The time parallel partitioning scheme divides the simulation time of a simulation run into equally sized time-intervals. It assumes that the state of the simulation model is already known at the beginning of each interval and, thus, simulation of each interval is considered independent from other intervals. The drawback of this scheme is that a network simulation usually has significant complexity and thus its state at specific points in time is hard to know in advance. Therefore, this scheme is in general impracticable for network simulation [4].

   **Space Parallel Partitioning:** The simulation model is divided, in this scheme, into multiple finite partitions by considering the connections between simulated nodes. The result of this partition scheme is that each cluster of nodes that tightly communicate is placed into a single partition (see Figure 4). This paradigm suits the network simulation model in general and can be applied to any complex simulation model [4].

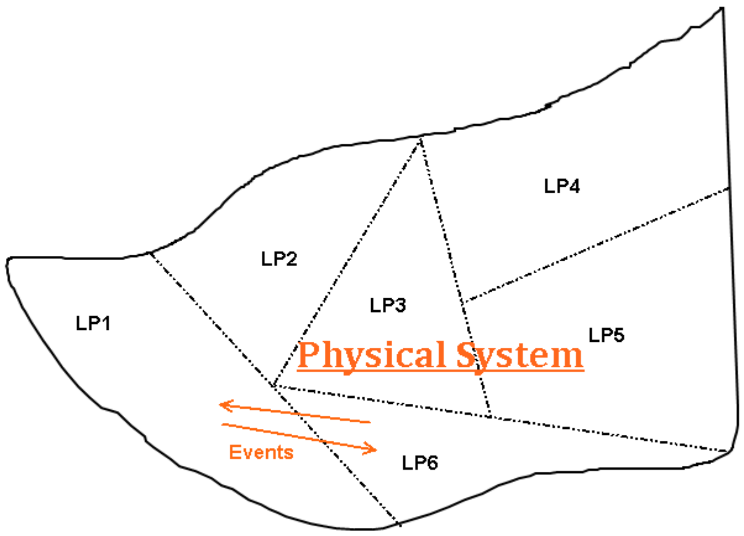

Figure 4: Space Parallel Partitioning

The system under examination is usually referred as a physical system. This physical system is viewed as being composed of some finite number of physical processes which interact with each other at various discrete points in simulated time. The simulation model is partitioned with respect to these physical processes in such a way that each partition is mapped to exactly one physical process. These partitions are subsequently executed by a run-time component called a **L**ogical **P**rocess (LP) [4]; each LP models a different part of the (physical) system. Furthermore, each LP maintains its own time-stamped event list, local clock, and state variables. Thus, an LP resembles a SDES by suitably maintaining all these three data structures.

The inter-LP communication is restricted to sending time-stamped messages, usually through *FIFO channels* to preserve the local FIFO characteristics. These FIFO characteristic ensures that the messages will arrive at the receiving LP in exactly the same order in which they were sent.

Limiting communication to time-stamped messages facilitates parallelization of asynchronous system simulation where events occur at irregular time intervals and are not synchronized to any global clock. However, this requires some mechanism to avoid possible causal violations and to address the synchronization overheads. Further, this paradigm demands concurrent execution of events, which may also cause synchronization problems. In the next section, we will discuss the challenges to creating a PDES and describe some of the prevailing schemes that have been introduced to deal with these challenges.

### 2.1.2 Core Challenges to PDES

In this section, we will discuss some important challenges to developing a PDES and some proposed solutions to deal with these challenges.

#### 2.1.2.1 Support user behavior

The modeling of systems is a crucial challenge for obtaining realistic results when estimating the performance of large scale P2P systems [4]. A proper evaluation of a system must also take user behavior into account, along with modeling of the underlying network. A good model of user behavior is an important factor for modeling network applications in general and for P2P applications (where participating peers are consumers and providers at the same time, in their nature) in particular. Therefore, a PDES should support the inherent property of user behavior modeling of a (P2P) system to provide realistic results.

The behavior of P2P users is rather complex [4] and for simplicity it can be broken down into the following three independent components:

1. **Churn**

   Users join the network, leave, and may rejoin several times. Sometimes they even leave the system for-ever and do not return. For example, the users of systems such as BitTorrent [61] are interested in exactly one download per torrent and after downloading that particular torrent they usually do not rejoin that particular distribution overlay.

   Churn consists of the complete lifetime of a P2P node which starts when that node initially joins the system and ends when the node permanently leaves the system.

During this lifetime, however, the node may go through several online and offline cycles as shown in figure 5.

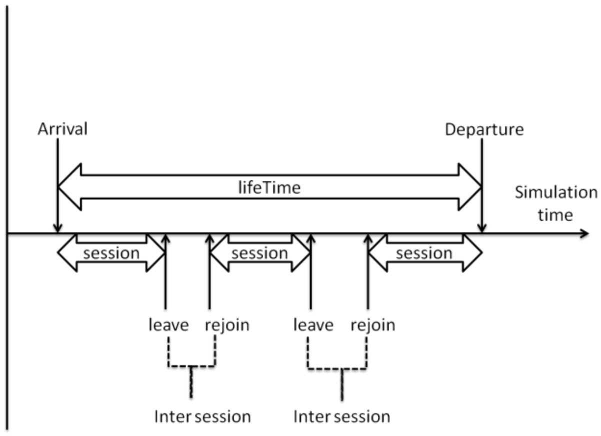

Figure 5: Lifetime of a peer (adopted from figure 20.2 in [4])

The time span a node is online is referred to as a session [4], and the time between sessions is called an intersession. In general when a node orderly leaves the system it informs and sends the notifications to its neighbors assisting them to reorganize their state, routing table, etc. But it may also happen that a node simply crashes, i.e., it disappears from the overlay without any notification (because of a software crash, hardware crash, loss of connectivity, etc.).

2. **Workload**

   A realistic workload model is very important for the performance evaluation of any system. Workload specifies the provision and consumption of resources in the given system. Interestingly, workload is not uniform over all the resources [60], as some resources are scarce and many are abundant. Furthermore, many resources may be far more popular than others.

3. **Use properties**

   Generally a user does not have uniform interest in all resources rather s/he consumes very specific resources. This interest clustering is very important for the reputation system and has been used to build specialized content-clustered overlays [61]. Furthermore, many users are not willing to cooperate and try to maximize their own benefit, e.g., by not uploading anything in return for downloading some content. Yet, in contrast there are many users who do not leave the system immediately after a complete download [62], thus their resources continue to be available. That is how user strategy influences both the workload and churn [4].

Modeling of all the above mentioned components is crucial to obtain realistic results from a simulation run, and for evaluation of a model of a (network or P2P) system. Therefore, support of user behavior is a critical challenge in developing a PDES.

#### 2.1.2.2 How much speedup?

Equally important challenge in developing a PDES is to understand how much speedup one can achieve by running the simulation in parallel? The developers/researchers should keep in mind that one cannot improve the rate of completion of a (simulation) task by merely adding more processors or by running the program by multiple processors in parallel.

Each program can be divided into two fundamental parts: sequential and parallel. Parallel portions are the fractions of a program which can be run on many processors at the same time, and independent of each other. Sequential portion is the fraction of the program which cannot be run in parallel and concurrently. For example, if task B is dependent on some task A for any reason such as lock retrieval, synchronization etc., then they must be run sequentially (i.e., task B can only be processed after task A) [87].

It is inferred by the arguments of G.M Amdahl [85] that the speedup of a program using multiple processors in parallel processing (computing) is limited by the sequential portion of that program. For example, if a parallelized implementation of an algorithm (for a given problem size) can run 25% of the algorithm's operations quickly while the remaining 75% of the operations is not parallelizable, then Amdahl's law states that the maximum speedup of the parallelized version of the algorithm is 1/(1 - 0.25) = 1.33 times faster than the non-parallelizable version. In the case of parallelization, if P is the fraction of the program that can be made parallel and (1 - P) is the fraction that remain serial, then the maximum speedup that can be gained by using *N* processors is

$$S\,(N) = \frac{1}{(1-P)+\frac{P}{N}} \qquad 3.1$$

By solving the equation 2.1 with various values of *N* and keeping *P* constant, we can observe the diminishing return effect, i.e., at some point[1] it will not be speedup any faster than the sequential portion of the program ( 1 - P), as shown in Figure 6.

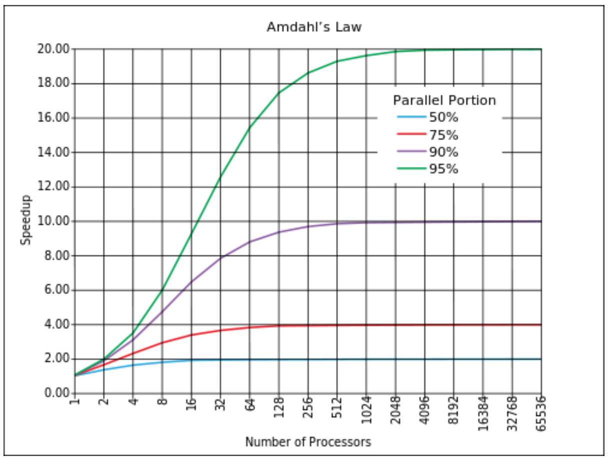

**Figure 6: Amdahl's Law [88]**

In the words of Fred Brooks, "When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned." [86]. Although Brooks made this statement to illustrate the management of software projects, yet the principle

[1] If we increase the value of N and keep the value of P fixed, then ultimately for the bigger values of N (approaches to infinity) P / N will approach to zero, and thus will not have any (significant) effect on speedup from that point on.

applies in hardware/software architectures, as well. To establish the fact, let's use a simple Gedanken-experiment to see how adding more and more processors for the execution of a program does not help much in speeding up the rate of time of execution. Assume that a given program takes 30 minutes for the execution of its parallel portion and 10 minutes for the execution of sequential portion. Intuitively, running the program on a single processor will take 40 minutes, whereas if we run the program on 10 processors in parallel then the execution time significantly drops to the 13 minutes. Now, if we run the program on 30 processors, then the execution time will drop to 11 minutes; the addition of 20 more processors just reduces 2 minutes. Table 1 illustrates the effects on the execution time of a given problem by adding more and more processors. Note that by increasing the number of processors from 100 to 1000 (a 10 times increase) we can achieve only 0.97 times speedup.

Table 1: Effects of adding processors on execution time of a given problem

| **Number of Processors** / **Execution Time (in minutes)** | **1** | **10** | **30** | **100** | **1000** |
|---|---|---|---|---|---|
| **Parallel Portion** | 30 | 3 | 1 | 0.3 | 0.03 |
| **Sequential Portion** | 10 | 10 | 10 | 10 | 10 |
| **Total** | 40 | 13 | 11 | 10.3 | 10.03 |

The researchers/developers should pay due intention on the fact whether there simulation model has much potential for parallelism or not; equally important, they should also consider the Amdahl's law while designing the architecture of PDES, and should adopt a scheme that facilitates the parallel execution of simulation model rather than hindering it. For example, if task A is dependent on task B for any reason such as lock retrieval etc., then the developers should not split the simulation model in such a way that task A and task B are assigned to two different LPs. As if they do so, then an LP can (unnecessarily) become dependent on some other LP, such as waiting for the retrieval of the lock etc., which will cause additional delay and will affect the overall execution time of the simulation run; by assigning both the tasks to the same LP can save this unnecessary block.

#### 2.1.2.3 Adherence to the causality constraint

Another crucial challenge in developing a PDES is to adhere to the causality constraint. The algorithm described earlier for a sequential discrete event simulator, and accordingly its model, does not support parallel execution of processes. Naïvely attempting to use this algorithm in parallel may yield frequent causal violations. To illustrate this, consider the following scenario (taken from [4]): we have *N* number of processors to run the discrete event simulation in parallel. The central scheduler will be continuously removing events from the event list. Thus, *k* events may be processed by *k* processors in parallel at any time *t*.

Consider that we have two events $E_1$ and $E_2$ with time stamps $T(E_1)$ and $T(E_2)$ respectively such that

$$T(E_1) < T(E_2)$$

Both events are assigned to different CPUs to process them, in time stamp order. Now suppose a new event $E_3$ is generated by the processing of $E_1$, with time stamp $T(E_3)$ such that

$$T(E_1) < T(E_3) \textit{ and } T(E_3) < T(E_2)$$

Then a problem will arise because $E_2$ has already been scheduled to be processed and the processing of $E_2$ may cause changes in the system variables that $E_3$ depends on, thus leading to a causality violation. Hence, the fundamental challenge is to decide upon the execution of two events $E_1$ and $E_2$ in such a way that

i. If both events do not interfere then they may be executed in parallel, and
ii. If they have a dependency, then they must be executed in the correct sequential order.

FIFO channels (queues) are used for inter-LP communication in the underlying PDES architecture, to take the advantage of the FIFO characteristics of a queue [4]. This FIFO characteristic suggests a means to prevent the causal violation in the parallel execution of a discrete event simulation. This idea is called Local Causality Constraint, and is defined by Overeinde, et al. as:

> "A discrete-event simulation, consisting of logical processes that interact exclusively by exchanging time stamped messages obeys the local causality constraint *if and only if* each LP processes events in non-decreasing time stamp order." [31]

This constraint assures no causality violations if one adheres to this local causality constraint. Intuitively, this constraint ensures that '*cause must precede the effect*'. To illustrate this, consider two events $E_1$ and $E_2$ with time stamps 10 and 20 and scheduled on $LP_1$ and $LP_2$ respectively. Now consider that the processing of $E_1$ may generate another event $E_3$ having a timestamp lower than 20, say 16 and that execution of this event is scheduled on $LP_2$. The local causality constraint demands the processing of $E_3$ before $E_2$, and thus ensures sequential execution of all three events in non-decreasing time stamp order.

It is important to note that the local causality constraint is a *sufficient* condition and not a *necessary* condition [32] as two events in the same LP may have different timestamps without any direct or indirect dependency between them. Thus, the execution of such events in parallel will not cause any causality violation.

The two fundamental properties of correctness called *liveliness* and *safety*, of a PDES can be defined by taking local causality constraint as an underlying condition:

**Liveliness:** Each event in the event list will *eventually* be processed successfully in correct timestamp order.

**Safety:** *Cause* must always precede the *Effect*; each LP should process events in non-decreasing time stamp order.

The liveliness property ensures that the PDES proceeds, whereas safety guarantees that no causal violation will happen in the overall execution of the PDES. Both properties complement the technique of using a local causality constraint to tackle the challenge of causal violation. However, some questions still needed to be answered, such as whether two events $E_1$ and $E_2$ with the same timestamps, should be executed in parallel or in sequence? Another question is how can we know the events' outputs (i.e., whether any of them will produce another event having a timestamp lower than another event's or not?) in advance, i.e., without processing the events. A concrete example of this fundamental dilemma can be illustrated by a battlefield simulation in which two tanks A & B are on opposite sides of the battle. Assume that the time taken by the bomb shell from one tank to reach its enemy is one second. Now, suppose that in event $E_1$: *tank A fires at tank B with a 97%* probability of a hit at timestamp 00152 and $E_2$: *tank B fires at tank A with a 99%* probability of a hit at timestamp 00170 (with both timestamps in units of seconds). If processing of $E_1$ succeeds in annihilation of tank B, then there is no sense in processing $E_2$. But this fact can be known only after processing the $E_1$, thus the decision to process or discard $E_2$ is causally dependent on the processing of $E_1$.

Questions, such as how can we know about the system's state beforehand and without actually performing the simulation? How to avoid causal violations? How to decide about the concurrent execution of events? How to achieve a synchronous execution of parallel discrete event simulation? have been the focus of researchers for decades. Many approaches have been proposed to deal with these challenges. These approaches can broadly be categorized under the following two categories [17]:

1. Optimistic
2. Conservative

Optimistic approaches relax the local causal constraint and allow the causal violation to happen for a while. They use a '*detect* and *recover'* approach to deal with these causal violations, then a rollback mechanism is invoked to recover the earlier (correct) system state whenever a causal violation is detected [31]. The conservative approach, on the other hand, adopts a strategy where any causal violation is strictly avoided during simulation by using some mechanism to determine when it is safe to process an event in order to avoid the possibility of any causal violation ever occurring.

### 2.1.3 Optimistic Approach

In this group of protocols, the synchronization mechanism does not restrict LPs from receiving and processing events as they arrive. As this approach does not enforce a local causality constraint, it may cause a causality violation. The aim of this approach is greedy execution and the maximum utilization of the partitioning by allowing the LPs to advance based upon an optimistic assumption that no event will cause a causal violation.

The advantages of this approach are that it offers a potentially larger speedup than conservative approaches and needs no prior knowledge about the possible interactions between LPs. The disadvantage of this approach, however, is that a possible causal violation will leave the system in an incorrect state. To address this shortcoming, the optimistic algorithms offer a mechanism to recover from this incorrect state by requiring that the PDES engine continuously stores the simulation state, then whenever a causal violation is detected the recovery mechanism rollbacks the system to the last state which is known to be correct.

## 1. Time-warp algorithm

A well known optimistic algorithm is the *Time-Warp Algorithm* proposed by Jefferson Sowizral in 1985 [33]. In this approach, each LP is allowed to aggressively process its local events and to send new messages generated by the event being executed, to other LPs. However, when an event arrives at any LP, which has a timestamp smaller than the local simulation time of that LP, then a causality error is triggered; such an event is called straggler. To understand the algorithm that is used, consider an LP, say $LP_i$ whose local clock is $T(LP_i) = 025$. Now assume that it receives an event $E_k$ with timestamp $T(E_k) = 023$, thus

$$T(E_k) < T(LP_i)$$

This means that the simulation state may be incorrect following the simulated timestamp $T(E_k)$. To recover from this potential error, $LP_i$ needs to restore the simulation to a correct state by performing a rollback to the saved state of a simulation time no later than $T(E_k)$ then restarting the simulation from that state. This latest known correct state is also referred to as the *last well known checkpoint* [20]. These checkpoints are maintained by periodically saving the state of each LP.

It may happen that $LP_i$ has already sent messages to other LPs after the simulation time $T(E_k)$, thus potentially causing them to be in an incorrect state. Clearly, the subsequent rollback operation must include those LPs as well. Hence, $LP_i$ sends a special message called an *anti-message* to all such LPs. When an LP receives an *anti-message*, it examines whether it has processed the corresponding *positive* message or not. If it has not processed the corresponding positive message, then the *anti-message* and the (original) *positive* message will cancel each other. Otherwise, the given LP will have to perform a similar rollback operation and may have to send *anti-massages* to other LPs as well. This process ensures correctness at the cost of recursively throwing out all the incorrect states and messages out of the system.

### Performance considerations

One major drawback of this class of protocols is that it requires a significant amount of hardware resources for storing the simulation state checkpoints. Further, some operations such as input and output operations cannot be rolled-back [32].

This problem, however, is handled by introducing the notion of Global Virtual Time (GVT) which is determined by the smallest timestamp of all the unprocessed events in the simulation. The track of GVT is kept for ensuring that there will be no roll-back to prior to

GVT. Intuitively, all but one of the saved states with timestamps smaller than GVT can be safely discarded safely and thus it is sufficient to save only one state having a timestamp no greater than GVT. Using this approach, I/O operations can be committed only when GVT will have advanced beyond the simulated occurrence times of these operations [32].

A significant decrease in simulation performance is caused by the overhead of *anti-messages* and frequent *roll-back* operations. Many improvements have been suggested by researchers; we will briefly overview some of these suggestions.

### 2. Lazy cancellation

The original Time-Warp algorithm uses the aggressive approach employing cancellation and a rollback strategy which causes an extra influx of anti-messages and recurrent rollback calls. In contrast, Lazy Cancellation does not send the anti-messages immediately after receiving a straggler message, but rather asks the LP to rerun the simulation after rollback and examine if this rerun also generates the same positive message or not. It sends anti-messages to other LPs only if this rerun does not generate the same positive message, otherwise there is no need to send anti-messages.

The drawbacks of this approach are:

- It may allow erroneous computations to advance further.
- It requires additional time to determine whether the same messages are created or not.
- It requires additional storage to record the *positive* message(s) sent.

### 3. Lazy Reevaluation

The lazy reevaluation scheme is also known as lazy rollback and jump forward, and in some sense is similar to lazy cancellation [32]. But this scheme deals with state variables whereas lazy cancellation deals with messages. It examines if there is some change in the state of the LP between the timestamp of the straggler message and the current local clock of the LP. A rollback operation is called if some change in the state of LP has occurred during that period, otherwise the LP jumps forward directly to the new state.

Although this scheme prevents the simulator from performing unnecessary rollbacks, it demands additional storage and requires additional bookkeeping overhead, thus coding of optimistic protocols can be significantly complex.

### 4. Wolf calls

The wolf calls scheme was proposed with the intention of preventing erroneous computations from spreading widely. In this scheme, a control message is sent by the LP to all other LPs as soon as it receives the straggler. All the LPs freeze their computation immediately after receiving this control message.

A disadvantage of this scheme is that some correct computations may be frozen unnecessarily. In order to avoid this, application specific knowledge is required to know the speed of propagation of an erroneous computation and the speed at which the control message may be broadcast [32]. However, extracting this application specific information is a non-trivial task.

## 5. Optimistic time windows

In order to reduce the number of causality errors, another scheme was introduced by Sokol, Briskoe, and Wieland [34]. They proposed an approach called Moving Time Window (MTW) to use a time window with a fixed size, say W. They suggested that the number of causality errors can be reduced if LPs are allowed to process events only within the interval GVT and GVT + W.

However, critics question how the size of the window should be determined. Additionally, it is not easy to distinguish correct computations from the erroneous ones within this interval. It has been found that MTW provides better results only in some specific cases and offers little improvement in performance in general [32].

## 6. Direct cancellation

An optimized approach was presented by Fujimoto for an architecture where multiprocessors share memory with each other [35]. This scheme, called direct cancellation, keeps track of the set of events which are the causal effect of an event's processing. To illustrate this, assume that a set of events E are scheduled as the result of processing an event $E_k$. The simulator keeps a pointer from $E_k$ to E. If $LP_k$ needs to rollback and to cancel the effect of processing event $E_k$, then it can easily track the events E generated by the processing of $E_k$ and can thus cancel them.

This scheme is a good alternative to the anti-message approach, and exhibits good performance results on shared memory multiprocessors [35]. The disadvantage of this approach is that it is specific to multiprocessors with shared memory architecture, thus it cannot be applied broadly.

## 7. Filter algorithm

An improvement, called the filter algorithm, was proposed in conjunction with time-warp to reduce the cascaded rollbacks with respect to the spread of erroneous computation [36]. Unlike the wolf call scheme, the filter algorithm does not require freezing the correct computation; rather it requires each LP to keep the track of some information, such as the rollbacks carried out, total number of messages sent so far, etc.

Although this scheme reduces the number of cascaded rollbacks, it adds additional overheads to the standard time-warp approach by introducing requirements to track some additional information [36].

All of the above different flavors of synchronization algorithms belong to the optimistic approach, in that all attempt to exploit the non-zero probability of having no causal error by not strictly sticking with the local causality constraint, thus allowing LPs to process the events as they arrive. In next section we will discuss another set of synchronization protocols called conservative, which firmly enforce the local causality constraint.

### 2.1.4 Pessimistic (conservative) Approach

The first parallel simulation mechanism was based on the pessimistic or conservative approach [17]. In the late 1970s, Chandy, Misra, and Bryant independently developed the algorithms which are often referred to as *Chandy-Misra-Bryant (*CMB) protocols [32]. Since that time, a wide range of improvements, optimizations, and variations have been proposed, but all of them share a common requirement that '*cause must precede the effect*'.

The principal idea of the conservative approach is to strictly avoid causal violations. Hence, the local causal constraint is the focal point of the protocols belonging to this group of synchronization algorithms, which implies processing events strictly in a non-decreasing timestamp order. This approach is also called pessimistic, as it pessimistically considers the non-zero probability of causality violations occurring as a result of processing an event. Hence, all LPs avoid processing an event until a set of events is determined to be processed safely.

The CMB mechanism is based on the following assumptions (see slide #22 of [37]):

- The simulation model is comprised of a finite set of LPs.
- Communication between all the LPs only occurs through the exchange of time-stamped messages.
- The network topology will be static, thus LPs will not be created dynamically.
- All LPs will be linked via some channels.
- Messages will be sent in timestamp order through each channel.
- The network will preserve the global message order and will provide reliable delivery.

Intuitively, it can be concluded from above assumptions that timestamp $T_k$ of the last message $E_k$ on a specific link will be the Lower Bound on the Time Stamp (LBTS) of all subsequent messages on that link. This LBTS ensures that the LP having the current simulation time $T(LP) = T_k$ will not receive an event at any time in the future whose timestamp will be less than $T(LP) = T_k$. Hence, the LP can safely process $E_k$ without fear of any future causal violation. The events which can be processed safely are referred to as *safe events*.

It is assumed, further, for the architecture of simulator that there is a FIFO queue and a clock is associated with each link. The primary characteristics of the FIFO queue ensure that the messages will be received by the LP in exactly the same order as they were sent and that messages will be sent in a non-decreasing order of their timestamps over a link. If the associated FIFO queue is empty for a particular link, then its clock will be set to the timestamp of the last received message; otherwise, the clock will be set to the message which is at the front of the queue [32].

The basic conservative algorithm can be expressed as (based on slide #23 of [37]):

```
While simulation is not over do
    Wait until there is at-least one message in each FIFO
    Remove the event with the smallest time stamp from its FIFO
```

```
        *Process* that event
End
```

An interesting question that may be raised here is: How can an LP determine this set of so called *safe events* whose execution will not cause any causal violation? To determine this set of *safe events*, the conservative algorithms depend on the following simulation properties [4].

- **Estimated Input Time (EIT)**

  EIT is the smallest time stamp of all the events that will be received by a given LP in the future through any channel.
- **Estimated Output Time (EOT)**

  EOT is the smallest time stamp of all the events that will be sent in the future by a given LP to any other LP.
- **Lookahead**

  Lookahead is the difference between the current simulation time of a given LP and the timestamp of the earliest event that it will cause at any other LP in future. It can be determined either, for example, by calculating the delay of the associated link between two LPs in the delivery of a message, or the time taken by an LP to process an event. Suppose if an $LP_k$ having current simulation time $T(LP_k)$, requires *T* units of simulation time to process an event then it can be guaranteed that the $LP_k$ will not generate any event with a timestamp less than $T(LP_k) + T$.

Based on these properties, an LP can now safely process the events which have a smaller timestamp than its current EIT as it is guaranteed that no message will arrive later having a smaller timestamp. But if an LP does not have events with a smaller timestamp than its current EIT, then it will be blocked and must wait for having some. This means that LPs which contain no safe events or have an empty queue must be blocked from processing[2] until they get some safe messages to be processed. Enforcing these properties ensures that each LP will execute events only in non-decreasing order and consequently will adhere to the local causal constraint. However, this simple mechanism may lead to a deadlock situation if appropriate precautions have not been taken while designing the mechanism. Many approaches to keep causality constraint have been suggested by researchers; we will briefly overview some of these suggestions.

## 1. Null-message algorithm (deadlock avoidance approach)

Consider a situation where a number of LPs are cyclically dependent upon each other. Since the LPs can only process events which are safe to be executed, these cyclically dependent LPs may lead to a deadlock where all these LPs are blocked and waiting to receive a message from another LP in the cycle. This situation is illustrated in Figure 7.

[2] Because the given LP may receive a message via this empty queue which will have a smaller timestamp than all of its other input messages via other channels.

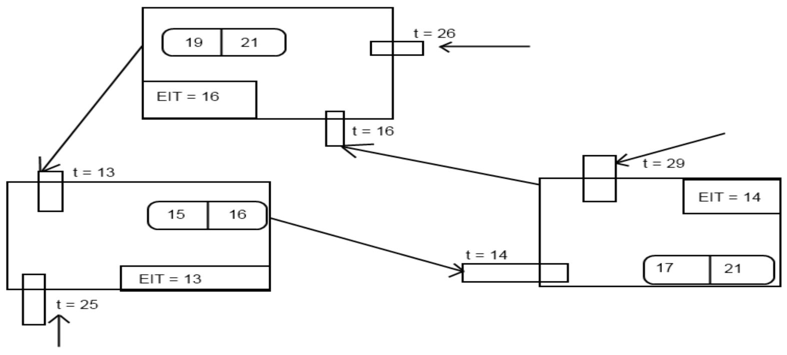

Figure 7: Deadlock Situation (Cyclic Dependency)

In the scenario illustrated in the figure 7, each LP's current EIT is less than the time of the event in its local event queue. Hence, no LP can process an event from their local event queue nor process any incoming message. As a result, each LP waits for a message from its direct neighboring LP, to increase its EIT. This situation is called a circular dependency and clearly results in a deadlock.

A mechanism called the Null-Message Algorithm (NMA) was introduced by Misra and Chandy [39]. The Null Message Algorithm is also referred to as a deadlock avoidance approach, as it prevents the simulation model from falling into a deadlock. They proposed to use null messages in order to avoid deadlocks. In this algorithm, whenever an LP finishes the execution of an event, it sends a null message with a timestamp $T_{null}$ on each of its outgoing links with a pledge that it will not send any message with timestamp smaller than $T_{null}$. These null messages are sent for only synchronization purposes and thus do not contain any simulation model related information, rather they are used to increase the EIT of all the neighbors of a specific LP. The time stamp $T_{null}$ of a null message, in fact, corresponds to the LP’s current EOT as determined by adding the value of lookahead to its current local time. Upon receiving a null message, each LP updates its EIT to a potentially greater value with respect to the $T_{null}$ and sends this information to its direct neighboring LPs; if this updated EIT has advanced beyond the events in the LP's event queue then these events can now be considered as safe events.

This explains how null messages and the lookahead approach considerably reduce the possibilities of deadlock. The extended version of the basic conservative algorithm will be (based on slide # 26 from [37]):

```
While simulation is not over do
    Wait until there is at-least one message in each FIFO
    Remove the event with the smallest time stamp from its FIFO
    Process that event
    Send null-messages to all direct neighboring LPs with Timestamp (Current Time +
    Lookahead)
End
```

### Performance considerations

The performance of synchronization algorithms based on the conservative approach is highly dependent on the size of the lookahead. Intuitively, excessively large numbers of null messages will be exchanged by the LPs without making any actual progress if the size of the lookahead is small. This unnecessary flood of null-messages is due to the lookahead creep problem. To illustrate this, assume that we have two LPs. Both of these LPs are blocked at some simulation time say T = 0125, while their next events are scheduled at simulation time T = 0250. Now suppose that the value of lookahead is only 0001. This small value of lookahead will result in the transmission of 125 null messages in order to get next safe event.

It is, nevertheless, argued that the actual size of lookahead is an inherent property of the simulation model rather than the synchronization model [4]. In a network simulation model, this size is usually determined by the link delay between nodes. This scheme works well with wired networks with large link delays, but fails with wireless networks when the link delay is relatively small. As a result, extensive research has been carried out to develop some techniques to maximize the size of lookahead for a simulation model.

## 2. Deadlock Detection and Recovery

Another alternative for dealing with deadlock was introduced by Chandy and Misra [41]. They suggest allowing simulation to enter deadlock rather than trying avoiding such a situation. In order to deal with deadlock situation, two complementary mechanisms were proposed:

1. detect a deadlock, then
2. resolve this deadlock.

A deadlock can be detected by following an approach similar to that followed in general distributed computing [32]. The fact that messages with the smallest time stamp are always safe to execute, can be helpful in breaking the deadlock; or, alternatively, the lower bound can be computed to enlarge the set of safe events for execution. More details on these mechanisms can be found in [40, 41, and 42].

The deadlock detection and recovery method has the advantage that it completely avoids the null message traffic, and further it does not prohibit cycles with zero timestamp increment; although it may give poor performance if there exist many such cycles. Critics say that a deadlock, nonetheless, may occur frequently if there are relatively few messages compared to the number of links in the network [32]. Furthermore, this scheme often results in sequential execution prior to occurrence of a deadlock. These facts can adversely affect the overall performance if the simulation model is prone to the deadlocks.

## 3. Synchronous methods and conservative time windows

Several researchers have proposed synchronous conservative methods which employ those LPs who cooperate within some interval of simulation time in order to:

1. Determine their safe events, and
2. then to process these safe events.

The focal point of these mechanisms is the notion of distance which is a subpart of the concept of lookahead. The distance can be defined as minimum amount of the simulated time taken by an event at one LP to affect the state of another LP [32]. This is the lower bound of the increment of the simulated time for an unprocessed event.

A scheme employing a moving simulated time window was introduced by Lubachevsky [55] for reducing the overhead associated with the determination of safe events. The lower edge of the window is defined, in this scheme, as the smallest timestamp of the any unprocessed event. Hence, only those events whose timestamps fall into this window are considered as safe for execution.

The problem with this Moving Simulated Time Window scheme is to determine the size of the window. If, for example, the size is too small, then there will be too few safe events; while if this size is too large then the simulation may behave the same way as it would without the time window. Application specific information, however, can be used to tackle this problem. This information can be obtained either from monitoring the simulation at run time, extracted by the compiler, or supplied by the programmer.

#### 4. Conditional events

Another scheme, to determine the safe events, called conditional events was introduced which classifies the events into two categories:

1. **Definite events:** The events which have smaller time stamps will be considered as definite events. Such events will definitely be processed and will not be disabled or canceled by other events.
2. **Conditional events:** All the other events are considered to be conditional events and will be processed when some specific condition are fulfilled. To implement this some predicates are associated with the events; when the predicate is satisfied then the event is considered as a *definite event*.

## 2.2 Comparison of Optimistic and Conservative approaches

Many analytical and investigative studies have been conducted to explore and to evaluate the performance and other features of various strategies associated with either optimistic or conservative approaches. Some of the important findings are [32]:

- There is no single approach which can satisfy all or most applications. An approach which provides good performance results for one kind of application may perform poorly for others.
- Conservative approaches are criticized because of their poor performance due to the fact:
    - They cannot fully exploit the degree of parallelism available in the simulation model.
    - They rely heavily on the lookahead value, thus their performance is sensitive to small changes in the system which can affect the lookahead values and can potentially negatively affect the overall performance.

    - They are subject to an “avalanche effect”, where the efficiency is poor for a small message population but dramatically increases with growing input size.
    - The performance is modestly affected by the amount of computation required for each event.
    - Most of the techniques require the programmer to have an ample understanding of the system.
    - They require static configuration between the LPs, so LPs cannot be created dynamically.
- Optimistic approaches inherently have greater overhead than conservative approaches. These overheads are rollbacks, periodically saving the state, fossil collection, calculation of GVT, etc.
- Optimistic approaches are criticized because:
    - They are much more complex and harder to implement.
    - They are memory intensive.
    - They must be able to recover from infinite loops and arbitrary errors.
    - Their performance can be affected by factors such as the frequency of state saving, the granularity of each LP, etc.

A very good comparison is presented in [43] by considering some operational principles of both approaches and some critiques of them. Table 2 explains the comparison of optimistic and conservative approaches.

Table 2: Comparison of optimistic and conservative approaches [43]

| Strategy / Feature | Optimistic approaches | Conservative approaches |
|---|---|---|
| **Principle** | Relax the local causality constraitnt; a rollback mechanism is provided if a causality violation occurs. | Strictly adhere to the local causality constraint. |
| **Synchronization** | If the local causal constraint is violated then some roll back mechanism is utilized to recover from this erroneous state. This adds state saving overhead. | The solutions require blocking LPs to avoid the violation of local causal constraint. However, if appropriate precautions are not taken, then deadlock may occur. |
| **Parallelism** | Allow maximum exploitation of the parallelism provided by the simulation model. | Do not allow the maximum exploitation of the parallelism provided by the simulation model. |
| **Lookahead** | No dependency on lookahead to achieve good performance. | Relies heavily on lookahead for achieving good performance and to avoid deadlocks. |
| **Deadlock** | No deadlock problem | Adopt detection and recovery strategy, avoidance strategy, or synchronous approaches. |
| **Memory Requirement** | Memory intensive | Require less memory |
| **Configuration of LPs** | Network configuration may be changed dynamically | Most existing approaches require static configuration |
| **Implementation** | Notoriously harder to implement; complex data manipulations. | Easier to implement; straightforward implementation and data structures. |

Another study [84] presents a very good comparative study of optimistic vs. conservative simulation. Nonetheless, selecting an approach is a matter of choice; however, if state-saving overhead is managed appropriately then optimistic approaches are good for general purpose simulation. Nonetheless, if one has deep and complete application specific knowledge of the system to be simulated, then the conservative approach offers greater potential performance [32].

# Chapter 3.

# Related Work

In this chapter, we will analyze a selection of contemporary simulators in terms of their PDES capabilities, architecture, programming language, and scalability.

## 3.1 PeerSim

PeerSim [14] is considered to be the most used simulator at present. A recent survey [65] concludes by saying “Among in these surveyed simulators PeerSim is best for p2p researchers…” PeerSim is a Java-based overlay network simulator for P2P, which has been designed to support scalability and dynamicity [65]. It offers two models of simulation:

i. Cycle-based and
ii. Event-based

In cycle-based mode, PeerSim sequentially executes all node protocols in one cycle, and developers can add control objects between two cycles. These control objects are used to add/remove nodes or to monitor the values of specified variables. In event-based simulations, events are defined along a time-axis and there may be zero or more events for each tick of time.

A cycle-based engine simulation is simplified as it ignores the transport layer in the protocol stack and it lacks support for concurrency [66], whereas an event-based engine simulation is more realistic as it supports dynamics and can simulate protocol stacks but decreases simulation scalability [14, 67, 68]. Some of the advantages and disadvantages [based on 66] of PeerSim are given below:

Advantages:

1. Offers very high scalability[3],
2. two modes (cycle-based and event-based) of simulation model,
3. supports some well known models, and
4. supports dynamic networks.

Disadvantages:

1. No support for distributed simulation,
2. poor documentation (only the cycle based engine is documented), and
3. no details of underlying communication protocols.

[3] Cycle-based engine offers support for $10^6$ nodes [65].

## 3.2 OMNet++

OMNet++ [26, 70] is a discrete event simulation environment, which is written in the C++ programming language. OMNeT++ itself does not provide components for simulations, instead there are simulation frameworks and models which are used with OMNet++ [65]. It has extensive GUI support, and its modular architecture makes the simulation kernel easy to embed into one's own applications. It has been successfully used in other areas such as simulation of IT systems, queuing networks, business processes, and hardware architectures [65]. It natively supports PDES by implementing the conservative NMA and the Ideal-Simulation-Protocol [26].

For distributing a simulation model to a set of LPs, OMNet++ uses a placeholder scheme [4]. In this scheme, a simple placeholder module is automatically created for each module which is assigned to a remote LP. When a message arrives at a placeholder module, it transparently marshals that message and sends it to the real module in the particular LP; that particular LP then un-marshals the message and processes it. Some advantages and disadvantages of OMNet++ [65] are given below:

Advantages:

1. Extensive GUI support,
2. Support for multitier topologies, and
3. Reusability of simulation models.

Disadvantage: Low scalability[4]

## 3.3 OverSim

OverSim [69] is an open source overlay and P2P network simulation framework for the OMNeT++ [26, 70] simulation environment. The programming language of OverSim is C++. This P2P simulator contains several models for structured (such as Chord [71], Kademlia [51] etc.) and unstructured (such as GIA) P2P protocols. It is a widely used P2P simulator and is reported [64] to simulate up to 100,000 nodes in an event-driven fashion. It shares many characteristics with OMNet++ such as GUI interface for validation and debugging of new or existing overlay protocols [65].

## 3.4 NS-2

NS-2 [73, 74] is a discrete event simulator which provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless networks [65]. NS-2 performs simulation using a mixture of C++ and OTCL [76] (an object oriented version of TCL [75]). NS-2 uses Network animator [78] (Nam) to provide visualization. Nam also allows the users of NS-2 to arrange network graphs to aid in debugging and designing of network protocols. Because of its realistic nature, the major drawback of NS-2 is scalability [66] and the simulation of larger networks is very difficult (if not impossible) [77].

[4] OMNet++ can be scaled up to 1000 nodes [65]

### 3.4.1 PDNS (Parallel/Distributed NS)

PDNS [72, 77] constitutes the parallel simulation architecture of NS-2 [4]. It coordinates distributed instances of NS-2 (which executed the partitions of the parallel simulation model) by employing a conservative synchronization algorithm [4]. These instances (or LPs) are called federates in PDNS terminology, and the links between nodes in different federates are replaced by dedicated 'remote links ', which implement PDES functionality. PDNS is built upon two communication libraries (LibSynk and RTIKIT) for providing actual inter-federate communication.

## 3.5 PlanetSim

PlanetSim [79] is an object oriented simulation framework for P2P overlay networks and supports structured and unstructured P2P simulation [66]. It is written in Java and is based on the query-cycle approach which makes its parallelization rather straightforward [64] as each process has to be processed separately, at each simulation step. It is, however, criticized for not providing any mechanism for statistics gathering [65], and for limited simulation of the underlying network [66].

## 3.6 DSIM

DSIM [80, 81] is a PDES which is based on an optimistic synchronization approach, and designed to run on a large scale simulation cluster comprised of hundreds of thousands of independent CPUs [4]. DSIM employs a variant of time warp synchronization algorithm. Resource management is performed by a local garbage collection algorithm, which increases locality by immediately reusing freed memory.

A comprehensive survey and comparison of a wide range of contemporary P2P simulators is given in the 8th chapter of [4], and in [65-68]. Interestingly, none of the P2P simulator reviewed in the cited surveys [65-68] seems to support parallel simulation to exploit multi-core architectures. One reason for this fact is explained in [64] as:

> "*In our opinion, the relative failure of PDES in this context comes from the fact that the simulation of distributed systems (such as P2P protocols) is different from the simulations classically parallelized in the PDES literature.*"

## 3.7 Summary

Table 3 presents a detailed summary of all the above explained P2P simulators.

Table 3: Detailed summary of different P2P simulators (based on [65] and [66])

| Simulator / Feature | PeerSim | OMNet++ | OverSim | NS-2 | PlanetSim | DSIM |
|---|---|---|---|---|---|---|
| **Architecture** | Query-Cycle and Discrete event | a modular simulation framework, Discrete event | a modular simulation framework, Discrete-event | discrete event | an object oriented simulation framework | discrete event |
| **Programming Language** | Java | C++ | C++ | C++ | Java | C++ |
| **Scalability (Max. nodes)** | 10^ 6 nodes[5] | 1000 | 100,000 | NA | 100,000 | NA |
| **Statistics Gathering** | Components can be implemented for gathering statistical data | NA | A Global Observer module can be used as a statistics collector | NA | no mechanism for collecting statistics | NA |
| **URL** | http://peersim.sourceforge.net/ | http://www.omnetpp.org/ | http://www.oversim.org/ | http://nsnam.isi.edu/nsnam/index.php/Main_Page | http://projects-deim.urv.cat/trac/planetsim/ | http://assassin.cs.rpi.edu/~cheng3/dsim/ |

[5] Only by using cycle based engine.

# Chapter 4.

# Design and Architecture

In this chapter, we will describe the overall architecture of our proposed PDES, “HydraNetSim”. We will first identify the objectives and the challenges to achieve our ultimate goal of parallelization of discrete event simulator, and then we will describe our proposed architecture.

## 4.1 Objectives

We have formulated a couple of steps (primary and secondary objectives) by identifying main challenges to achieve the desired parallelization. The primary objectives are essential components required to develop a fully functional simulator; whereas the secondary objectives are improved efficiency and improved performance as compared to the current SDES.

### 4.1.1 Primary Objectives

The primary objectives can also be seen as milestones. These objectives are stated and classified in the following paragraphs.

#### 1. Partitioning

The first challenge in developing a parallel simulator is to choose a specific type of scheme for exploiting parallelism. This scheme must ensure the division of simulation model into multiple, but finite number of parts in such a way that all these parts can be executed in parallel on independent processing units on different cores. To achieve this partitioning we need to consider the issues described in the following paragraphs.

##### a) Interoperability

Once the simulation model is decomposed into a finite number of sub-modules called partitions, the next challenge is to properly manage and organize these partitions to ensure cohesion. Since each of these partitions represents a different and distinct part of the simulation model, therefore they must work cohesively in coordination with each other to ensure interoperability. This challenge of managing the partitions can be further broken-down into the following tasks:

- Coordination of partitions,
- Collecting and aggregating the simulation output from them, and
- Writing this output to the database

##### b) Experiment definition

Another important challenge is how to distribute the experiment set (or simulation scenario) over these LPs? How should LPs read and write to the database?

c) **Assignment of nodes**

The simulation is run by simulated nodes on each LP. The configuration and definitions of these nodes will be loaded from a database, and then the nodes will be assigned to each LP. A crucial challenge is how to assign simulated nodes to each LP? Further, how should we balance the subsequent load on the processors as we assign the nodes to different LPs?

#### 2. Support User behavior

It is necessary to support the user behavior[6] to obtain realistic results when running a simulation of a (P2P) system. HydraNetSim should be capable of supporting node churn, workload, and use properties.

#### 3. Determinism

The random number generator (RNG) must generate the same sequence of numbers for each simulated node; no matter how many processes are used for simulation.

#### 4. Synchronization among all the LPs

The core and real challenge when building a functional PDES is to keep all the LPs fully synchronized. Synchronization among all the LPs is critical for getting correct results from the simulation. The simulator may produce erroneous results if the LPs are not synchronized with each other, and thus the simulation will fail to exhibit the desired behavior.

#### 5. Message Routing

Equally important challenge is correctly and reliably routing of each message to its intended recipient(s), regardless of whether the receiving node is running on a local LP or on a remote LP with respect to the sender.

#### 6. Gathering and writing of simulation results

Another fundamental objective is to gather the output results from each LP and then to store these results into a database.

### 4.1.2 Secondary Objectives

A couple of secondary objectives have been proposed for refining the architecture of HydraNetSim. These additional objectives are:

- Balancing the subsequent load between LPs while assigning the nodes,
- Adopting an optimal synchronization scheme which yields synchronous LPs, and
- Minimizing the synchronization overhead.

In the next section, we will explain the architecture of our simulator.

## 4.2 Architecture

We have discussed different optimistic as well as conservative approaches in chapter 2. While choosing an approach is purely a matter of choice, there are some characteristics of

[6] Explained in chapter 2.

these two alternatives which can lead to a particular approach being more attractive for one to adopt.

The implementation of a conservative approach is easier and more straightforward in comparison with the implementation of an optimistic approach [43]. Furthermore, a conservative approach offers greater potential performance, especially when developer is fully aware of application specific knowledge about the system to be simulated [32].

We have decided to adopt a conservative approach because:

- It promises to completely avoid the possibility of getting erroneous results, rather than producing incorrect results and then recovering using a roll back from an erroneous state.
- Since we have a deep understanding of the system that we wish to simulate, we can exploit the potential promises of a conservative approach.
- Implementation of a conservative approach is relatively easy in comparison to implementation of an optimistic approach.
- Optimistic approaches intrinsically have greater overheads (such as GVT, rollbacks, periodically saving states, etc.) in comparison with conservative approaches.

In the following subsections, we will explain the architecture of our simulator based on a conservative approach.

### 4.2.1 Partitioning

A PDES can be depicted as a collection of a finite number of sequential discrete event simulators each executing on different processers and which can communicate with others only through time-stamped messages called events. Therefore, the very first challenge in developing a PDES is to divide the simulation model into a finite numbers of partitions in such a way that each of them can be executed on a separate processing unit in parallel with the other partitions.

We have explained, in section 2.1.1, space parallel partitioning divides the system into a finite number of partitions called LPs, where each LP is composed of a cluster of nodes which in turn is mapped to a different processer. This paradigm offers great potential for both execution speed and network size scaling up in proportion to the number of processors. Thus, we will adopt the space parallel decomposition paradigm for decomposing our network model. The implementation details of this will be explained in the paragraphs below.

A subsequent challenge which arouse as the result of partitioning is how to coordinate and manage the LPs. This management and coordination will actually tackle the challenges of how to assign an experiment to each LP, how these LPs will be set up for an experiment, how these LPs will read/write to the database, and how results will be obtained from each LP.

#### Master/Slave paradigm

We have devised a master/slave approach to tackle all these challenges. In this simple scheme, we have formulated two types of processes:

1. **Master:** The master will coordinate all the LPs and will send instructions to each of the LPs for loading the experiment and it will write the simulation results to the database as they are received from the LPs.
2. **Slave:** The simulation model will be divided into a finite number of LPs which will carry out the simulation. These LPs will be called slaves[7] and will be managed by a master.

The master does not run any part of the simulation; its only job is to instruct the slaves how to set up a simulation run, to initiate the simulation run, and to write the simulation results to a database after receiving results from the slaves. The slaves and the master will each have their own local copies of the database. In spite of having their own copies of database, the slaves will not write anything to their own copy of the database, but rather they will send all their simulation results directly to the master. On the other hand, the master will receive these simulation results and will write these results to the database. The master will adopt some scheme to avoid redundancy and to preserve correctness; this scheme will be explained in section 4.2.7. Slaves will not set up the experiment on their own, instead the master will send them instructions about the name of an experiment that the slave is to load from the database, and then the slave will start processing its own part of the simulation.

The master will be started by giving it appropriate arguments, such as:

- Database name,
- Experiment name for simulation run, and
- Total number of slaves.

The format of the command to start the master will be:

*-mode master -database <dbName> -experiment <expName> -slaves <numberOfLPs>*

As a result of this command, the master will be started and will remain in '*waiting mode'* to receive the connection requests from the slaves. By using the term '***waiting mode***', we mean that a process is although active, but is waiting for some other process to perform some activity so that it can continue its processing.

Each slave that is started will be given the network address of the master. Each slave will send the master its IP address (and port number) in a connection request in order to establishing a channel to the master. The master and slaves will communicate with each other using a channel-based communication protocol CUSP [46]. CUSP is briefly explained in subsequent paragraphs.

A typical command for starting a slave will be:

*-connectTo <Network Address of master> < port number>*

As the result of this command, the slave will be connected to the master and will wait to receive instructions to set up the simulation run from the master.

[7] 'LP' and 'slave' have same meanings and concept in our model. Therefore, we will use slave and LP alternatively for our convenience, in rest of the report.

The master will receive connection requests and populate an ‘LP information table’ (LIT) containing the network addresses and port number of each of the slaves. The master will assign a unique ‘id’ to each LP in this table. This LIT will help a slave to establish a connection with its peer slaves. The format of LIT is shown in Table 4.

Table 4: LIT

| Id | Network Address of LP | entryPoint |
|---|---|---|
| **X** | x.x.x.x:x | X |
| ⋮ | ⋮ | ⋮ |
| **N** | **z.z.z.z:z** | **z** |

Once the master receives a connection requests from all of the slaves, it will reject any further connection requests, and will send a copy of the LIT to each and every slave along with its simulation set up instructions. The format of the LP configuration message (LCM) sent by the master is shown in figure 8.

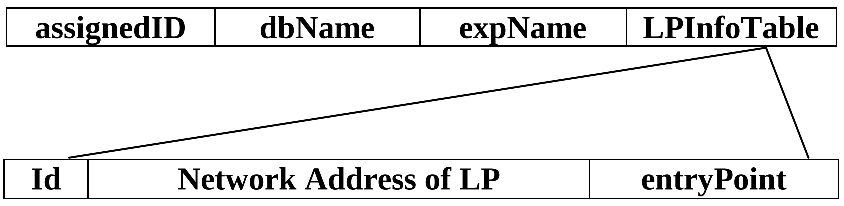

Figure 8: LCM

The details of LCM are as follows:

Table 5: LCM Description

| | |
|---|---|
| **assignedID** | The assignedID is the ‘id’ of a given slave as assigned by the master in the LIT. |
| **dbName** | The dbName is the name of the database. |
| **expName** | The expName is the name of the experiment which will be loaded by the slave, for simulation**.** |
| **LPInfoTable** | The LPInfoTable will act as an ‘LP lookup table’, and will contain the necessary information about all the slaves participating in the simulation run. |

Although each slave will now have all the necessary information for starting the simulation it will not start the simulation yet, as it first needs to establish a channel with:

i. peer slaves for sending/receiving messages, and with
ii. the master for sending log entries.

The communication between master and peer slaves is depicted in Figure 9.

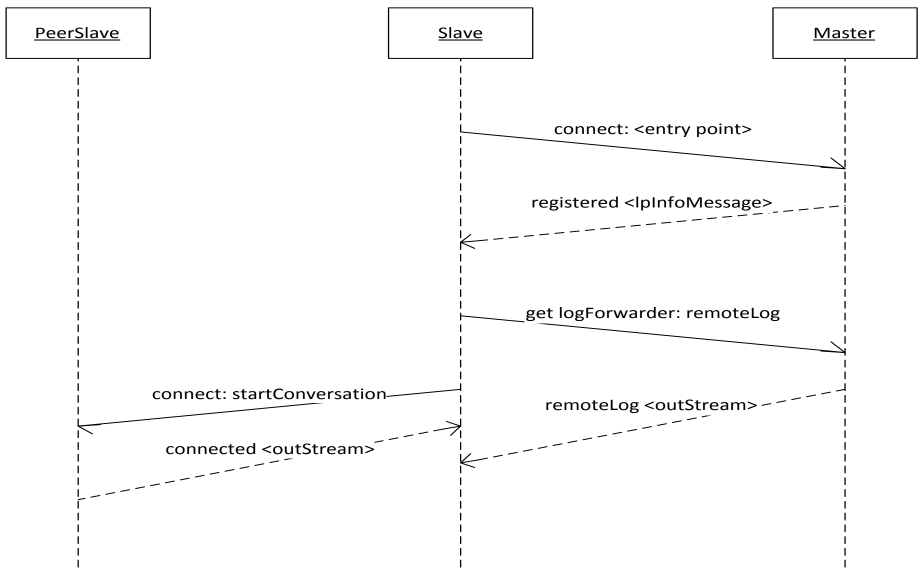

Figure 9: Master/Slave Communication

The network topology, as a result of using this paradigm, will be a fully connected mesh topology where each slave will be connected with every other slave and with the master as well as shown in figure 10.

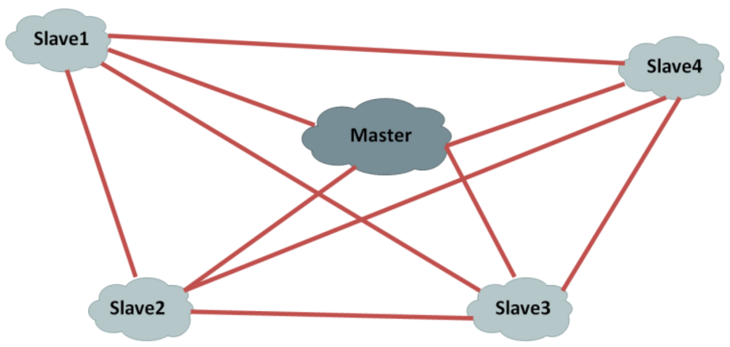

Figure 10: fully connected MESH topology

## Communication Protocol

We will use a channel-based, TCP like protocol called CUSP [46], as communication protocol for exchanging messages between different components of our model over the network. CUSP is a transport protocol implemented on top of UDP. CUSP is specifically designed for complex and dynamic network applications. It has been built on the ideas of SST [47] and SCTP [48]; it divides the transportation connection into channels. These different channels are then responsible for low-level packet management and streams. Streams in CUSP are unidirectional, because not all messages expect an immediate or direct response; bidirectional streams are based on two streams. In spite of its feature-richness, CUSP can be implemented in few lines of code because of its simplicity. Please see [46] or [49] for further details.

## Partitioning scheme

Once the slave has established conversation channels to communicate with the master as well as with its peer slaves, the next step is to load the named experiment from the database as per the instructions received from the master and to start the simulation. Each slave will create simulated nodes based upon the node definitions from the database.

A crucial challenge here is to balance the load when assigning the nodes. We will follow a simple scheme to ensure this load balance. We compute mod of '*node id*' from '*total numbers of LPs*'; if the remainder is equal to the id of this LP, then we create a local version of the 'simulatorNode'. Otherwise, if the remainder is not equal to this LP, then we create a remote version of this 'simulatorNode' by setting the value of the remoteNode's '*runningOnLP*' to the remainder. The node, in this way, is allocated based on:

$$\frac{\text{nodeID}}{\text{total numbers of LPs}}$$

If there are 3 LPs (having ids 0, 1, 2 respectively) participating in a simulation, then the slave with id 0 will read node definitions from the database and will instantiate the nodes shown in Table 6.

Table 6: Node Assignment Scheme

| <Node id> mod <total LPs> | Remainder | Running on LP |
|---|---|---|
| **0 mod 3** | 0 | 0 |
| **1 mod 3** | 1 | 1 |
| **2 mod 3** | 2 | 2 |
| **3 mod 3** | 0 | 0 |
| ⋮ | ⋮ | ⋮ |
| **N mod 3** | X | X |

This scheme will be followed by each LP giving a uniform and identical way of node creation. Additionally, this scheme provides additional, but handy information about which node is running on which LP. Note that since each node follows the same algorithm, it also knows which node every LP is assigned to.

### Simulator Node Structure

This partitioning scheme yields two kinds of nodes: local and remote nodes.

1. **Local** nodes are assigned to a given LP and will be local for that LP.
2. **Remote** nodes have been assigned to other LPs, hence they will be remote relative to that LP.

Each LP, consequently, will have to create two versions of simulator nodes:

i. Local version: for local nodes
ii. Remote version: for remote nodes

The local version of the simulator node structure will provide the full structure of a simulator node and contains all the information associated with this simulator node. Table 7 shows some of the most important information provided by a local version of a simulator node's structure.

Table 7: Important properties of the local version of a simulator node

| Property | Description |
|---|---|
| **id** | Node's identifier. |
| **current address** | Current IP address of a node, if available. |
| **location** | Geographical location of a node. |
| **random** | Random number generator (RNG) of a node. |
| **isActive** | If node is currently active. |
| **isOnline** | If node is currently online. |
| **isRunning** | If node is both active and online, only then it is actually running the application and connected to the database. |

The remote version of a simulator node's structure will only be used to maintain an up-to-date mapping of the remote node's 'node ID' and 'IP address'. This remote simulator node's structure acts as a stub for nodes running on remote LPs. The remote simulator node's structure is shown in Table 8.

Table 8: Important properties of remote version of a simulator node

| Property | Description |
|---|---|
| **id** | Node's identifier. |
| **current address** | Current IP address of a node, if available. |
| **location** | Geographical location of a node. |
| **running on LP** | LP's identifier which is responsible to create and run the given node. |

We need these two versions of simulator nodes because of the turbulent behavior of nodes when simulating the churn in a P2P network. This remote version of a simulator node provides all the necessary information about a remote node, thus it is used to route a message to a node running on a remote LP.

### 4.2.2 Support User behavior

User behavior must be considered when designing a P2P system in order to obtain realistic results [4]. Therefore, a simulator must also be capable of supporting different aspects of user behavior. We have introduced a new component named the '*churn manager*' to provide this capability.

**Churn manager** Each LP will have a special component called churn manager to simulate churn, workload, and use properties of each node that is assigned to that particular LP. The churn manager will read the nodes' properties from the DB and then simulate churn, workload, and use properties of each node accordingly. The churn manager will be responsible for the joining of simulated nodes, going on-line/off-line (scheduled leaves), or terminating (i.e., a crash) of nodes, assigning/removing the addresses of nodes into/from the address table, and assigning the workload to simulated nodes.

The churn manager reads the information specific to a particular simulated node such as the random number generator (RNG), session/intersession duration, the ratio of on-line time vs. total (on-line + off-line time), workload, etc. The churn manager assigns an IP address to a node when that node will be created and joins the system; it removes these addresses from the address table when the nodes go off-line or crash.

### 4.2.3 Determinism

Determinism is a very important feature for any simulator, because otherwise simulation results will not be reproducible. In order to achieve determinism, we utilize a strategy based upon two flavors of RNGs:

1. **Global**
   One central RNG will be used for everything that all LPs simulate synchronously, such as churn.
2. **Local**
   Each node will have its own individual RNG that will be used for all simulation operations of that particular node. Importantly, the Global RNG will be used to seed the local RNGs of the nodes.

By using a common scheme of RNGs we can ensure that when simulating churn of nodes all nodes run identically on each LP. If we initialize the global RNG to a given value, then we can re-run the simulation and expect to get the same results as for an earlier run with this same initial global RNG value.

### 4.2.4 Major components of a slave

After describing the underlying principles of our design, in this section we will explain the salient components of a slave. Figure 11 depicts the general skeleton of a slave:

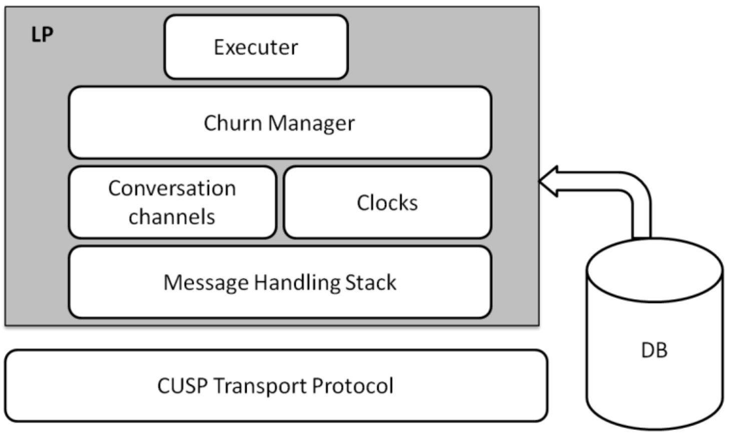

Figure 11: Salient components of a Slave

Each LP will have its own copy of the DB; it will load the experiment from DB as per the instructions sent by the master. Furthermore, each LP communicates with all other LPs using CUSP as a communication protocol. We have explained the details about both (the DB and the CUSP) in previous sections. Now we will explain the details about components specific to a slave.

- **Message handling stack**
  The message handling stack is a container of components which directly deal with messages. This message handling stack consists of:
  - **In-queue**
    The '*Message-Router*' adds any incoming event for a slave to its in-queue. These received events are placed into the event-queue for execution.
  - **Event-queue**
    Each slave maintains a central queue for execution of events. This event queue holds all the events (either *local* or *remote*) sorted by their timestamps for execution. The '*Executer'* removes an event from the front of the event queue and processes that event.
  - **Out-queue**
    Any event which is meant to be sent is added to this queue. The '*Executer'* will remove an event from the front of out-queue and forwards it to the *'Message-Router'* which will route the message to its designated receiver.
  - **Message-Router**
    The '*Message-Router*' takes the message from the out-queue of a sending node and if the message is designated for a node on same LP then it puts that message into the in-queue of receiving node, otherwise it consults the '*address table*' and routes the message to the LP which hosts the destination node. Furthermore, the message-router determines the delay or loss based upon a network model that is loaded from DB as part of the simulation. This delay is added to the local clock value to determine the timestamp which this message should have upon arrival at the destination.
  - **Address Table**
    Address table serves as a look up table for the *'Message-Router'*. This table associates each simulator node with an IP address (and UDP port number). It also stores information about which LP is responsible for a particular node. This information can be used to determine whether the node is running on a local LP or on some specific remote LP. Figure 12 illustrates the message handling stack.

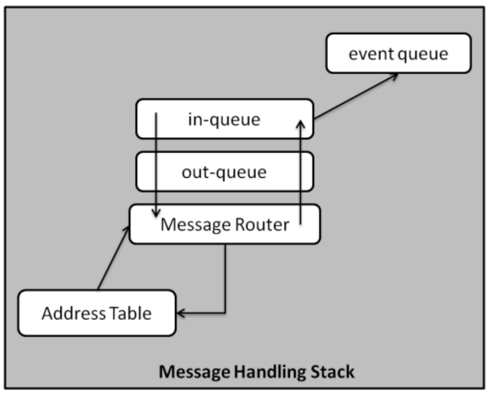

Figure 12: message handling stack

- **Conversation channels**

  Each LP has two types of conversation channels:

  i. With peer slaves, and

  ii. With the master.

  - **Conversation channels with peer slaves**

    As explained in section 4.2.1, each LP establishes a connection with every other LP. Figure 13 depicts the conversation channel between two slaves. The conversation channel is comprised of two streams:

    - **incoming stream**

      Incoming stream will receive messages from peer slaves.

    - **outgoing stream**

      Messages will be sent to the peer slaves via the outgoing stream.

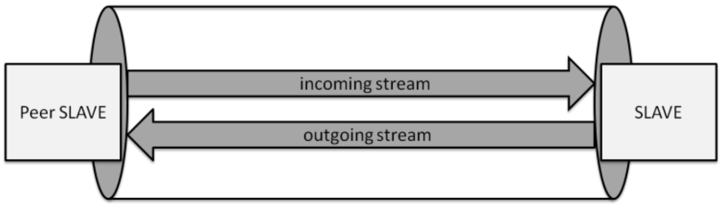

Figure 13: Conversation channel between LPs

  - **Conversation channel with master**

    Each slave will establish, as stated in section 4.2.1, a conversation channel with the master. This channel is unidirectional in that a slave will not receive acknowledgements or any other message from the master rather it will simply send (Log) messages to the master. The master, on the other hand, will receive these (Log) messages and will perform the required operations on them, for example either print them out or writing them into the database.

- **Clocks**

  Each LP has two types of clocks:

  - **Local clock**

    Each slave has a '*local clock*' for keeping the track of current simulation time.

  - **Remote clock**

    A remote clock is associated with each conversation channel established with every other peer slave. This clock will be used to keep the track of LBTS from the specific peer slave associated with a given channel.

- **Churn manager**

  The churn manager is responsible for bringing the simulator nodes online and taking them offline. It assigns an address whenever a simulator node comes online or joins the system and registers that address in the address table. Similarly it removes the address from the address table when that node goes offline or leaves[8] the system. Global RNG is used to assign addresses to a simulator node, in order to ensure

[8]Either it is planned or unplanned.

identical and synchronized information about nodes' addresses. The details about the functionality of the churn manager have been explained in section 4.2.2.

- **Executer**

  Executer executes events and updates all the other components.

The components of a slave can be, thus, categorized into the following two categories:

**localLP** The localLP contains the components which are specific to this particular LP, such as executer, message handling stack, local clock, etc.

**remoteLP** The remoteLP contains the components which are meant to communicate with peer LPs, such as outgoingStream, remote clock associated with a specific (remote) LP, etc.

To realize all of this state information the local LP contains an array of *RemoteLPs*. Each element contains all the required components to communicate with and to keep track of the LBTS from this specific (remote) LP. The size of this *RemoteLP* array will be equal to the maximum number of peer (remote) slaves. Figure 14 shows the specialized components of a slave.

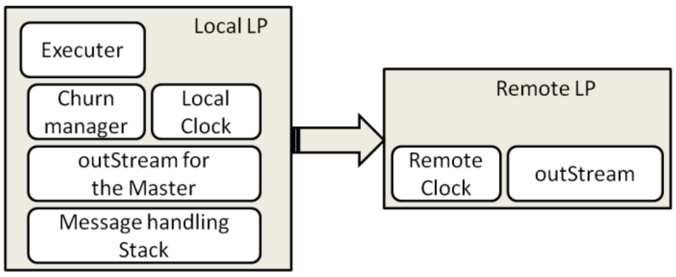

Figure 14: LP components

### 4.2.5 Message Routing

The previous sections have described the skeleton for '*HydraNetSim*'. This section describes how the simulated nodes exchange time-stamped messages via message routers. There are two cases to consider:

i. How to forward a message when both sender and receiver reside on the same LP?
ii. How to forward a message when the receiver resides on a remote LP with respect to the sender?

Whenever a simulator node needs to send a message to another simulated node (regardless of whether it is local or remote), it adds the message into its '*out-queue*'. If network congestion is to be simulated, then a delay is added and the message is forwarded to the '*Message Router*'. The '*Message Router*' consults the '*Address Table*' to determine the network address of receiver and the responsible LP which is hosting that receiver, for this message. The '*Message Router*' determines the backbone delay or network delay and adds this to the message's timestamp.

#### i. Routing a message to a local receiver

If the receiver is also running on the local LP, then the '*Message Router*' connects the bottom of sender's message handling stack to the bottom of receiver's message handling stack. The message is then added to the in-queue of the receiving node. This process is shown in Figure 15.

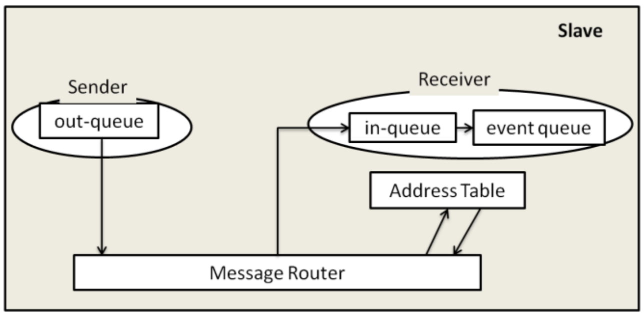

Figure 15: Routing the message to a local receiver

## ii. Routing a message to a remote receiver

If a message is destined to a receiver which is running on a remote LP, then the 'Message-Router' determines this responsible (remote) LP. The 'Message-Router' identifies the outStream associated with that LP, based upon this (remote) LP's entry in 'RemoteLP' (as explained in section 4.2.4). As it is already explained that each LP is connected with every other LP through a conversation channel, the message will be transferred through this channel over the network to the target LP (as shown in Figure 16).

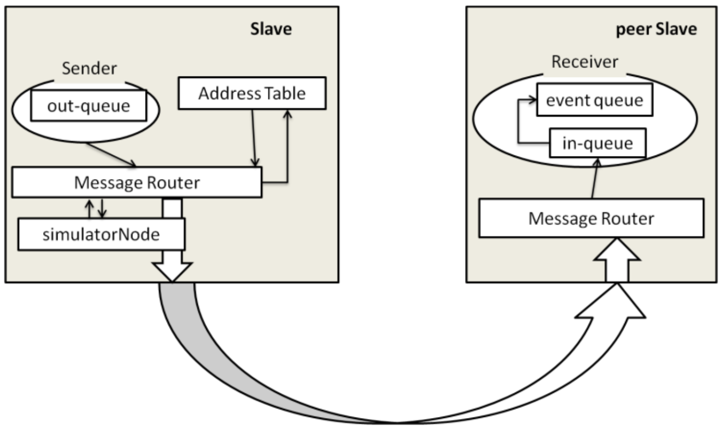

Figure 16: Routing the message to a remote receiver

The receiving LP will forward this message to its *'Message Router'* which in turn forwards it to the '*in-queue*' of receiving node. The receiving node will subsequently take the message out of the in-queue an place it into the '*event-queue*'; from where *'Executer'* will extract it and process it.

The *'Message Router'* learns the current network address of a destination node from the *'Address Table'*. As noted earlier we assume that churn for all simulated nodes is implemented globally (and synchronously) on each LP, hence the *'Address Table'* will always contain an up-to-date address of a node irrespective of that this node is local or remote. The stub version of simulatorNode, as explained in section 4.2.1, will inform the '*Message Router*' which peer LP is responsible for as the destination node of a given message. This information ('current address' and 'target LP') is critical for *'Message Router'* to successfully route the message to its destined receiver.

#### iii. Handling a received message

Whenever a slave receives an application message, it will extract it from the in-queue and store it in its event execution queue by following a specific scheme. The following properties are useful to ensure a total order of events and to address the problem of concurrently arriving events, thus enabling all of the parallel executions to preserve the '*Local Causality Constraint*'.

- **Departure time**

  Departure time is the timestamp when the message actually leaves the out-queue of the sender. If network congestion is to be simulated, then a delay is added into the sending timestamp of message. The '*departure time*' is therefore:

  Departure time = message sending time + out-queue delay

- **Arrival time**

  Arrival time is the timestamp when the message is received by the in-queue of receiver. The '*Message Router*' adds the appropriate network delay to the timestamp of message before sending it through the physical network. Therefore, a message received in the in-queue of the receiver has a timestamp of the form:

  Arrival time = Departure time + network delay

- **Sender's id**

  The sender's id is the identifier of the simulator node which originated the message.

Whenever a message arrives at the in-queue of receiver, it is stored in the event queue according to its arrival time. If two events having the same '*arrival time*' arrive at the receiver, then we sort and store them by comparing their '*departure time*'; if this '*departure time*' is also the same then we compare their originator's ids and use this as the basis of sorting them. This scheme ensures consistent handling of the concurrently arriving events as all the received events are '*totally ordered'* (as defined by Leslie Lamport in [45]) by following this scheme.

Furthermore, repeating the simulation will give a consistent output. However, we should note that this result is not the same results as would occur if messages arriving at the same time and with the same departure times were randomly ordered when placing them in the event queue.

### 4.2.6 Synchronization of slaves

All the slaves will have loaded the experiment from the database and have started their simulation after creating simulator nodes; each slave can, also, now route messages to its peer slaves through established channels. Now, we need to enforce some synchronization scheme to prevent the slaves from producing erroneous simulation results.

We have already explained that we will follow a conservative approach to avoid any causal violation. Before describing our proposed synchronization algorithm; it is helpful to illustrate a few concepts according to our model:

- **Estimated Input Time (EIT)**

  EIT is the smallest timestamp of all the events that will be received by a given LP in the future through any channel. For a slave, in our model, *'EIT'* will be the minimum value of all '*remoteClocks*' from the array '*RemoteLP*', where each '*remoteClock*' is associated with a conversation channel to a specific remote LP, and represents the timestamp of last received event through that specific link.
- **Estimated Output Time (EOT)**

  EOT is the smallest timestamp of all the events that will be sent in the future by a given LP to any other LP. '*EOT'* can rightly be replaced with LBTS; as, in our model, all the messages from a given LP to any other peer LP will be sent in a non-decreasing timestamp order, so each message will contain the lower bound on timestamps of all future messages from that LP.
- **Lookahead**

  Lookahead (LH) is the difference between the current simulation time of a given LP and the timestamp of the earliest event that it will cause at any other LP in the future. It is, in fact, the ability to look into future that yields the earliest time in the future when a given slave may send a message to any other peer slave. This prediction helps to determine the EOT. EOT can be computed as:

  $$\text{EOT} = \text{current time} + \text{LH}$$

Here the '*current time*' represents neither the EIT nor the current simulation time. We will explain this in detail when we explain the details of our algorithm.

## Assumptions for the Model

As explained in section 4.2, we will follow a conservative approach to avoid any causal violation. We have devised a variant of NMA to ensure sticking with causality constraint. Therefore, all LPs are blocked from processing an event until a set of events is determined to be processed safely.

The implementation of any algorithm belonging to the conservative family of synchronization approaches demands a deep understanding of system, and ample knowledge of the topology *a priori*. We will explain the salient features of our model here, in spite of the fact that some aspects have already been explained earlier in this chapter. Our model is based on following assumptions:

- The simulation system will be consisting of 'N' LPs, where each LP will be connected with every other LP and thus can communicate directly to it. This means that we use a fully connected mesh topology as shown in Figure 17.

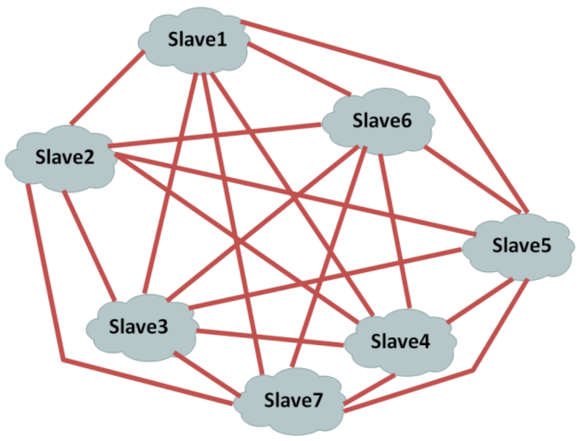

Figure 17: Simulation Model topology: Fully connected mesh topology

- Each LP can send two kinds of messages to any other peer LP:
  **Null** message carries only the '*current EOT'* of the sending LP.
  **Application** message contains application specific data and the '*current EOT'* of the sending LP.
  - There will be further two sorts of '*Application messages*' in our model:
    **UDP messages** These messages are meant to exchange application specific data between nodes.
    **ICMP messages** the purpose of these messages is to notify nodes about errors, such as receiver is unreachable (when a receiver goes offline while the message is in flight), receiver has not bound to the given port, etc.
  - These '*Application messages*' can be further classified into:
    **Local** The designated receiver of given message resides on the same LP as the message sender.
    **Remote** The designated receiver of the given message resides on a different (peer) LP of the sender.

  All of these message types and their relationships are shown in Figure 18.

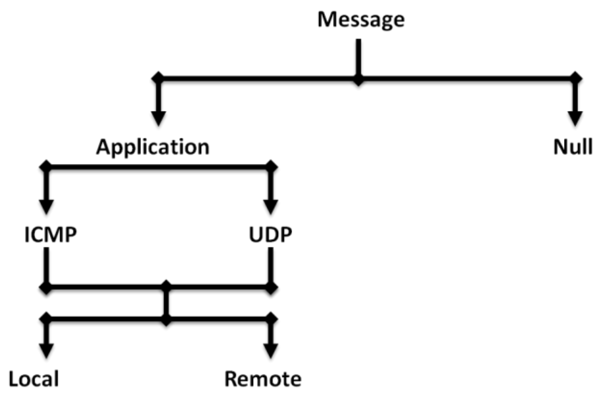

Figure 18: Message Types

- Each LP will have N-1 '*clocks*' called '*remoteClocks*'; where each '*remoteClock*' is associated with a communication channel established with a peer LP.
- The only way to communicate between LPs will be through time-stamped messages.
- LPs will send messages to each other only through streams over already established conversation channels.

- Each LP has a *'Local Clock'*, which reflects its current simulation time.
- There will be a central queue for each LP called the '*event queue* or *execution queue*' in which all the local as well as remote events will be stored in a '*total order*' following the approach used for the '*Lamport Clock*' [45].
- Messages will be sent strictly in a non-decreasing timestamp order. The network will preserve this order and will provide reliable delivery.
- Each message in the queue will eventually be processed (the liveliness property as described in section 2.1.2.3).
- All LPs will be started with same simulation time as their starting time.
- Churn for all nodes will be simulated identically on each LP. Hence, a remote version of the '*simulatorNode*' and '*address table*' will have up-to-date information about all nodes.

## Synchronization algorithm

We have designed our synchronization algorithm based on the objective:

"Wait; until it is safe to process"

Each LP remains in blocked mode (by default), i.e., it will not simulate any event, until it has one or more events which are safe for execution. To determine this set of safe events, we have formulated a set of properties for our model as explained at the start of section 4.2.6. The events in the event-queue of a given LP which have a timestamp less than its current '*EIT'* are safe to execute, because this prerequisite assures that there will not be any event in the future triggered by any remote LP that can have a timestamp less than the current EIT of the given LP.

As it has been already described that each LP will have a special component called *'Executer'* to execute the events, this *'Executer'* waits for the current '*EIT'* to be greater than the timestamp of the event which is currently at the front of the event-queue. Whenever *'EIT'* is greater, then the executer will extract the event out of event-queue and process that event. It will continue this processing until it finds no event having a timestamp less than '*EIT'*. When an LP has finished execution of this set of safe events, it will stop its simulation and compute its current '*EIT'*. If this new '*EIT'* is greater than the timestamp of events in the event-queue, then the LP will start simulating again, otherwise it will remain in *'block* mode' until it determines a new set of safe events.

To handle deadlocks we follow the deadlock avoidance scheme with some variations to the conventional CMB Null-Message-Algorithm [39]. In true spirit of CMB NMA, whenever an LP finishes the execution of an event, it sends a null message with its current simulation time as the timestamp $T_{null}$ on each of its outgoing links as a pledge that it will not send any message with a timestamp smaller than $T_{null}$. This null message contains the current *'EOT'* of the sending LP, and thus it is a LBTS of subsequent messages from that LP. Upon receiving a null message, each LP updates it's *'EIT'* to a potentially greater value with respect to that contained in $T_{null}$ and sends this information to its direct neighboring LPs. If the updated EIT has advanced beyond the timestamp of events (at least the first event) in its event-queue, then these events can be considered as safe events, and therefore can be processed.

### Lazy Push mechanism

Although this strategy is good for keeping the LPs in synchronization with each other and for minimizing the probability of a deadlock, it comes with a high synchronization overhead as many null messages must travel through the network after execution of each event. For this reason, we have changed the scheme to minimize the synchronization overhead. We call our strategy "Lazy Push" as it '*sends a null message only when it is necessary*'. During simulation a LP will either be in '*block mode*' or in '*simulation mode*', therefore we have divided our strategy into two parts, one to deal with each state. The details of this will be described in below.

#### i. Simulation mode

- A null message will not be sent eagerly, rather a null message will only be sent before starting the execution of safe events; specifically null messages will be sent only once before determining the set of safe events for execution.
- No null message will be sent during the execution of safe events.
- Each LP will send its current '*EOT*' along with every application message to other LPs.

This variant of classic NMA considerably decreases the frequency of null messages in the network, especially when there will be more safe events to execute.

#### ii. Blocked mode

This decrease in null messages is directly proportional to the magnitude of safe events. Therefore this scheme will work well until there are *'some'* events to execute safely, otherwise it will continuously sending null messages when there are no more safe events for execution. This will produce a large number of null messages when the LPs are blocked. To address this overhead problem (which only occurs in '*blocked mode'*), we have made following additional change into our scheme: the null message will not be sent unless it can carry useful information. Therefore, if a null message 'x' contains an '*EOT'* equal to the '*EOT'* sent in its preceding null message 'y', then the LP will cancel the sending of message 'x'.

In this way, the LP will not continuously send null messages when it is in '*blocked mode',* which will decrease the number of null messages in network.

## Implementation details

In this section, we will discuss all the operational activities of a slave in general and the activities of a 'simulation run' in particular. We will further explain the implementation details of our scheme for synchronization.

Each slave has established connections with the master and with every other peer slave in the network. This activity, termed as '*make connections*', has been explained in detail in section 4.2.1.

The major sub-activities, nonetheless, in this phase are:

- Establish a conversation channel with master, and
- Establish a conversation channel with all peer slaves

In next step, each slave creates the simulator nodes and initializes all of the necessary components, as explained in sections 4.2.1 and 4.2.4. This activity can be termed as *'initialization'*. The major sub-activities in this phase are:

- Create all components, such as '*remoteLP*', '*local clock*', etc.
- Load the named experiment from the database.
- Initialize all of the components, such as clocks with '*some*' initial value.
- Create node groups and local or remote versions for each simulatorNode.

After creation and initialization, the slave is ready to start the simulation run. This is where we will enforce our synchronization scheme, in order to ensure correct (and consistent) simulation results. This step is termed as '*start simulation*'. The state diagram for a slave (up to this point) is shown in Figure 19.

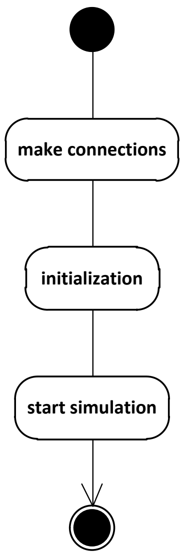

**Figure 19: (*partial*) State diagram of slave**

## Simulation run details

Initially the value of '*Local clock*' will be set to some fixed value as the experiment's starting time:

Local clock = experiment's starting time

The slave will send its '*EOT'* to every other slave before computing the EIT and determining the set of safe events. Initially, this '*EOT*' will be

EOT = LH + Local clock (experiment starting time)

Intuitively, at this point '*EIT*' will be equal to '*EOT*' because the lookahead is a fixed value (and same for all slaves), and the local clock of all slaves is set to '*experiment starting time*'. Therefore, each slave will send the same value of '*EOT*' to its peer slaves. This value is:

EOT = LH + Local clock (experiment starting time) = EIT

Each LP will compute '*EIT'* and will simulate the events having a timestamp less than this computed value. After reaching '*EIT'*, it will stop its simulation and examine the value of '*EIT'*. It will start simulating events having a timestamp less than this newly computed '*EIT'*; if there are no such events, then it will remain in '*block mode*' until it receives either some application specific message or a null message which potentially increases its current '*EIT'*.

Each LP, in this way, will simulate for a specific simulation time (until it is not safe to simulate further) and thus will advance beyond their peer LPs, then it will stop its simulation when there are no more safe events to process. At this point, the node will wait for its peer LPs to reach that same simulation point and to exchange their '*EOT*' in order to allow the LP to start its simulation from that point. This synchronous behavior of slaves is depicted in the Figure 20.

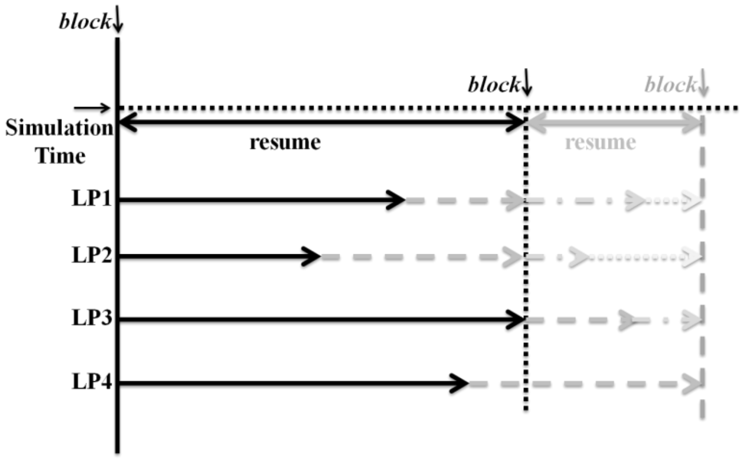

Figure 20: Synchronous simulation of four slaves

## Sending null messages

Each LP will send a null message to advance the simulation of every peer LP, before computing '*EIT'*. This message transmission is subject to the constraint introduced earlier, i.e., a null message will only be sent when it will have a new value of '*EOT*' (i.e., different from its immediate preceding one null message). As pseudo code this can be stated as:

```
IF $T_{null}$ (y) NOT EQUAL $T_{null}$ (x) THEN
        Send $T_{null}$ (y)
ELSE
        Cancel $T_{null}$ (y)
ENDIF
```

Where, $T_{null}$ (y) is the current null message, while $T_{null}$ (x) is its immediately preceding message. The format of the null message is shown in Figure 21.

| | |
|---|---|
| ***message type: null*** | ***EOT*** |

Figure 21: Null message format

Each null message and application message will contain the current '*EOT*' of a given LP. It is already explained that each slave will be exclusively in one of two modes: *simulation* or *block*. So, we have introduced two different ways to compute the value of current EOT according to the current mode of given LP.

## Computing EOT

'EOT' is a pledge of a given LP that it will not send any event in the future on a specific link to a specific peer LP containing timestamp less than this '*EOT*'. The value of '*EOT*' will be computed in one of two ways (depending upon whether this LP is blocked or not).

### Case 1: Blocked Mode

When the LP is in blocked mode, then '*EOT*' will be equal to

$$\text{EOT} = \text{LH} + \text{EIT}$$

This is because the local clock will not be advancing in blocked mode, thus if we replace '*EIT*' with the 'current simulation time' in this mode then it will try to send (one or more) '*EOT*' each time containing the same timestamp. As should be apparent there is no need to do this more than once, because CUSP guarantees a reliable and ordered delivery of messages[9]. Furthermore, our proposed algorithm cancels sending of all such null messages containing the same value of EOT. Therefore, no peer LP will receive null messages from the given LP. There may be a deadlock in case of cyclic dependency (a worst case scenario).

### Case 2: Resume Mode

When an LP is not in blocked mode, then '*EOT*' will be

$$\text{EOT} = \text{LH} + \text{Local Clock (current simulation time)}$$

This is because the local clock will be advancing with the execution of events. Furthermore, in our design, '*EIT*' will only be computing before starting the execution of safe events. So if we had chosen to send '*EIT*' in place of the local clock value, then all messages would contain the same timestamp for that span of simulation mode. Instead we send a new pledge ('*EOT*') as it may advance the simulation time.

## Sending application messages

Each LP can send application specific messages to its peer LPs at any point in simulation time. Such an application message will contain the current 'EOT' and some application specific data. The format of an application message is shown in Figure 22:

| *EOT* | *Application specific data* |
|---|---|

**Figure 22: Application message**

Since application messages will only be sent in resume mode, therefore

$$\text{EOT} = \text{LH} + \text{Local Clock (current simulation time)}$$

[9] It is claimed that, "The standard reliable transport TCP does not offer anything that CUSP cannot do equally well or better." [46]

#### Receiving a Null/Application Message

Whenever a slave receives a message, regardless of whether this message is a null or application message, it updates the clock associated with the link of the LP which sent this message (as a result, this may also update '*EIT*'). After updating its remote clock, it will re-compute its '*EIT*', and if this new '*EIT*' is greater than the previous value of '*EIT*', then if it is blocked, it examines if it is now safe to resume and if so, then it switches itself to the simulation mode from blocked mode, otherwise it remains in the blocked state.

In simulation mode, the '*Executer*' removes safe events from the execution queue and processes them until it reaches the check bound, i.e., the computed value of '*EIT*'. Upon reaching the check bound, the '*Executer*' stops its simulation and examines if it has received messages that have already increased '*EIT*' and subsequently the check bound, as well; if so, then it resumes simulation until the new check bound and repeats its behavior; otherwise it remains in '*blocked mode*' and waits for other LPs to reach the same simulation time point.

The simulation state of each slave will flip-flop between two states: *block and resume* (as shown in Figure 23).

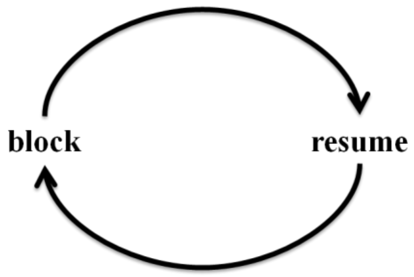

Figure 23: Flip-Flop of Simulation states

Consequently, all the LPs will simulate in a synchronized fashion, and the simulation will not produce erroneous results as all simulations will only occur within the safe limits.

This is how our scheme fulfils the desired properties of system, as discussed in section 2.1.2.3, and preserves correctness:

- **Liveliness:** Each event in the event list will *eventually* be processed successfully in the correct timestamp order.
- **Safety:** Cause always precedes the effect; each LP processes events in non-decreasing timestamp order and thus adheres to the '*local causality constraint*'.

### 4.2.7 Gathering and writing simulation results to the database

Slaves will produce simulation results during a simulation run. These results are critical as this is the only way to analyze the behavior of an experiment running on the simulator. The simulator nodes produce log entries for debugging purposes, and/or to report some measurements (such as transmission speed, number of bytes uploaded, etc.). These measurements will be referred to as "statistics" because the simulation results will be in the form of statistical data, such as an average of values, sum of squared values for calculating standard deviation, etc. These statistical results are very important to realistically evaluate the behavior of any (P2P or network) application.

As per our model, the slaves will run the named experiment and will produce simulation output. This output will not be written to the database by the slaves themselves, but rather each slave will send its results to the master. Additionally, the master will not be running any simulation and therefore will not generate any output of its own, instead it will simply gather the outputs from all the slaves and write this information to the database.

There will be two types of output produced and sent by slaves: log message and statistics. We have proposed two different schemes for dealing with these two types of output. We will describe them separately in following paragraphs.

### i. Log messages

Log messages are generated to keep the track of the simulation run's output and are expected to be used mainly for debugging purpose. These log messages must be stored somewhere (usually into the database) to produce a persistent record.

The processing of an event, in our model, may generate many log messages. This means that all such log messages will share the same timestamp as the event which originates them. The slaves will send these messages to the master for writing into the database. These log messages must be written in the same order as they were issued, but unfortunately there will be no information in these messages which could assist the master in ordering these messages. The gravity of problem further increases when more than one log message is generated and sent by a slave during the processing of a single event.

We have already described in section 4.2.1, a slave will establish a conversation channel with the master before starting its simulation. This will create an outStream for sending messages through the network. Furthermore, we have stated that an assumption of our model is that the network and CUSP will preserve the order of the messages as they were sent and will also provide reliable delivery. So, whenever a log message is generated it will be sent to the master using the established connection and by sending it through an outStream. This outStream will ensure that all messages will be received in the same order as they were sent.

This integrity of log messages provided by outStream and network jointly solves the problem of writing the log messages into the database in the correct order by the master. So, all a slave needs to do is to send the log message as soon as it is generated; then the outStream and the network will jointly ensure that the master will receive all these messages in the same order as they were originally sent. The master can write these messages in the order it receives them. This ensures that the master will write the log messages to the database in the correct order. This process is shown in Figure 24.

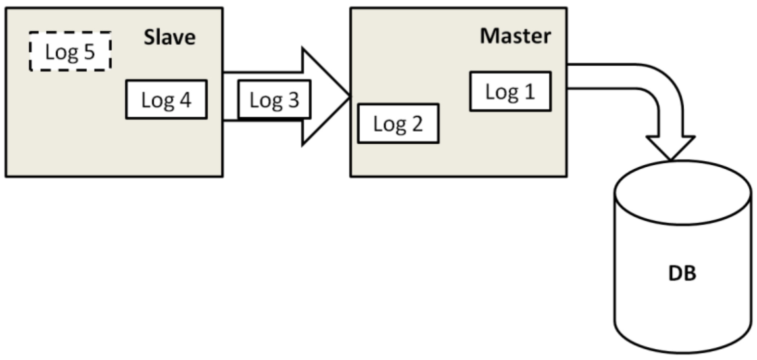

Figure 24: Writing log messages to the DB

### ii. Statistics information

Simulation results are frequent measurements (such as transmission speed, number of bytes uploaded, etc.) produced as the result of a simulation run. This information is accumulated to give a meaningful result. This accumulated information is referred to as ***statistics*** in our model.

These statistics are essential to visualize the simulation results in a graphical format. They are stored and aggregated in a collector within a logging period and then are sent to a statistics writer in aggregated form. Subsequently, they will be written to the database. The goal is to have a single record per statistic and log period in the database. Such an entry, in our model, is called a ***measurement.***

At the start of each simulation run, a 'new' statistic of the given name will be written to the DB and its values (measurements) will be updated periodically. Subsequently, multiple measurements may be generated corresponding to one specific statistic. A '*measurement*' consists of:

1. ***Average*** of all values since the last update. Following two measures are computed for calculating the average:
   - **i.** ***Sum*** of all values since the last reset.
   - **ii.** ***Count*** of all datapoints or values since the last reset. Both these values are needed to compute an average, i.e.,

$$\text{Average} = \text{Sum} / \text{count}$$

2. ***Minimum and Maximum*** of values since the last update.
3. ***Standard deviation*** of all values added since the last update. Additionally, following measure is computed to aid in calculating standard deviation.
   - i. ***Sum of Squared*** values for calculating the standard deviation.

A '*statistic'* consists of:

- ***Name*** of the statistic,
- ***Label*** associated with this particular statistic, and
- ***Type*** of the statistic.

All measurements are categorized with respect to their *'name'*, *'label'*, and *'type'* (statistic). So, there must be some mechanism for writing all measurements into the database based upon their particular category (as identified by the statistic's triple). The relationship between a statistic and a set of measurements is shown in Figure 25.

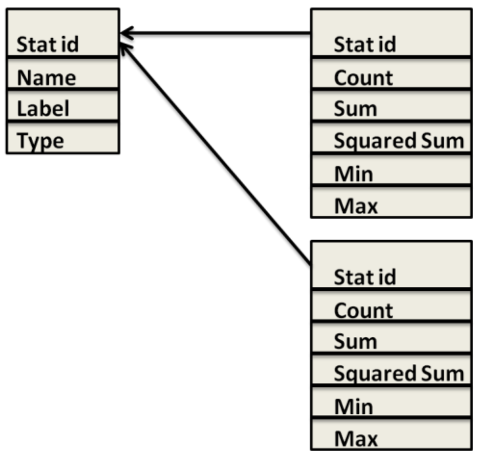

Figure 25: Statistic-Measurements

In Figure 25, *'Stat id'* is used to couple all the measurements with their corresponding statistic, for writing into the database. This scheme should work if each slave itself writes these measurements into the database, but in our model the slaves sends all this information to the master which in turn stores the measurements into the database. All these measurements must be updated corresponding to the category they belong to. Therefore, there must be a consistent '*Stat id*' on both master and slaves to ensure storage of the correct simulation results. For example, measurements sent by a slave under the category of '*statistic 1*' must be received and updated by the master as measurements belong to '*statistic 1'*, and so on. This is illustrated in Figure 26.

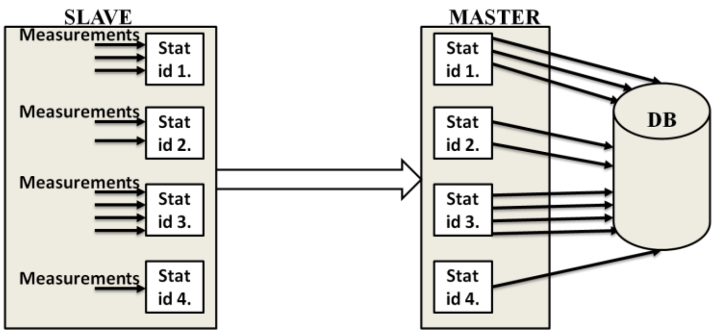

Figure 26: Sending and receiving measurements

Nevertheless, this master/slave paradigm raises problems of how to handle multiple measurements belonging to the same category on the master's side; i.e., how should the master will map a given measurement to its corresponding category? The severity of the problem increases due to the fact that the identifier *'Stat id'* is provided by the database which, in turn is, accessible only to the master.

We have formulated a novel systematic approach for sending/receiving simulation results over the network and writing them into the DB. Whenever a slave needs to send measurements to the master, it will first send the *'statistic'* corresponding to that measurement. The master will, then, check whether it already has stored this category (*'statistic'*) into the DB, if so then it will send the identifier *'Stat id'* to the slave; if this is the first time that it has received this *'statistic'* then the master will store this *'statistic'* into the DB and will send the *'Stat id'* corresponding to that particular *'statistic'*. The slave will send measurements along with this *'Stat id'* to the master; this additional information (*'Stat id'*) assists the master in associating this received measurement with its corresponding 'statistic' and to correctly store the measurement into the DB. This process is shown in Figure 27.

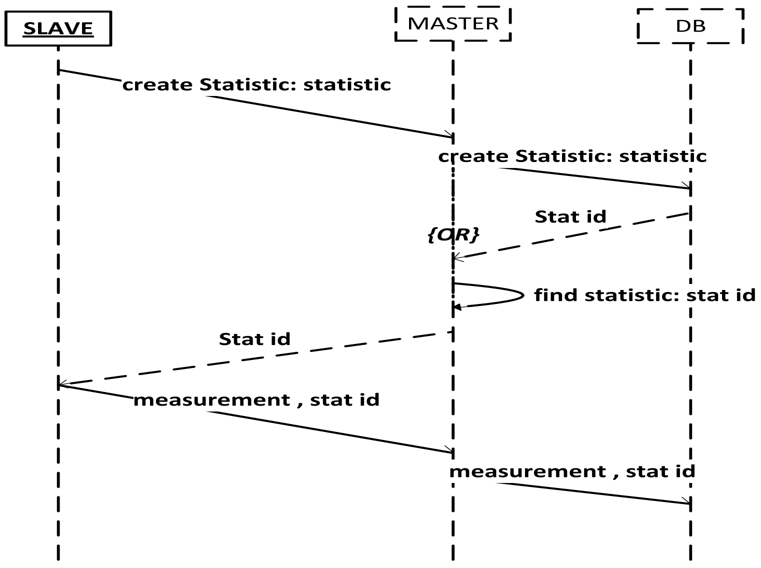

Figure 27: Sending-Receiving-Storing a statistics measurement

Statistic forwarding and storage into the DB is a four step activity:

1. A slave will forward a *'statistic'* to the master in order to store this in the DB.
2. The master will receive this statistic and examine whether it has already registered such a statistic:
    a. If so, then it will retrieve its corresponding id;
    b. Otherwise, it will store this new *'statistic'* into the DB.

    In either case, it will send the corresponding *'Stat id'* to the slave.
3. The slave will then send a *'measurement'* along with the *'Stat id'* corresponding to a particular *'statistic'*.
4. The master will receive and store this measurement into the DB, under its appropriate *'statistic'* based upon the *'Stat id'*.

In this way, we store simulation results accumulated within a logging period in order to collect meaningful information.

# Chapter 5.

# Evaluation and Analysis

This chapter presents an evaluation of the HydraNetSim (based on our proposed architecture explained in chapter 4). As explained in section 1.2.1, we are interested in examining the:

1. simulation speed of HydraNetSim in comparison with an SDES given the same (settings of the) experiment, and the
2. scalability (in terms of support for multiple parallel simulation nodes) of HydraNetSim in comparison with an SDES.

## 5.1 Environment setup

To explore the above mentioned features of HydraNetSim, we have chosen a simulation scenario of BuubleStrom [50] as our test scenario. BubbleStorm is defined as "a rendezvous-based peer-to-peer search overlay" [57]. It is a probabilistic platform for exhaustive search based on random multi-graphs [58]. Generally, a rendezvous system is a distributed system that ensures that a query must meet every datum available in the system (it means that at least one node that has a copy of the data item will execute the query in its local dataset) [57 - 58]. The underlying topology of Bubblestorm can easily deal with crashes and node churn by connecting to different participants, as there is no need to maintain a particular tree shape [56]. The novel communication primitive (bubblecast) of the BubbleStorm on this topology provides optimal per-node bandwidth complexity even when every query must rendezvous with every datum [58]. For further details, please read [50], and [56 - 58].

We will consider the following two potential modes of the simulator:

1. Non-parallel/ sequential (i.e. SDES), and
2. Parallel (i.e.) with 2x cores slaves.

The non-parallel mode is a SDES. This mode will provide a point of reference for evaluating the simulation speed and scalability of HydraNetSim. We will test the HydraNetSim under the following three modes:

1. HydraNetSim -1Slave,
2. HydraNetSim-2Slave, and
3. HydraNetSim-3Slave.

The tests are run on two different machines. The system details of both machines are given in table 9.

Table 9: System details of the two platforms that have been used to run the simulations

| | Processor | No. of Cores | No. of Threads | RAM (in GB) | Operating System |
|---|---|---|---|---|---|
| **1.** | Intel® Core™ i5-2410M [59] | 2 | 4 | 6 | Windows 7 Professional – 64 bit |
| **2.** | Intel® Core™ i7- 920 [83] | 4 | 8 | 6 | Windows 7 Professional– 64 bit |

## 5.2 Parameter configuration

Tests are based on a simulation scenario using BubbleStorm, which is referred to as KV-BS. The tests are performed with the parameters settings as shown in Table 10.

Table 10: Parameters settings for the simulations

| Parameters | Values | | |
|---|---|---|---|
| | Performance test | Scalability test | |
| **Experiment scenario name** | KV-BS | KV-BS | KV-BS |
| **Average Number of online nodes** | 1,000 | 2,000 | 3,000 |
| **Number of fixed[10] nodes** | 19 | 19 | 19 |
| **Simulation time for accomplishing the scenario** | 5h | 5h | 5h |
| **LH** | Last hop delay | Last hop delay | Last hop delay |
| **Total (online + offline) number of nodes** | 19,600 | 39,600 | 59,600 |

As explained in section 4.2.7, the simulation results are sent by the slaves to the master when running the simulator (HydranetSim) in parallel mode. This sending of results may incur network delay and is also dependent on the capacity of communication protocol (CUSP in this case), which is not the case when running the simulator (as a SDES) in single mode. For the sake of a fair analysis, therefore, we run our all tests (whether running in SDES mode or HydraNetSim) by factoring out the time taking by sending, receiving, and storing the simulation results to the DB. Furthermore, all experiments are run on multi-core computers, where each slave is assigned to a different core and to a specific CPU.

## 5.3 Performance test

In this section, we will examine the resource usage (specifically the memory consumption and CPU utilization) and actual execution time for completing a given simulated scenario. At

[10] Bootstrap + Default Login nodes

first, we will run KV-BS in the non-parallel mode of HydraNetSim (to investigate the behavior of a SDES) and then we will compare these results with different parallel modes of HydraNetSim. The test case scenario for all tests is KV-BS with 19,600 total (online + offline) nodes, and 5 runs of the given scenario are made for each mode. Because of using (global & local) RNGs, the resources (CPU and memory) consumption was identical in each run for a given mode of simulator. However, we have presented the readings in the following three statistical measurements[11]:

**Maximum** The maximum value of memory utilization at any point in a complete simulation run.

**Minimum** The minimum value of memory consumption in a complete simulation run. Usually this value comes when the simulation nodes were not created at their full strength (at start of the simulation).

**Average** The average consumption of memory by the given mode in a specific simulation run.

## 1) Non-parallel mode (SDES)

Table 11 shows the resource utilization during a specific simulation run by SDES (in non-parallel mode) on different points of time.

Table 11: Resource utilization by SDES

| | **SDES** | |
|---|---|---|
| | CPU | Memory |
| **Maximum** | 99.844% | 1.4772GB |
| **Average** | 99.484% | 1.4372GB |
| **Minimum** | 98.659% | 0.7590GB |

The memory consumption by SDES is shown in figure 28, and the CPU utilization is shown in figure 29.

[11] Rounded to 5 significant digits.

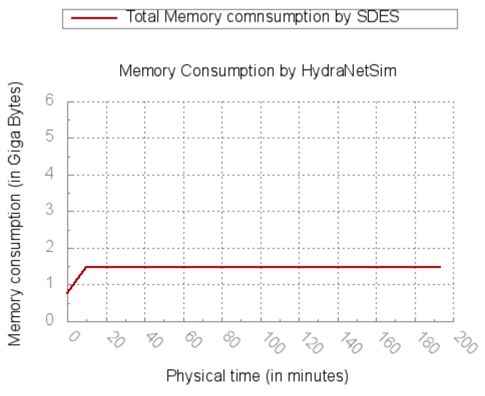

Figure 28: Memory Consumption by SDES as a function of time

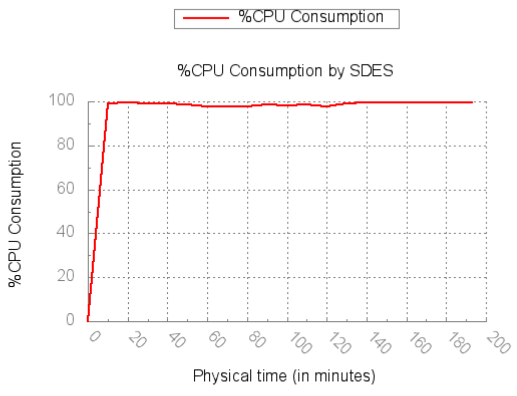

Figure 29: %CPU consumotion by SDES as a function of time

To complete the given simulation scenario, SDES took ***3 hours, 13 minutes and 51 seconds***[12] (~194 minutes in total).

## 2) HydraNetSim – 1Slave

HydraNetSim-1Slave took ***3 hours, 14 minutes and 43 seconds*** (~195 minutes in total) to complete the KV-BS simulation scenario. The memory consumption as a function of time is shown in figure 30.

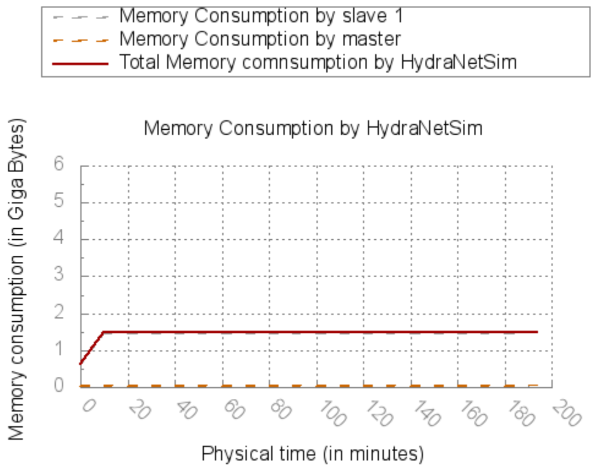

Figure 30: Memory consumption by HydraNetSim-1Slave as a function of time

The resource utilization by HydraNetSim-1Slave is given in table 12.

Table 12: Resource utilization by HydraNetSim-1Slave

| | **Memory(GB)** | | **%CPU** |
|---|---|---|---|
| | Master | Slave | Slave |
| **Maximum** | 0.02566 | 1.4733 | 99.431 |
| **Average** | 0.02564 | 1.3782 | 99.283 |
| **Minimum** | 0.02563 | 0.6173 | 98.610 |

[12] The simulation run completion time is physical time.

## 3) HydraNetSim-2Slave

Both slaves run on two specific CPUs on different cores. Table 13 represents the resources utilization by both slaves.

Table 13: Resource utilization by HydraNetSim-2Slave

| | %CPU | | Memory (GB) | | |
|---|---|---|---|---|---|
| | Slave1 | Slave2 | Slave1 | Slave2 | Master |
| **Maximum** | 99.874 | 99.870 | 1.4716 | 1.4732 | 0.02565 |
| **Average** | 99.771 | 99.801 | 1.3140 | 1.2782 | 0.02564 |
| **Minimum** | 99.407 | 99.615 | 0.6358 | 0.5776 | 0.02561 |

The memory consumption by '*HydraNetSim-2Slave*' as a function of time is shown in figure 31.

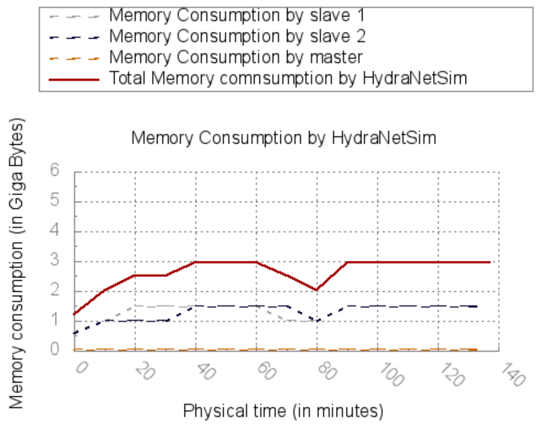

Figure 31: Memory consumption by HydraNetSim-2Slave as a function of time

HydraNetSim-2Slave took ***2 hours and 17 minutes and 14 seconds*** (~137 minutes in total) for completing the simulation run of KV-BS.

## 4) HydraNetSim-3Slave

All three slaves were assigned to 3 different CPUs on different cores; it took ***2 hours and 37 minutes and 43 seconds*** (~157 minutes[13] in total) for completing the simulation scenario of KV-BS. The resource utilization is shown in table 14.

Table 14: Resource utilization by HydraNetSim-3Slave

| | %CPU | | | Memory (GB) | | | |
|---|---|---|---|---|---|---|---|
| | Slave1 | Slave2 | Slave3 | Slave1 | Slave2 | Slave3 | Master |
| **Maximum** | 99.894 | 99.527 | 99.756 | 1.0889 | 1.4622 | 1.1143 | 0.02566 |
| **Average** | 99.835 | 99.406 | 99.651 | 0.9797 | 1.0070 | 0.9851 | 0.02139 |
| **Minimum** | 99.665 | 99.051 | 99.340 | 0.3783 | 0.3735 | 0.3734 | 0.00957 |

[13] Note this value is longer than the results of HydraNetSim-2Slave. We will analyze this fact in analysis section of this chapter.

Figure 32 shows the overall memory consumption as a function of time by HydraNetSim-3Slave.

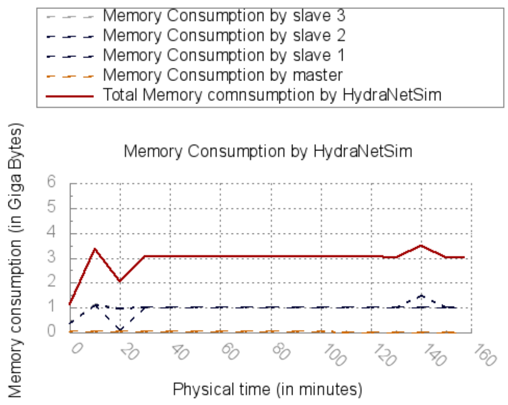

Figure 32: Memory consumption by HydraNetSim-3Slave as a function of time

## 5.4 Scalability test

To test the maximum of nodes supported by HydraNetSim, we ran the same simulation scenario[14] as explained in section 5.1, but with different parameter settings. First we increased:

- the total number of participating nodes from 19,600 to 39,600, and
- the number of average online nodes from 1000 to 2000.

Then we further increased the

- total number of participating nodes to 59,600, and
- the number of average online nodes to 3000.

The SDES version did neither support 39,600 nodes nor 59,600 nodes and simply crashed. The same results occurred with the HydraNetSim-1Slave. Whereas, HydraNetSim-2Slave supported the simulation scenario of 39,600 total nodes and took ***5 hours, 4 minutes and 50 seconds*** (~305 minutes in total) for completing the simulation run. However, the HydraNetSim-2Slave did not support the scenario of 59,600 total participating nodes and simply crashed. The resource utilization by HydraNetSim-2Slave is shown in table15.

Table 15: Resource utilization by HydraNetSim-2Slave

| | %CPU | | Memory (GB) | | |
|---|---|---|---|---|---|
| | Slave1 | Slave2 | Slave1 | Slave2 | Master |
| **Maximum** | 99.899 | 99.899 | 1.4692 | 1.4687 | 0.03114 |
| **Average** | 99.664 | 99.611 | 1.3882 | 1.4319 | 0.02798 |
| **Minimum** | 98.918 | 98.762 | 0.1735 | 0.8809 | 0.02771 |

HydraNetSim-3Slave supported both simulation scenarios of 39,600 and 59,600 total participating nodes, and took ***4 hours, 52 minutes and 48 seconds*** (~293 minutes in total) for

[14] KV-BS

completing the scenario with 39,600 nodes. The memory consumption by HydraNetSim-3Slave and HydraNetSim-2Slave are shown in figure 33 and figure 34, respectively.

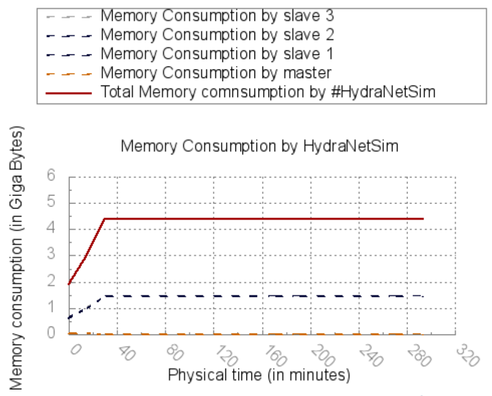

Figure 33: Memory Consumption by HydraNetSim-3Slave (39,600 nodes) as a function of time

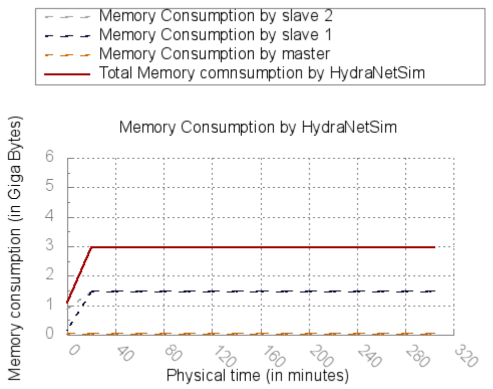

Figure 34: Memory Consumption by HydraNetSim-2Slave (39,600 nodes) as a function of time

## 5.5 Summary of results

In this section, we analyze the test results and validate our hypotheses[15]. Table 16 shows a summary of the all tests that were performed.

Table 16: Summary of experimental results

| | | SDES | HydraNetSi-1 Slave | HydraNetSi-2 Slave | HydraNetSi-3 Slave |
|---|---|---|---|---|---|
| No. of CPUs (~100% utilization) | | 1 | 1 | 2 | 3 |
| Support (No. of Participating nodes) | 19,600 | ✓ | ✓ | ✓ | ✓ |
| | 39,600 | × | × | ✓ | ✓ |
| | 59,600 | × | × | × | ✓ |
| Memory consumption (avg. in GB) | 19,600 | 1.4372 | 1.40384 | 2.61784 | 2.99319 |
| | 39,600 | - | - | 2.84808 | 4.25100 |
| Physical Time(min) | 19,600 | 194 | 195 | 137 | 157 |
| | 39,600 | - | - | 305 | 293 |

We can see by carefully examining the results that HydraNetSim-1Slave performs no better than SDES and utilizes almost the same amount of resources, whereas HydraNetSim-2Slave performs 1.42 times faster than SDES, and consumes 1.18 times more memory than SDES. Furthermore, it can support 20,000 more nodes than SDES. HydraNetSim-3Slave performs 1.24 times faster than SDES, and consumes 2.08 times more memory than SDES. It supports 3.04 times more nodes than SDES.

[15] These hypotheses were explained in section 1.2.1.

SDES takes 41.6% more time than HydraNetSim-2Slave and 23.6% more (physical) time than HydraNetSim-3Slave, to complete a simulation run with same settings. HydraNetSim-3Slave takes 14.6% more (physical) time [16] than HydraNetSim-2Slave to complete a simulation scenario consists of 19,600 total participating nodes, and takes 1.14 times more memory than HydraNetSim-2Slave. But when we increased the (total) number of participating nodes, then HydraNetSim-3Slave takes 3.93% less (physical) time than HydraNetSim-2Slave and takes 1.49 times more memory than HydraNetSim-2Slave for completing a simulation scenario consists of 39,600 total participating nodes.

HydraNetSim-2Slave needs only 1.09 times more memory, and takes 2.23 times more (physical) time when we double [17] the scale of simulation scenario, whereas HydraNetSim-3Slave needs 1.42 times more memory, and it takes 1.87 times more (physical) time when we increase the scale of simulation scenario from 19,600 to 39,600. HydraNetSim-3Slave took 1.007 times more memory, and 1.49 times more (physical) time more time when we increase the scale of simulation scenario from 39,600 to 59,600. Figure 35 and figure 36 show the comparison in (physical) time and memory consumption respectively, by different modes of HydraNetSim and SDES for completing a given simulation scenario.

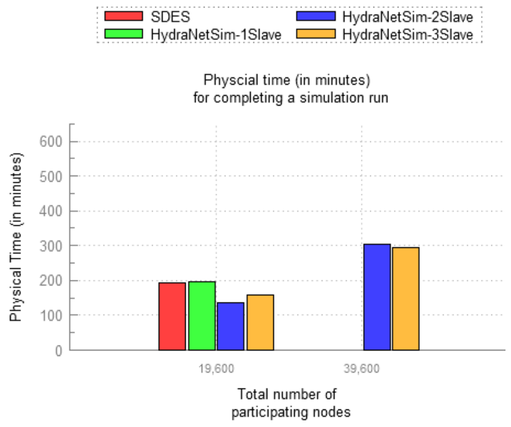

Figure 35: Comparison of (physical) time for completing a simulation run

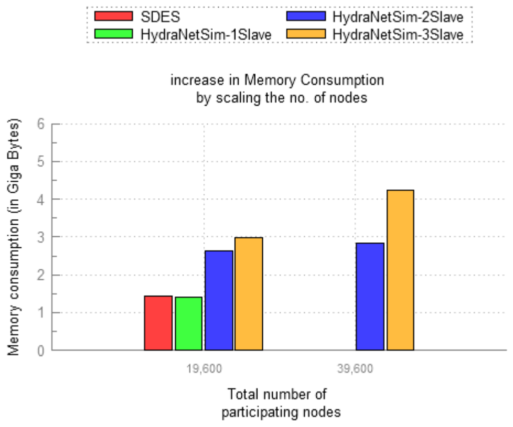

Figure 36: Comparison of memory consumption for a complete simulation run

## 5.6 Analysis of results

Results validate our both hypotheses that

i. HydraNetSim performs faster than SDES, and
ii. HydraNetSim supports a greater number of simulation nodes and runs a larger simulation scenario than SDES.

As we have noticed that HydraNetSim-3Slave performs slower (in executing the simulation scenario of 39,600 simulated nodes) than HydraNetSim-2Slave, but still it is 1.24 times faster than SDES and can support 3.04 times more nodes than SDES. Amdahl's law[18] explains this

[16] We will analyze the fact in subsequent paragraphs.
[17] Here, by double we mean to increase the average number of nodes from 1,000 to 2,000, and to increase total number of participating nodes from 19,600 to 39,600.
[18]As explained in section 12.1.2.2

to a degree that the speed up of a program using multiple processors in parallel computing is limited by the sequential portion of that program. But, in case of HydraNetSim, this sequential portion belongs to the simulation model instead of our proposed architecture of HydraNetSim, because the application code of each slave is completely independent and there is no sharing of components or data structures in slaves. Furthermore, each slave has its own virtual memory, event handling routines, data structures, etc, and runs its part of the simulation model independently of other slaves. All the dependency of slaves lies into the adherence to causality constraint[19]. Therefore, the simulation speed is limited to the two factors: 1) the sequential fraction of the simulation model, and 2) the synchronization overhead[20].

In section12.1.2.2, we have discussed the implications of Amdahl's law in detail and analyzed the relation of the sequential portion of a given program to the execution time with respect to the increasing number of processors. But note that Amdahl's law does not consider different overheads (such as setup time, data communication time, synchronization overhead, etc.) associate with parallelization. In some situations, these overheads can cause the worst performance of a program (as discussed in subsequent paragraphs) running in parallel on multi-cores.

Let's use a simple thought experiment to see the effects of synchronization overhead on the overall speedup of a given program by running it parallel on different processors. Assume that the execution of the parallel fraction of a given program takes 50 minutes and the execution of the sequential portion takes 25 minutes. Therefore, the execution time for running the given program by 1 processor is 75 minutes. However, if we run the program on 10 processors, then the execution time reduces to 30 minutes (i.e., 5 minutes for the parallel portion and 25 minutes for the sequential portion); by running the program in parallel on 50 processors reduces the time to 26 minutes (i.e., 1 minute for the parallel portion and 25 minutes for the sequential fraction) which is 1.15 times faster than 10 processers and 2.88 times faster than 1 processor. Unfortunately, things are not so simple; running a program in parallel usually labor with some overheads such as setup time, data communication time, synchronization, etc. Now, assume that the synchronization overhead is 1 minute for each processor. Surprisingly, the total execution time (as shown in table 17) taken by 10 processors rises to the 40 minutes, and by 50 processors rises to the 76 minutes which is even worse than running the program on a single processor.

[19] Each slave waits for getting EOT from other slaves to increase its EIT, to determine the set of safe events.

[20] The magnitude of synchronization overhead depends on the algorithm used for the synchronization.

Table 17: Execution time of a given program by different number of processors

| Execution Time (in minutes) \ Number of Processors | 1 | 10 | 50 |
|---|---|---|---|
| **Parallel Portion** | 50 | 5 | 1 |
| **Sequential Portion** | 25 | 25 | 25 |
| **Synchronization overhead**[21] | 0 | 10 | 50 |
| **Total** | 75 | 40 | 76 |

The ratio of sequential and parallel fractions is an inherited property of the simulation model, yet the synchronization overhead is a property of synchronization algorithm. In our proposed algorithm, we have tried to keep this synchronization overhead as low as possible. For this, we rely more on application specific messages (as we piggyback the EOT information on an application message) to reduce the magnitude of synchronization overhead in the network as much as possible, to improve performance. It means that *higher the number of application messages* (containing a unique EOT) being sent between LPs, *lower the number of block-resume-block cycles*; this leads to a relatively faster simulation speed of simulator. Therefore, another reason that HydraNetSim-2Slave performs faster than HydraNetSim-3Slave may be because there are only 2 LPs which communicates with each other frequently, therefore they go into block mode less frequently and thus most of the (physical) time the HydraNetSim remains in simulation mode. But in case of HydraNetSim-3Slave, there are 3 LPs and each LP may not have sufficient application specific messages for both other LPs all the time. Therefore, the LP which has an EIT less than the timestamp of the event T ($E_x$) to be executed next, goes into block mode and has to wait until its EIT is greater than T($E_x$); which ultimately limits the overall simulation performance as HydraNetSim remains in blocked mode.

It is possible that an LP can be less dependent on other LPs (and can simulate partly independent of other LPs for a period of simulation time), but under the current settings of our proposed design that LP has to wait for EOTs from other LPs to determine (and the execution of) its safe events. This wait and resume cycle of an LP effects the overall performance of HydraNetSim. This may also be a reason for the relatively poor performance of HydraNetSim-3Slave. This also yields that our proposed variant of NMA performs better when LPs communicates with each other more frequently and there are less block-resume-block cycles which is possible if there are more application specific messages sent in between LPs or the value of lookahead is great enough to produce a larger set of safe events.

[21] The synchronization overhead is directly proportional with the number of processors, in this experiment.

Another potential reason for this behavior of HydraNetSim-3Slave can be the inadequate lookahead value. Consider figure 37 which illustrating this.

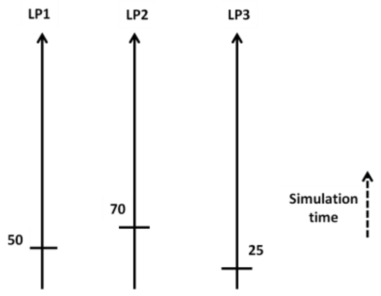

Figure 37: Lookahead inadequacy

In figure 37, all three LPs are blocked on simulation time 50, 70, and 25 respectively. Now, according to our synchronization protocol, each LP will send null messages containing its EOT[22]. Intuitively, all three LPs will pass null messages containing EOT of LP3 because

$$\text{EIT } \textit{of } \text{LP}_2 = \text{EIT } \textit{of } \text{LP}_1 = \text{EOT } \textit{of } \text{LP}_3$$

The gravity of the problem increases when the next event scheduled on $\text{LP}_3$ and $\text{LP}_1$ has a timestamp at least equal to the event scheduled on $\text{LP}_2$, and the value of LH is relatively small such as 2ms; then, there must be many rounds of null message passing between the LPs before they can resume and start simulation.

However, if we do not have two choices for EOT with respect to the LP's modes[23] then we may get into the zero-lookahead cycles. To illustrate this, consider figure 38.

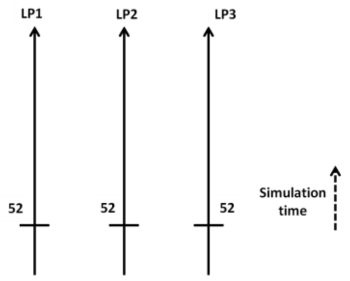

Figure 38: zero-lookahead cycles

In figure 38, three LPs are shown in block mode having the same current simulation time, i.e. 52. Now, if we set the

$$\text{EOT} = \text{current simulation time}^{24} + \text{LH}$$

even in the *blocked* mode, then all null messages from each LP would have contained the same information; this scheme may cause a deadlock. That is why we adopted the previously explained scheme for null message passing. However, the given scheme experiences

[22] If the LP is in block mode, then

$$\text{EOT} = \text{EIT} + \text{LH}$$

otherwise,

$$\text{EOT} = \text{Current Simulation time} + \text{LH}$$

[23] As we have explained in chapter 4, each LP will be exactly in one of two modes: *Block or Resume*.

[24] Here the current simulation time depicts the timestamp of the last executed (application) event.

problems with an inadequate lookahead value, and ultimately negatively affects the overall performance of HydraNetSim. Furthermore, that scheme results in all of the LPs simulating at the pace of the slowest LP; which also slows the pace of HydraNetSim. There may be many workarounds for this problem; however, we leave solving this problem to future work.

Another important finding is that the execution time by a given mode (2Slave or 3Slave) of HydraNetSim for completing the simulation scenario at a greater scale (39,600 simulated nodes) is more than the execution time for smaller scale (19,600 simulated nodes) of the same mode (2Slave or 3Slave). It is because of the fact that the complexity of the simulation model and the message density of the simulation model also increases due to the increase in the number of participating nodes. Hence, simulation of a larger scenario takes more physical time and more physical memory than the simulation of a smaller scenario.

An interesting point to note here is that HydraNetSim-3Slave performs relatively better than HydraNetSim-2Slave when we increase the size of simulation scenario from 19,600 simulated nodes to the 39,600 simulated nodes. This increase in the number of simulated nodes also increases the message density of the simulation model, which yields more application specific messages in the simulation model. Hence, we observe an improvement in performance by HytdraNEtSim-3Slave, as we have explained in above paragraphs that our proposed parallel solution performs better on running more processers when there are a higher number of application specific messages in the simulation model.

Interestingly, we can gain benefit from parallel processing to a specific limit, even by increasing the problem size. To illustrate the reason, assume that the problem size explained in table Table 17 is increased[25] to 15 times. Now, 1 processor will execute the program in 1125 minutes; the use of 10 processors will dramatically decrease the time to 460 minutes (a 2.45 times speedup); the use of 50 processors reduces the time to 440 minutes (2.56 times faster than single processor). If we compare the results shown in table 18 with results of table 17, then we can observe a speedup in performance of running a larger program on 50 processors.

Table 18: Execution time of a given program by different number of processors

| Number of Processors / Execution Time (in minutes) | 1 | 10 | 50 | 100 |
|---|---|---|---|---|
| **Parallel Portion** | 750 | 75 | 15 | 7.5 |
| **Sequential Portion** | 375 | 375 | 375 | 375 |
| **Synchronization overhead** | 0 | 10 | 50 | 100 |
| **Total** | 1125 | 460 | 440 | 482.5 |

[25] Keeping all other factors (such as overhead ratio, parallel to sequential portion ratio, etc.) constant.

However, note (in table 18) a decrease in performance by increasing the number of processors from 50 to 100, as the execution time rises to 1.097 times more than the execution time by 50 processors. It shows that we can gain benefit from parallel processing only to a certain limit[26] which depends upon different factors such as ratio of parallel and sequential fractions of the given program, overheads[27] associated with parallel execution, number of processors, computing power of given hardware, etc.

Table 18, also demonstrates the behavior described by the Gustaffson's law that we can solve larger problems in the same time by using more parallel equipment. A 10 times larger problem than the one assumed in table Table 17, takes 750 minutes by running on a single processor, whereas table 18 shows that by using 50 processors we can solve a 15 times larger problem in 440 minutes, and even a 25 times larger program needs only 650 minutes by 50 processors. HydraNetSim-3Slave has also shown the same behavior: SDES takes 194 minutes for running a simulation scenario of 19,600 nodes, and when we increase the scenario to 39,600 (i.e., ~ 2x times larger than the previous scenario) then *intuitively* it should have taken ~(2 x 194 = 388) minutes[28] whereas HydraNetSim-3Slave takes 293 minutes to simulate the scenario of 39,600 nodes.

From all above discussion, we can identify the relation of execuation time of HydraNetSim to complete a given simulation scenario with different influencing factors

$$P_H \propto \frac{\alpha\rho\tau\delta}{EnP\lambda L} \qquad 6.1$$

Where,

- $P_H$ represents the performance of HydraNetSim (in terms of Physical time for executing a given scenario),
- $\alpha$ represents the non-parallelizable fraction of the simulation model,
- $\rho$ represents the synchronization overhead,
- $\tau$ represents the message latency of the physical simulation hardware,
- $\delta$ represents the complexity (i.e., in terms of participating simulated nodes) of the simulation model,
- $E$ represents the event density of simulation model,
- $n$ represents the number of slaves,
- $P$ represents the computing power of physical simulation hardware in terms of processing events per second,
- $\lambda$ represents the coupling factor of LPs in terms of application messages, and

[26] We leave this to future research, to find this limit and the influence of the underlying factors (such as overheads, the computing power of the given hardware, ratio of parallel and sequential fraction, etc.) on this limit.

[27] Such as synchronization overhead, setup time, etc.

[28] We recommend, this must be verified in future by increasing the scalability of sequential counterpart of HydraNetSim (SDES) to at least 39,600 nodes.

- $L$ represents the lookahead.

All these factors affect the performance of HydraNetSim in terms of (physical) time to complete a given simulation run. More importantly, blocking of LPs is affected by the fluctuations of E, τ, λ, and L (as we have discussed above). However, the effect of all these factors must be analyzed in terms of the overall performance of HydraNetSim and the degree of proportionality must also be determined in detail; we leave this to future work.

Furthermore, we can also identify a pattern in increase in scalability of HydraNetSim[29]by incrementing the numbers of slaves.

$$S_{H\beta} = \alpha + (\beta - 1)k \qquad \forall \beta > 0 \qquad 6.2$$

Where,

- $S_{H\beta}$ is number of (maximum number of) nodes supported by HydraNetSim-$\beta$ slave,
- $\alpha$ represents the (maximum number of) nodes[30] supported by SDES,
- $\beta$ represents the (total) number of slaves, and
- $k$ represents a constant value[31].

And the scalability factor is $\lceil(\beta - 1) * 102.04\%\rceil$ of α.

[29] Under the parameter settings given in Table 10: Parameters .
[30] 19,600 under the current settings.
[31] This constant value is determined by the ratio of online_time and offline_time of the nodes. However, this constant value is 20,000 according to the settings given in Table 10: Parameters

# Chapter 6.

# Conclusion & Future work

## 6.1 Conclusion

Computer simulation has become an essential part of research and development (R&D) in various fields [7] over the years, including communication systems, architecture, engineering, production management, military, business, government, logistics, transportation, etc. Discrete event simulation is more popular than any other type of simulation because it provides faster results and also imitates the system's behavior in a convincingly accurate manner. The evaluation and analysis of complex systems, such as P2P mostly rely on simulation [64]. Parallel Discrete Event Simulation (PDES) has been an active area of research for at least three decades, to deal with the continuously growing computational complexity and structural complexity of systems such as P2P networks. Recent technological advances have greatly reduced the prices for parallel computing hardware and thus have made such hardware available to a larger research community. Although many P2P discrete event simulators[32] can support thousands of thousands simulated nodes, surprisingly none of the main stream P2P discrete event simulator (covered in surveys [65-68]) support parallel simulation to exploit the multi-core architecture of the modern computing hardware [64]. Furthermore, the P2P simulators such as PlanetSim do not provide any mechanism for collecting statistics.

In this thesis, we have explored the fundamental challenges in developing a PDES and proposed a new design for a discrete event simulator focusing on parallelization, specifically suited for multi-core machines. We have divided the simulation model into a cluster of processes (several partitions) called slaves, which are managed and coordinated by a master. The master does not directly participate in the simulation, rather the role of the master is to instruct the slaves what simulation to load, and then to write the simulation results into a DB after receiving them from slaves. Furthermore, we have proposed a variant of classic NMA for keeping all partitions in synchronization with a focus on keeping the network overhead for synchronization (the traffic of null messages) as low as possible. Another contribution is a novel mechanism for gathering the log information for debugging purposes and for collecting statistics over a specific period of simulation time in order to get meaningful simulation results.

We evaluated the performance of HydraNetSim (our proposed PDES) in terms of resource utilization, such as memory and CPU consumption and the (physical) time to complete a given simulation scenario, and then we explored HydraNetSim's scalability in comparison

[32] Such as PeerSim

with SDES, in different modes - specifically 1Slave, 2Slave, and 3Slave. Importantly, each slave is run on a specific CPU on a different core.

The results show that HydraNetSim-1Slave is no better than SDES and exhibits similar behavior to SDES in terms of memory consumption, (physical) time for completing a given scenario, and scalability. Additionally, SDES takes 41.6% more (physical) time than HydraNetSim-2Slave and 23.6% than HydraNetSim-3Slave. HydraNetSim-2Slave is 1.42 times faster, consumes 1.18 times more memory, and supports 2.02 times more nodes than SDES. Whereas, HydraNetSim-3Slave performs 1.24 times faster, consumes 2.08 times more memory, and supports 3.04 times more nodes than SDES. The scaling factor of HydraNetSim is $\lceil(\beta-1) * 102.04\%\rceil$ of the maximum nodes supported by its sequential counterpart[33].

One reason that HydraNetSim-3Slave performs relatively slower than HydraNetSim-2Slave (although it still performs 1.24 times faster than its sequential counterpart) to complete the simulation scenario of 19,600 nodes can be the inherited property of sequential portion of the KV-BS. This sequential fraction may limit the speedup by running the simulation on multiple processors in parallel, as explained by Amdahl's law. But, we have noticed the limitation of Amdahl's law in addressing the negative effects of overheads associated with parallel execution on expected speedup by using parallel computing. We have analyzed the drastic effects of these overheads on the overall performance of parallel processing and discussed a worst case scenario when the parallel execution may prove even worse than the sequential execution because of these overheads.

Furthermore, we analyzed the relative slow performance of HydraNetSim-3Slave for small problem size, and observed the fact that our synchronization approach is based on highly coupled LPs (in terms of application messages). This means, greater the number of application messages sent in between LPs, the lower the number of block-resume-block cycles, yielding the higher performance for HydraNetSim. However, not all the slaves can have application specific messages for others at all times. So, increasing the number of slaves there is an increase in block-resume-block cycles which relatively slows the relative performance of HydraNetSim-3Slave more than HydraNetSim-2Slave. Another reason can be the smaller value of lookahead, as there may be the situations when LPs may have gone through many null messages rounds before resuming even a smaller set of safe events. This can also lead to a relatively larger number of block-resume-block cycles.

Additionally, we observed that we can gain benefit from HydraNetSim by running larger simulation scenarios on multi-core architecture. As we have seen that, HydraNetSim-3Slave performs relatively better and faster than HydraNetSim-2Slave when we increase the size of simulation scenario to double. One reason that we observed for this fact is the increase in the message density, as we discussed that a higher number of application specific messages results into the higher performance of HydraNetSim. An interesting fact that we noticed is that we can gain benefit from parallel processing even by enlarging the complexity of the given problem to a certain limit which depends upon various factors such as the magnitude of

[33] Where, $\beta$ is the number of slaves.

overheads associated with parallelization, number of processors, computing power of given hardware, ratio of parallel and sequential portions of the given program, etc.

We have also examined the influence of an increase in number of participating nodes and resultantly increase in message density in the simulation model by increasing the number of nodes and concluded that HydraNetSim needs more memory and more physical time[34] to complete a simulation run due to the increased complexity of the simulation model due to increase in number of participating nodes.

## 6.2 Future Work

By and large, in this thesis we have addressed the difficulties concerning fine-grain level details in developing a parallel version of a discrete event simulator that proves to be faster than its sequential counterpart, and have proposed a novel solution (HydraNetSim) for these problems by focusing on exploiting the multi-core architecture modern computing hardware. Furthermore, we have identified that the potential speedup by running the simulation in parallel using multi-core architecture depends upon the inherited property of non-parallelizable fraction of the simulation model. We have identified equally important factors in limiting the performance of parallel execution, i.e., the overheads (such as synchronization, setup, etc.) associated with parallelization. These overheads play an important role in limiting the potential speedup expected by using a multi-core computing hardware. The future research must analyze the role of these overheads in detail. Additionally, we have found that we can gain benefit from existing faster (multi-core) architecture by running larger programs on it. This potential benefit is restricted to a certain limit imposed by different factors (as discussed in section 5.6) for a given scenario. The future research must explore this limit and the influence of underlying factors on this limit. Following paragraphs highlights some important areas to explore for future research.

As future work, first of all we recommend improving the node assignment technique following the space-parallel-partitioning scheme. One suggestion is to consider the geo-location proximity. For example, if we are going to simulate the world as a network topology, then the nodes belonging to a particular country should be assigned to the same LP, in this way the neighbors of a given LP belong in the same continent. This scheme may decrease the coupling factor in between LPs.

Another important consideration for the future work is to increase the virtual memory size of the sequential counter part of the HydraNetSim, so it can support more simulated nodes. Then, the behavior of HydraNetSim (with x-Slave mode) must be analyzed on comparison with SDES for very large scale (in terms of simulated nodes) simulation scenario.

Equally important area of improvement for future work is to find a better scheme for determining the value of EOT, because the current scheme causes all the LPs to simulate at

[34] In comparison with the time taken by the same mode (2Slave or 3Slave) to execute a scenario of smaller scale.

the pace of the slowest LP. Furthermore, this work should also suggest a workaround for the problem of inadequate lookahead as discussed in the section 5.6. One suggestion is to set

$$EOT = currentSimTime + L$$

for both (block and resume) modes, and then each slave should report its EOT to the master only rather than reporting every other slave. The master will then determine the lowest EOT of the system and will broadcast this information to all slaves. To avoid the zero-lookahead cycles and the simulation at the pace of the slowest LP, we can use the conditional events in such a way that while reporting its EOT to the master each LP will also inform the master about its state (i.e., block or resume), and the timestamp of the next scheduled event. This additional information will help in identifying the minimum of the estimated minimum timestamp of next event in the system and can then be utilized to inform the LPs to execute the events having timestamp less than that estimated next event. However, the implications of this proposed scheme must be studied in more detail.

Future work must also include visualization and/or GUI support for topology representation, visualization of the simulation through graphs, and customization of HydraNetSim (such as setting the path to the DB, exporting simulation results into a customize form (such as bar-chart, histograms, etc), selection of number of slaves, etc.).

The coupling factor between LPs must also be explored so that the LPs which have significantly few application messages for each other can resume their simulation without waiting for null messages (so frequently) from each other. This is very important for improving the simulation speed of HydraNetSim in greater than 2Slave mode.

Furthermore, the effects of all the factors (as described in equation number 5.1) must be explored in detail to understand their effect upon the performance of HydraNetSim. The dependency of these factors on each other must also be examined. For example, how do the fluctuations in $\lambda$ affect the PH and how does the variation of n affect PH. Furthermore, the relationship between event density and $\lambda$ must be examined and then their effects on PH must be studied. The degree of proportionality of each factor must also be examined individually to identify the most influential factor. Lookahead is a very important parameter of the proposed synchronization algorithm. The effects on PH of lookahead in relation to $\lambda$ and n must be examined. This future study must also analyze the effects of increasing the value of lookahead on the overall performance of HydraNetSim by keeping all other factors constant. These studies will be useful for improving the overall performance of HydraNetSim at a larger scale.

# References

[1] Gregory R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison Wesley, 2000

[2] George Coulouris, Jean Dollimore and Tim Kindberg, *Distributed Systems: Concepts and Design*, 3rd edition, Pearson Education ltd. , 2001 ISBN 0201-619-180

[3] Saranya Saetang, Distributed System, Peer-to-Peer and Privacy Policy, Slide presentation for a seminar, HCI Research Group, Department of Computer Science University of Bath, 17 January 2006. [Online]. Available:
http://www.cs.bath.ac.uk/~hci/HCI-Seminars/Ay_06.pdf
[Accessed on 2011-04-16]

[4] Wehrle Klaus, Günes Mesut, and Gross James, *Modeling and Tools for Network Simulation*, 1st edition Springer, 2010

[5] Jerry Banks, John S. Carson II, Barry L. Nelson, and David M. Nicol, *Discrete-Event System Simulation*, Prentice Hall, 4th edition, 2005

[6] Averill M. Law, *Simulation Modeling and Analysis*, McGrawHill, 4th edition 2007

[7] Roger Mchaney, *Understanding computer simulation*, Roger Mchaney and Ventus Publishing ApS 2009

[8] Wouter Duivesteijn, Continuous Simulation, Slides for a presentation in a course on Simulation, Department of Information and Computing Sciences, Faculty of Science, Utrecht University, 6 June 2006. [Online]. Available:
http://www.cs.uu.nl/docs/vakken/sim/continuous.pdf
[Accessed on 2011-04-17]

[9] Donald C. Craig, "Extensible Hierarchical Object-Oriented Logic Simulation with an Adaptable Graphical User Interface", Department of Computer Science, School of Graduate Studies, Memorial University of Newfoundland, St. John's Newfoundland, Canada. [Online]. Available:
http://web.cs.mun.ca/~donald/msc
[Accessed on 2011-04-17]

[10] Avrill M. Law and W. David Kelton, Simulation Modeling and Analysis, 3rd Edition, McGraw-Hill, Boston, MA. , 2000

[11] “Monte Carlo simulation”, Palisade Corporation, [Online]. Available:
http://www.palisade.com/risk/monte_carlo_simulation.asp
[Accessed on 2012-02-10]

[12] Alex F. Bielajew, “Monte Carlo Modeling in External Electron-Beam Radiotherapy - Why Leave it to Chance?”, *Proc. 11th Conference on the Use of Computers in Radiotherapy*, 1994

[13] Avrill M. Law and W. David Kelton, *Simulation Modeling and Analysis*, 3rd Edition, McGraw-Hill, Boston, MA. , 2000, Figure 1.1

[14] Alberto Montresor and Márk Jelasity, “PeerSim: A scalable P2P simulator”, *presented at the IEEE Ninth International Conference on Peer-to-Peer Computing*, 2009. P2P ’09, Seattle, WA, USA, 2009, pp. 99–100, DOI:10.1109/P2P.2009.5284506, [Online]. Available:
http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5284506

[15] Wehrle Klaus, Günes Mesut, and Gross James, *Modeling and Tools for Network Simulation*, 1st edition Springer, 2010, fig 1.2

[16] Bernard P. Zeigler, Herbert Praehofer and Tag G. Kim, *Theory of Modeling and Simulation*, 2nd edition, Academic Press, January 2000

[17] Richard M. Fujimoto, "Parallel Discrete Event Simulation", Communication of the ACM, vol. 33 Issue 10, Oct. 1990

[18] Richard M. Fujimoto, Slides for tutorial: Parallel & Distributed Simulation Systems: From Chandy/Misra to the High Level Architecture and Beyond, "Slide # 5: Reasons to use Parallel / Distributed simulation", College of Computing Georgia Institute of Technology Atlanta, 17 June 2008, [Online]. Available: http://www.slidefinder.net/t/tutorial_parallel_distributed/simulation_systems_chandy_misra/202693 [Accessed on 2012-02-10]

[19] Tran, Van Hoai, Slides parallel Discrete Event Simulation, Department of Computer Science and Engineering, HCMC University of Technology, 15 October 2007, [Online]. Available: http://www.cse.hcmut.edu.vn/~hoai/download/parallel-computing/2010-2011/talk_parallel-discrete-simulation.pdf,
[Accessed on 2012-02-10]

[20] B. W. Hollocks, "Forty years of discrete-event simulation—a personal reflection", *Journal of the Operational Research Society* volume 57, pp. 1383–1399, 2006

[21] Cheng-Hong Li, Alfred J. Park, and Eugen Schenfeld, "Analytical Performance Modeling for Null Message-Based Parallel Discrete Event Simulation", in *19th Annual IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 349-358, 2011

[22] H. Ringberg, M. Roughan, and J. Rexford, *"The need for simulation in evaluating anomaly detectors"*, SIGCOMM Computer Communication Review, 2008

[23] Stephan Hartmann, "The world as a process: Simulations in the natural and social sciences", In: *Modelling and simulation in the social sciences from a philosophy of science point of view* (eds. R. Hegselmann, U. Mueller and K. G. Troitzsch) Kluwer: Dordrecht. pp. 77-100, 1996

[24] Steven Strogatz, "The end of insight", In: *What is Your Dangerous Idea? Today's Leading Thinkers on the Unthinkable* (eds. J. Brockman, H. Perennial) New York, 2007, pp.130-131.

[25] Jonathan Gabbai, Areas of flight simulator applications, The Art of Flight Simulation [Internet]. Version 6. Knol. 29 Jul 2008. [Online]. Available: http://knol.google.com/k/jonathan-gabbai/the-art-of-flight-simulation/1brpsoyxacwpz/2, accessed on 2011-04-21

[26] Ahmet Y. Şekercioğlu, András Varga, and Gregory K. Egan, "Parallel Simulation made easy with OMNeT++". In *Proc. of European Simulation Symposium*, Delft, Netherlands, 2003

[27] Rajive L. Bagrodia, *"Perils and Pitfalls of Parallel Discrete-Event Simulation"*, in *Proc. of the 1996 Winter Simulation Conference*, IEEE 1996

[28] Larry Soule and Anoop Gupta, "An Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation", *ACM Transactions on Modeling and Computer Simulation*, Vol.1 No. 4, October 1991, pp. 308-347

[29] Gilbert Chen and Boleslaw K. Szymanski. "DSIM: Scaling Time Warp to 1,033 processors", In *Proceedings of the 37th Winter Simulation Conference*, pp. 346–355, 2005

[30] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse, "JiST: An Efficient Approach to Simulation using Virtual Machines", *Software Practice & Experience*, 35(6):539–576, 2005

[31] Benno Overeinder, Bob Hertzberger, and Peter Sloot, “Parallel discrete event simulation*”*, Third workshop on design and realization of computer systems, Eindhoven, pp 19-30, ISBN 90-6144-995-2, May 1991

[32] Von-Yee Vee and Wen-Jing Hsu, “Parallel Discrete Event Simulation: A Survey”, Technical report, Centre for Advanced Information Systems, Nanyang Technological University, Singapore, 1999

[33] Gilbert Chen and Boleslaw K. Szymanski, “Lookahead, Rollback and Lookback Searching for Parallelism in Discrete Event Simulation” , *in Proc. Summer Computer Simulation Conference*, 2002

[34] Lisa M. Sokol, Duke P. Briscoe, and Alexis P. Wieland, “MTW: A strategy for scheduling discrete simulation events for concurrent execution”, *in Proc. Summer Computer Simulation Conference, 2002*

[35] Richard M. Fujimoto, “Time warp on a shared memory multiprocessor”, *Transactions Society for Computer Simulation*, Volume 6 Issue 3, Jul. 1989

[36] A. Prakash and R. Subramanian, “Filter: An algorithm for reducing cascaded rollbacks in optimistic distributed simulations”, *In Proc. 24th Annual Simulation Symposium*, pp. 123-132, 1991

[37] Richard M. Fujimoto, Slides for tutorial: Parallel & Distributed Simulation Systems: From Chandy/Misra to the High Level Architecture and Beyond, College of Computing Georgia Institute of Technology Atlanta, 17 June 2008, [Online]. Available: http://www.slidefinder.net/t/tutorial_parallel_distributed/simulation_systems_chandy_misra/202693/p2, accessed on 2011-27-04

[38] Rimon Barr, Haas J. Zygmunt, and Robbert van Renesse, “JiST: Embedding Simulation Time into a Virtual Machine”, In *Proc. of EuroSim Congress on Modelling and Simulation*, 2004

[39] J. Misra and K. M. Chandy, *“*Distributed Simulation: A Case Study in Design and Verification of Distributed Programs*”, IEEE Transactions on Software Engineering*, SE-5(5):440–452, 1978

[40] Jayadev Misra, “Distributed discrete-event simulation”*, ACM Computing Surveys*, 18(1):39- 65, 1986

[41] K. Mani Chandy and Jayadev Misra, “Asynchronous distributed simulation via a sequence of parallel computations”, *Communications of the ACM*, 24(11):198- 205, 1981

[42] L. Z. Liu and C. Tropper, *“Local deadlock detection in distributed simulations”,* Proceedings of the SCS Multi-conference on Distributed Simulation, 22(1):64 - 69, 1990

[43] Von-Yee Vee and Wen-Jing Hsu, “Parallel Discrete Event Simulation: A Survey”, Technical report, Centre for Advanced Information Systems, Nanyang Technological University, Singapore, 1999, pp. 13, Table 1

[44] Los Alamos National Laboratory, "Largest Computational Biology Simulation Mimics Life's Most Essential Nanomachine." ScienceDaily, 1 Nov. 2005. [Online]. Available: http://www.sciencedaily.com/releases/2005/11/051101223046.htm, accessed on 2011-04-28

[45] Leslie Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System” ,*Communication of the ACM* , 21(7): 558-565(1978)

[46] Wesley W. Terpstra, Christof Leng, Max Lehn, Alejandro P. Buchmann, “Channel-based Unidirectional Stream Protocol (CUSP)”, *Proceedings of the IEEE, INFOCOM Mini Conference,* March 2010

[47] B. Ford, “Structured Streams: a New Transport Abstraction”, in *SIGCOMM*, 2007

[48] R. Stewart, “Stream Control Transmission Protocol”, *IETF RFC 4960*, September 2007

[49] "Channel-based Unidirectional Stream Protocol (CUSP)", Database and Distribute Systems Research Projects, DVS: Technische Universität Darmstadt. [Online]. Available: http://www.dvs.tu-darmstadt.de/research/cusp/, accessed on 2012-01-26

[50] W. W. Terpstra, J. Kangasharju, C. Leng and A. P. Buchmann, "BubbleStorm: Resilient, Probabilistic, and Exhaustive Peer-to-Peer Search." *In Procs. Of SIGCOMM*, 2007, pp. 49–60. ACM Press, New York (2007)

[51] P. Maymounkov and D. Mazi`eres, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric." *In Procs. of IPTPS*, 2001

[52] M. Lehn, T. Triebel, C. Leng, A. Buchmann and W. Effelsberg, "Performance Evaluation of Peer-to-Peer Gaming Overlays." *In Procs. of P2P*, 2010

[53] C. Leng, M. Lehn, R. Rehner and A. Buchmann, "Designing a Testbed for Large-scale Distributed Systems", *in Proc. of ACM SIGCOMM'11*, ACM, August 2011

[54] MATLAB version 7.10.0 (R2010a). Natick, Massachusetts: The MathWorks Inc., 2010.

[55] B.D. Lubachevsky, "Efficient distributed event-driven simulations of multiple-loop networks" *Communications of the ACM*, Volume 32 Issue1, January, 1989

[56] "BuubleStorm", Database and Distribute Systems Research Projects, DVS: Technische Universität Darmstadt. [Online]. Available: http://www.dvs.tu-darmstadt.de/research/bubblestorm/, accessed on 2012-05-07

[57] C. Leng, and W. W. Terpstra, "Distributed SQL Queries with BubbleStorm", in *Active Data Management to Event-Based Systems and More* (Lecture Notes in Computer Science 6462), 1st ed. K. Sachs, I. Petrov, and P Guerrero, Eds. Springer, November 2010, ISBN 978-3-642-17225-0

[58] W. W. Terpstra, C. Leng and A. P. Buchmann, "BubbleStorm: Analysis of Probabilistic Exhaustive Search in a Heterogeneous Peer-to-Peer System", Dept. Fach. Info. Tech. Univ. Darmstadt, Germany, Tech. Rep. TUD-CS-2007-2, May 2007

[59] Intel® Core™ i5-2410M Processor, Intel Corporation, [Online]. Available: http://ark.intel.com/products/52224, accessed on 2012-05-26

[60] S. Gundavelli, K. Leung, V. Devarapalli, K. Chowdhury, and B. Patil. Proxy Mobile IPv6. RFC 5213, IETF, August 2008.

[61] B. Cohen, Incentives build robustness in bittorrent. In Proceedings of the Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, 2003.

[62] Crespo, A., Garcia-Molina, H.: Semantic overlay networks for P2P systems. In: Moro, G., Bergamaschi, S., Aberer, K. (eds.) AP2PC 2004. LNCS (LNAI), vol. 3601, pp. 1–13. Springer, Heidelberg (2005)

[63] Uichin Lee, Min Choi, Junghoo Cho, M. Y. Sanadidi, and Mario Gerla. Understanding pollution dynamics in p2p file sharing. In 5th International Workshop on Peer-toPeer Systems (IPTPS'06), 2006.

[64] Martin Quinson, Cristian Rosa, Christophe Thiéry, "Parallel Simulation of Peer-to-Peer Systems" ccgrid, pp.668-675, 2012 *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (ccgrid 2012), 2012, ISBN: 978-0-7695-4691-9, DOI: 10.1109/CCGrid.2012.115

[65] R. Bhardwaj, V. Dixit, and A. K. Upadhyay. "An Overview on Tools for Peer to Peer Network Simulation" in *International Journal of Computer Applications*, 1(1):70–76, Feb. 2010.

[66] H. Xu, S. Wang, R. Wang, and P. Tan, “A Survey of Peer-to-Peer Simulators and Simulation Technology” in *JCIT: Journal of Convergence Information Technology*, 6(5):260–272, May 2011.

[67] A. B. Stirling and M. K. Stirling, “Tools for Peer to Peer Network Simulation”, IRTF P2pRG

[68] S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai, “A Survey of Peer-to-Peer Network Simulators,” *Proceedings of The Seventh Annual Postgraduate Symp.*, Liverpool, UK, 2006

[69] I. Baumgart, B. Heep, and S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework", In Proceedings of the 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA, pp. 79-84, 2007

[70] OMNet++, OMNeT++ Community, ] [Online]. Available: http://www.omnetpp.org/, accessed on 2012-05-22

[71] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: a scalable peer-to peer lookup protocol for internet applications,” IEEE/ACMTransactions on Networking, vol. 11, no. 1, pp. 17–32, Feb.2003.

[72] G. F. Riley, R. M. Fujimoto, and M. H. Ammar. “A Generic Framework for Parallelization of Network Simulations”, In *Proc. of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1999.

[73] The Network Simulator - ns-2, ns, [Online]. Available: http://www.isi.edu/nsnam/ns/, accessed on 2012-05-22

[74] User Information, ns-2, [Online]. Available: http://nsnam.isi.edu/nsnam/index.php/User_Information, accessed on 2012-05-22

[75] Welcome, TCL Developer Xchange, Tcl community, [Online]. Available: http://www.tcl.tk/, accessed on 2012-05-22

[76] Object-Oriented Programming in Tcl, TCL Developer Xchange, Tcl community, [Online]. Available: http://www.tcl.tk/about/oo.html, accessed on 2012-05-22

[77] PDNS - Parallel/Distributed NS, [Online]. Available: http://www.cc.gatech.edu/computing/compass/pdns/, accessed on 2012-05-22

[78] Nam: Network Animator, Tim Buchheim, [Online]. Available: http://www.isi.edu/nsnam/nam/, accessed on 2012-05-22

[79] J. Pujol-Ahull´o, P. Garc´ıa-L´opez, M. S`anchez-Artigas, and M. Arrufat-Arias. “An extensible simulation tool for overlay networks and services”, In *SAC ’09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 2072–2076, New York, NY, USA, 2009 ACM.

[80] G. Chen and B. K. Szymanski. “DSIM: Scaling Time Warp to 1,033 processors” In *Proc. of the 37th Winter Simulation Conference*, pp 346–355, 2005.

[81] DSIM: A Distributed Optimistic Parallel Discrete Event Simulator, Gilbert Chen and Boleslaw Szymanski, Center for Pervasive Computing and Networking, Rensselaer Polytechnic Institute, [Online]. Available: http://assassin.cs.rpi.edu/~cheng3/dsim/, accessed on 2012-05-22

[82] András Varga, Ahmet Y. Şekercioğlu, and Gregory K. Egan, “A Practical Efficiency Criterion for the Null-Message-Algorithm” In *Proc. of European Simulation Symposium*, Delft, The Netherlands, 2003.

[83] Intel® Core™ i7-920 Processor, Intel corporation, [Online]. Available: http://ark.intel.com/products/37147/Intel-Core-i7-920-Processor-(8M-Cache-2_66-GHz-4_80-GTs-Intel-QPI), accessed on 2012-05-26

[84] S. Jafer, “PARALLEL SIMULATION TECHNIQUES FOR LARGE-SCALE DISCRETE-EVENT MODELS”, Ph.D. dissertation, Dep. Sys. Comp. Eng., Carleton University, Ottawa, Ontario, Canada, 2011, [Online]. Available: http://www.sce.carleton.ca/~sjafer/PhD_Thesis_Shafagh_V6_Final.pdf, accessed on 2012-05-28

[85] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the April 18-20, 1967, spring joint computer conference (AFIPS '67 (Spring))* ACM, New York, NY, USA 1967, pp 483-485. DOI=10.1145/1465482.1465560

[86] F. P. Brooks Jr., *The Mythical Man-Month*, Addison Wesley Longman, Inc., 1995, ISBN: 0-201-83595-9
R.J Lorimer (2006- 11-08), Performance: Understanding Amdahl’s Law, [Online]. Available: http://www.javalobby.org/java/forums/t84101.html
Accessed on 2012-06-10

[87] Daniel, SVG Graph Illustrating Amdahl's Law, Permission={{cc-by-sa-3.0}} [Online]. Available: http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg
Accessed on 2012-06-12

TRITA-ICT-EX-2012:125