國立交通大學

資訊科學與工程研究所

博士論文

一套適用於無線車輛通訊網路研究的網路與交通模擬器

A Network and Traffic Simulator for Wireless Vehicular Communication Network Research

研 究 生 ：周智良

指導教授 ：王協源 教授

中華民國九十八年七月

一套適用於無線車輛通訊網路研究的網路與交通模擬器

A Network and Traffic Simulator for Wireless Vehicular Communication Network Research

研 究 生：周智良　　Student：Chih-Liang Chou

指導教授：王協源　　Advisor：Shie-Yuan Wang

國 立 交 通 大 學

資訊科學與工程研究所

博士論文

A Dissertation
Submitted to Department of Computer Science
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in

Computer Science

July 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年七月

一套適用於無線車輛通訊網路研究的網路與交通模擬器

學生：周智良　　　　　　　指導教授：王協源教授

國立交通大學資訊工程學系（研究所）博士班

摘　　要

當智慧型運輸系統的概念對全世界展示出其改善交通運輸品質與效率的潛力後，工業界與學術界對於可應用於智慧型運輸系統的技術一直都抱持著高度的興趣。直到現今，智慧型運輸系統已經發展成為一個非常熱門的研究領域，也已經有許多在日常生活中的實際應用，例如常見的車上廣播電視、車上衛星導航、車上電子自動收費系統等等，這些應用不僅為智慧型運輸系統的成功做了見證，相信在未來日子裡，更多相關領域的研究也將陸續地投入去發展更新穎與便利的應用。

智慧型運輸系統的目標，是利用將資訊與通訊科技(例如各式各樣的無線網路技術)加到基礎運輸建設與車輛上，以達到改善車輛與行人安全、增加運輸效率、提供車上娛樂等目的。舉例來說，透過車輛間互相交換自己的地理位置、移動方向與移動速度的訊息來達到碰撞的預警，透過參考及時路況的車輛導航系統來降低行車時間與燃料消耗，透過電子式自動收費系統來增加行車的流暢性，透過數位電視廣播提供給車上乘客各種的視聽娛樂，以及車上的網際網路連線服務等等都是這一類型的應用。

可以肯定的，任何應用在實際部署以提供服務之前，一定得經過某些程度的測試來證實其可行性。而對於測試上述類型的應用，在實際的道路環境中做測試通常是為了獲得可信的測試結果所必須的。然而，這樣的測試有時會需求大量的車輛與人力，有時會需要能穩定控制的車輛駕駛路徑等等，這些特性可能會需要昂貴的花費，或是使測試人員在測試的過程中處於有潛在危險的環境裡，更甚者，由於非測試人員所駕駛的其他車輛並不會配合測試而做出特定的移動方式，某些要求特定車輛相對移動方式的測試在實際的道路環境中是很難準確控制的。有鑑於此，為了消除或減輕以上所提到在測試中所可能遭遇的困難，利用模擬工具便成為一種可以取代或是輔助實地測試的一種選擇。

在此論文中，我們將介紹一套適用於無線車輛通訊網路研究的網路與交通模擬器。此模擬器整合了網路通訊協定的模擬能力與車輛移動行為的模擬能力。前者包括 TCP/IP、UDP/IP、IEEE 802.16e、IEEE 802.11p 等等適用於車輛網路的通訊協定，後者包括車輛於道路網路上的移動行為、車輛對於紅綠燈號誌的反應、車輛的跟車行為與車輛的變更車道行為等等。關於此模擬器的設計、實作、驗證與效能分析，都將呈現於此論文中。

關鍵字：網路模擬器、交通模擬器、智慧型運輸系統。

A Network and Traffic Simulator for Wireless Vehicular Communication Network Research

Student : Chih-Liang Chou　　　Advisor : Shie-Yuan Wang

Department of Computer Science
National Chiao Tung University

ABSTRACT

Since the idea of Intelligent Transportation Systems (ITS) has shown the world its great potential to improve the quality and efficiency of transportation, intelligent transportation technologies have gained the attention from the industry and the academia. Nowadays, the research field of ITS is very popular and it’s believed that the importance of ITS will keep growing up in the future.

The goal of ITS is to apply information and communications technologies to transportation infrastructures and vehicles for improving vehicle/pedestrian safety, increasing transport efficiency, and offering in-vehicle entertainment, such as collision warning, driving guidance, electronic toll collection, digital video broadcasting, Internet access, and so on. The practical usages of these applications in our daily lives have shown us their usefulness and also given us the confidence on the viability of this kind of applications that will be applied in the future.

Certainly, any application must undergo some tests before it is deployed for service. For a vehicular application, conducting field trials in real vehicular environments is usually necessary. However, these trials are sometimes costly, hard-to-control (unrepeatable), dangerous, and even infeasible when many vehicles and people are required to involve in the trials. In this case, using simulation tools before conducting field trials or even to replace field trials is a feasible alternative to alleviate or eliminate the above problems.

This dissertation presents a network and traffic simulator that is useful for wireless vehicular communication network research. This simulator integrates the simulation capabilities of network communication protocols (e.g., TCP/IP, UDP/IP, IEEE 802.16e, IEEE 802.11p, etc.) and vehicular movement behaviors (e.g., moving on road networks, obeying traffic light signals, car following, lane changing, etc.). The design, implementation, validation, and performance of this simulator are presented in this dissertation.

Keywords: network simulator, traffic simulator, intelligent transportation systems (ITS).

Acknowledgments

I would like to thank my advisor, Prof. Shie-Yuan Wang, for his guidance and concern for me during my graduate life. I would also like to thank my dissertation committee members: Prof. Yi-Bing Lin, Prof. Yu-Chee Tseng, Prof. Jang-Ping Sheu, Prof. Yau-Hwang Kuo, and Prof. Chu-Sing Yang. They gave me many valuable comments on my Ph.D. thesis and suggestions about writing system implementation papers.

Also, I want to give my thanks to all my lab-mates. Their care and assistance gave me a lot of courage and happiness during my stay at NSL lab. In particular, I want to thank Chih-Che Lin, from whom I learned a lot of knowledge and received a lot of help. Besides, I want to thank all my good friends who kept encouraging me during my graduate life.

Finally, I want to dedicate my Ph.D. dissertation to my family. Without their support, I could not obtain my Ph.D. degree. I am so lucky and so grateful to have such a good family. I want to give my best wishes and appreciation to all of them.

July 31, 2009

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Vehicles is definitely one of the greatest inventions impacting the human world. Undoubtedly, the transportation contributes a lot on the modernization of human life. Vehicles help shorten traveling time and thus bring a lot of convenience to people's daily lives. For example, people can work in an urban area and live in a suburban area and commute between them every day. Also, people can take just a few hours to travel to vacation resorts located several hundreds of miles away from their home. Moreover, because people can meet with their friends or relatives more easily by vehicles, the emotion sharing among people becomes more frequent and that results in more harmonious human world. From the above examples, we can see how vehicles play an important role in most people's lives. In other words, nowadays vehicles has become a necessary for most people.

Obviously, people's needs on vehicles lead to growing vehicular industry. In the past couple of decades, many efforts have been continually put into the vehicular industry to create more value on vehicular business. The vehicular manufacturers have designed more robust vehicular structures for passenger security, built lighter materials for less fuel consumption, built electricity-power engines for eco-friendly purposes, used electronic driving control assistance systems for vehicular motion stability, installed navigation systems on vehicles for route guidance, and so forth. Some of the above vehicle-related research and manufacturing efforts can be connected to Intelligent Transportation Systems (ITS).

The goal of ITS is to apply information and communications technologies to transportation infrastructures and vehicles. For example, providing instant emergency traffic situations to improve driver/passenger security, installing electronic driving control assistance systems in vehicles to increase the vehicular motion stability, using electronic navigation systems for route guidance to reduce traveling time and fuel consumption, delivering video and audio programs to vehicles to improve passengers' traveling experience, and providing ubiquitous Internet access for passengers while traveling on vehicles are just a small part of ITS applications. The usage of these applications in our daily lives has shown us their usefulness and also given us the confidence for those under-developing applications.

Nowadays, the field of ITS is very popular. Many kinds of technologies are currently used or under development to support ITS applications. One of them is wireless communications. Wireless communications technologies provide the solutions to exchange information among roadside base stations and vehicles moving on the road. For examples, the IEEE 802.11p [1] and IEEE 1609 draft standards [2, 3, 4, 5] are instances of wireless communications technologies. They have been proposed as a networking technology for vehicular environments and under development for providing services in the future. Besides, the IEEE 802.16e [6] is also an alternative to provide wireless communications in vehicular environments.

It is certain that any ITS application must be thoroughly tested and evaluated before being deployed on the roads. This means that many experiments under different parameter settings, configurations, scenarios, and conditions must be performed to verify the feasibility and effectiveness of an application and the used networking technology in the real-life environment. According to the results obtained from experiments, the designs of the application and the used networking technology might need to be revised many times before acceptable performances can be achieved.

A field trial of wireless vehicular communications sometimes involves a large number of vehicles and people (drivers and computer operators) for generating convincing results. However, conducting such field trials is very costly because many vehicles need to be rented (or purchased), many communication equipments need to be purchased, and many experimenters need to be employed for the field trials. Sometimes, during a field trial with a specifically-designed high-speed scenario, the experimenters may even face potential dangers such as collisions with vehicles or pedestrians. Besides, it is very difficult to accurately control and repeat a field trial on the roads, which is bad for debugging the found problems of a new protocol or application. In order to eliminate or alleviate the above problems, a feasible solution is to use software simulation to perform preliminary tests and performance evaluations before conducting field trials. Comparing to conducting field trials, using software simulations is usually cheaper, safer, and easier to control. Besides, based on the simulation results, a set of field trials can be planed effectively to focus on crucial problems or system parameters. This definitely saves a lot of time and of course a lot of cost spent on the field trials.

For studying wireless vehicular communication networks, a software simulator must be able to simulate both communication/network protocols and microscopic vehicular movements. The two requirements are not new and such capabilities are already provided by existing network simulators or traffic simulators, respectively. Regarding network simulators, they are usually used to test the functions and evaluate the performances of network protocols and applications under various network conditions. One can use them to test how his/her protocols (e.g., routing protocols, medium access control protocols, or transport protocols) and applications (e.g., HTTP, FTP, or VoIP) would perform under various network conditions. Regarding traffic simulators, they can be roughly categorized into two kinds: macroscopic and microscopic. A macroscopic traffic simulator ignores each vehicle's mobility detail but provides macroscopic view of traffic flows on the roads. Instead, a microscopic traffic simulator deals with each vehicle's mobility detail on the roads, such as acceleration/deceleration, car following, lane changing, and so on. A traffic simulator is usually used in the research areas of transportation engineering, such as transportation planning and traffic engineering.

In general, a network simulator is dedicated only to the studies of network protocols and applications, while a traffic simulator only to the studies of transportation engineering. However, in order to study advanced ITS applications, a simulation platform must be able to simulate both network and traffic simultaneously. For example, a navigation system may try to collect the immediate road conditions through some wireless medium and guide a driver through a better route for saving traveling time and fuel consumption. To simulate this application in the simulation platform, the moving path of a vehicle may need to be changed after the driver receiving a message from the wireless vehicular communication network. To achieve this, the simulation platform must tightly integrate both network and traffic simulations.

In this dissertation, we present an integrated simulation platform, called NCTUns, that is useful for wireless vehicular communication network research. NCTUns 1.0 [7, 8] was originally developed as a network simulator with unique network simulation capabilities. Later on, NCTUns 4.0 incorporates traffic simulation (e.g., road network construction and microscopic vehicle mobility models) with its existing network simulation. Therefore, the current version of NCTUns [9] has the simulation capabilities of network communication protocols (e.g., TCP/IP, UDP/IP, IEEE 802.16e, IEEE 802.11p, etc.) and vehicular mobility behaviors (e.g., moving on road networks, obeying traffic light signals, car following, lane changing, etc.). It is a useful simulation platform for wireless vehicular communication network research.

The rest of this dissertation is organized as follows. In Chapter 2, we survey related work that combines the capabilities from both a network simulator and a traffic simulator. In Chapter 3, we present the architecture of NCTUns, including its features, components, design and implementation issues to support the functionalities of a network simulator. Based on Chapter 3, we then present the extended architecture of NCTUns to support the traffic simulation in Chapter 4. In Chapter 5, we validate the mobility control of vehicles on NCTUns with mathematical models based on Newton's laws of motion. In Chapter 6, the scalability performances of NCTUns are evaluated. In Chapter 7, we present future work that we think can be pursued further. Finally, the conclusion is presented in Chpater 8.

Chapter 2

Related Work

As stated in Chapter 1, a simulator suitable for conducting wireless vehicular communication networks research should have the capabilities supported by both a traffic simulator and a network simulator. An intuitive method to construct such a simulator is to write a middleware to combine an existing network simulator with an existing traffic simulator to provide the required capabilities. Such a method is called the "federated approach" in this dissertation. This approach has the advantage that one need not spend time and effort on developing both a new network simulator and a new traffic simulator. However, because the chosen two simulators may have different design architectures, in such a case it is usually difficult or even impossible to combine them with an efficient interaction between them. Another method, called the "integrated approach," tries to add network simulation functions into an existing traffic simulator, or add microscopic vehicle movement simulation functions into an existing network simulator, or develop a new simulator from scratch with all of these capabilities. Using this approach, except the last choice, one need not develop both a network simulator and a traffic simulator from scratch. It saves time and effort by taking advantage of the existing products. In addition, the program code of the traffic and network simulation subsystems are tightly integrated as a single program. The interaction between them is usually more efficient.

Regarding the federated approach, several existing network simulators (e.g., [10],

Table 2.1: Microscopic mobility and other features of traffic/network simulators

Traffic and Network Simulator	Approach	Microscopic Mobility		Other Features	
		Car Following	Lane Changing	Radio Obstacle	Visualization Tool
CORSIM and QualNet [19]	federated	yes	yes	no	yes
VISSIM and ns2 [20]	federated	yes	yes	no	yes
CARISMA and ns2 [21]	federated	yes	no	yes	yes
SUMO and ns2 (TraNS) [22]	federated	yes	no	no	yes
NCTUns	integrated	yes	yes	yes	yes
SWANS [23]	integrated	yes	yes	no	no
AutoMesh [24]	integrated	yes	no	yes	yes
MoVES [25]	integrated	yes	no	no	yes
VANET [26]	integrated	yes	yes	no	yes

[11], [12], and [13]) and traffic simulators (e.g., [14], [15], [16], [17], and [18]) are available alternatives from which one can choose to combine. Some of them are commercial products while some of them are free and/or open source software. In Table 2.1, we list four existing federated traffic/network simulator combinations: CORSIM/QualNet [19], VISSIM/ns2 [20], CARISMA/ns2 [21], and SUMO/ns2 (TraNS) [22]. Among the listed network and traffic simulators, only SUMO and ns2 are free open source software. This means that only the SUMO/ns2 (TraNS) combination is totally open source and allows researchers to freely modify for their research.

The conceptual architecture of a federated traffic/network simulator is shown in Figure 2.1. This approach uses a middleware to interconnect a traffic simulator and

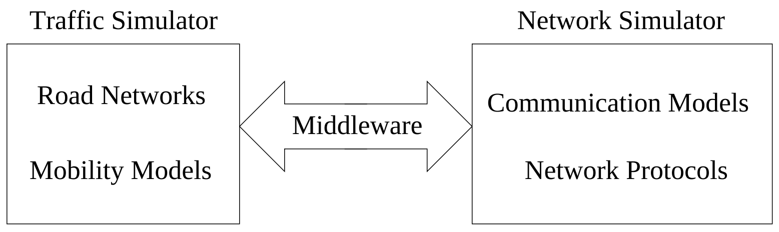
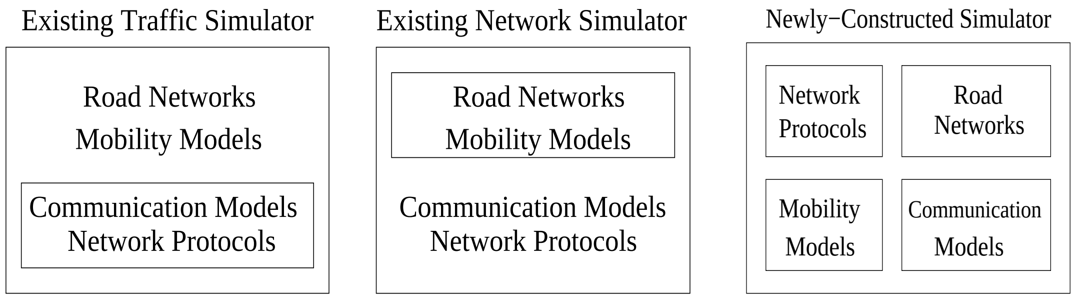


Figure 2.1: The conceptual architecture of a federated traffic/network simulator

a network simulator. The middleware provides bidirectional links, usually realized by TCP connections, between the two simulators. Because the traffic simulator is responsible for simulating the road network and vehicle mobility, the latest position of every vehicle is sent from the traffic simulator to the network simulator during simulation. The position update is performed periodically or by the request from the network simulator. In the network simulator, the vehicular position changing affects the calculation of wireless communication models. Based on the calculated result, the network simulator determines whether a network message sent from a vehicle will be successfully received at another vehicle or not. Then, the message transmission/reception result affects the operations of network protocols. This is basically what a network simulator does. In the network simulator, if a vehicle receives a message and this message makes it decide to change its driving behavior (e.g., to change the current route to avoid congested or dangerous area or to stop immediately to avoid a forthcoming collision, etc.), the network simulator sends a request to the traffic simulator asking for such a change. Usually, a vehicular agent is run within the network simulator to receive the message, analyze the information contained in the message, and issue such a request on behalf of the virtual driver driving in a particular vehicle.

The architectural advantage of the federated approach is that in theory it can combine any traffic simulator with any network simulator. However, in practice it is difficult to do so (if not totally impossible). Two independent simulators may have different designs, implementations, and definitions. For example, the used coordinate systems and the representations of continuous vehicular movement may be different. A transformation mechanism for accommodating these differences must exist in the middleware at the cost of performance degradation during simulation. Besides, a commercial software normally does not release its source code but only exports some pre-defined application program interfaces (API) for external software programs to use. Since the development of new ITS applications advances so quickly, the pre-defined API's may not meet the demands of new ITS applications.

Regarding the integrated approach, Table 2.1 lists five integrated traffic/network simulators: NCTUns, SWANS [23], AutoMesh [24], MoVES [25], and VANET [26]. This approach works only when the used simulator is open source. Three different methods are possible for constructing a simulator using the integrated approach and they are shown in Figure 2.2. In Figure 2.2a, communication models and network protocols are added into an existing traffic simulator. Because a plenty of communication models and network protocols exist in the real life and they can be very complicated (e.g., IEEE 802.11p [1], IEEE 802.16e [6], etc.), this method usually takes a huge amount of time and effort. Therefore, our survey found no integrated simulator adopting this method. In contrast, in Figure 2.2b, an existing network simulator is extended to include the simulations of road networks and vehicle mobility models. This method is more feasible and incurs less cost because a network simulator already has the capability to simulate the mobile node movement (e.g., the commonly used random waypoint mobility model). Based on this capability, it just needs to support road network simulation and apply vehicle mobility models on the mobile node movement. Relatively, this task is easier to accomplish than simulating various complicated communication models and network protocols. Therefore, NCTUns, SWANS, and AutoMesh adopt this method to take advantage of their existing network simulation capabilities. Yet another method is to develop all required components from scratch to construct a new simulator, which is depicted in Figure 2.2c. MoVES and VANET adopt this method. Conceivably, this method will require a very huge amount of time and effort to develop a complete simulator from scratch. It is also the most time-consuming approach to developing an integrated traffic/network simulator.



(a) Adding communication models and network protocols into a traffic simulator

(b) Adding road networks and mobility models into a network simulator

(c) Developing all required components from scratch

Figure 2.2: Three different approaches to constructing an integrated traffic/network simulator

Table 2.1 also compares the microscopic mobility and other important features of each simulator. These features include car following, lane changing, radio obstacles, and the visualization tool. The car-following capability is supported by all simulators listed in Table 2.1. To support this, a road network topology is necessary for vehicles to move on it. Besides, a vehicle's moving speed is restricted by the vehicle (if any) moving in front of it. With this capability, the simulated vehicular moving paths are more realistic than those generated by the random waypoint mobility model. Regarding the convincingness of simulation results, the random waypoint mobility model commonly used before is unsuitable when running an ITS-related simulation [27]. Thus, the car-following capability is very fundamental for a network/traffic simulator.

To have the lane-changing capability, a platform must support multi-lane roads and lane-changing driving logic in the vehicle mobility model. Table 2.1 shows that some platforms do not have this capability. Lacking this capability will reduce the applications of vehicle mobility simulation. For example, without this capability, if a vehicle breaks down on a multi-lane road, the vehicles behind it can only stop because no lane-changing move can be taken. However, in the real life, they can change lanes to move around this broken vehicle. In addition, lacking the lane-changing capability means that no overtaking will occur during simulation. Therefore, the topology of the vehicular ad hoc network (VANET) formed on the roads will not change much during simulation. Since this does not reflect the situations in the real life, the simulation results of network protocol studies on such a simplified and unrealistic VANET may be misleading.

As for radio obstacles, they are capable of totally blocking the transmission of wireless signal or reducing the power of wireless signal. In the real world, vehicles are usually moving on roads surrounded by buildings. The buildings are radio obstacles that influence the wireless signal transmission. When one wants to construct a vehicular network on a simulation platform where more realistic wireless signal transmission among vehicles can be supported, the radio obstacles are needed. Thus, for a network simulator, supporting radio obstacles is an important capability. Table 2.1 shows that only NCTUns, CARISMA/ns2, and AutoMesh have this capability.

The last capability compared is the support of a visualization tool. It is a graphical user interface (GUI) program displaying the road networks and vehicular movement during simulation or after a simulation is finished. This tool provides visual observations of a simulated network. Moreover, some visualization tools also provide the capabilities of interacting with the simulated network during simulation, such as dynamically changing some system parameters of the simulated network or dynamically controlling vehicular mobility. A visualization tool is very useful in specifying, controlling, and observing the simulated network. Table 2.1 shows that every platform has a visualization tool except SWANS.

Chapter 3

Architecture of NCTUns

As stated in Chapter 1, NCTUns was originally developed as a network simulator and then extended with traffic simulation capabilities. In this chapter, we present the original architecture of NCTUns, including its features, components, design and implementation issues. The extended architecture of NCTUns to support the traffic simulation is left to Chapter 4.

3.1 Introduction to NCTUns

Network simulators implemented in software are useful tools for researchers to develop, test, and diagnose network protocols. Simulation is economical because it can carry out experiments without the actual hardware. Also, simulation is flexible because it can study the performances of a system under various conditions. Moreover, simulation results are easier to analyze than experimental results because they are repeatable.

Developing a useful network simulator requires much time and efforts. A network simulator needs to simulate the hardware characteristics of networking devices (e.g., hub or switch), the protocol stacks employed in these devices (e.g., the learning bridge protocol used in a switch), and the network traffic flows (e.g., TCP/IP or UDP/IP connections). It also needs to provide utilities for configuring network topologies, specifying network parameters, monitoring traffic flows, gathering statistics about a simulated network, and so forth.

To save developing time and efforts, some traditional network simulators only simulate real-life network protocols with limited details. For example, OPNET [12] uses a simplified finite state machine model to model complex TCP protocol processing. As another example, ns2 [10] simulates no dynamic receiver advertised window on TCP connections. The simplified simulation design may lead to inaccurate simulation results. In addition, these traditional simulators usually do not support the standard API's that are supported by operating systems (e.g., the UNIX POSIX socket API). Thus, existing and developing real-life application programs cannot run directly on these simulators. Instead, they must be rewritten to use the internal API provided by these simulators (if any). It is inconvenient for users to use these simulators.

To avoid the inaccurate results and inconvenient application-developing environments, the authors in [28, 29] proposed a kernel re-entering simulation methodology and used it to develop Harvard network simulator [30]. Based on the simulation methodology, the NCTUns network simulator [7, 8, 9] was developed as the successor of Harvard network simulator. Using the kernel re-entering simulation methodology, NCTUns directly uses the real-life TCP/IP and UDP/IP protocol stacks to generate more realistic simulation results. Besides, real-life application programs can run directly on NCTUns because NCTUns supports the standard API provided by the Linux Fedora operating systems. Moreover, NCTUns improves the simulation capabilities that Harvard network simulator does not support, such as various media access control protocols and wireless communication/channel models.

3.1.1 Major Components

NCTUns uses a distributed architecture to support the remote simulation and the construction of a simulation service center. Functionally, it can be divided into eight separate components. In Figure 3.1, each component is depicted and classified according to where it is run in the operating system – the user space or the kernel

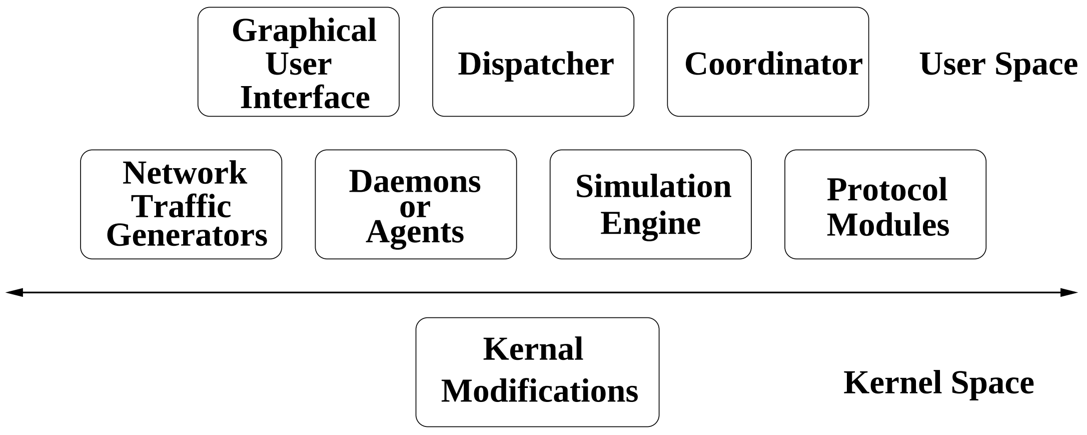


Figure 3.1: The eight components of NCTUns

space. Each component's functionalities are described below.

- **Graphical User Interface (GUI)**

  The GUI is a user-space program. It provides the utilities to edit a network topology, configure the protocol modules used inside a network node, specify mobile nodes' moving paths, plot performance curves, display packet transfer animations, etc. Instead of doing lots of editing work on lots of configuration files, users can easily complete the setting of a simulated network by the user-friendly utilities. Based on the setting, the GUI automatically generates the required configuration files. This not only saves lots of time but also avoids typos that usually occur when a user has to edit all of the configuration files. During simulation, users can change/query some system parameters through the GUI, such as some network interface's maximum/current output queue length. When a simulation is finished, the simulation results can be displayed by some utilities provided by the GUI. For example, the GUI can record each packet's transmission/reception into a binary-format log file during simulation. When the simulation is done, the GUI can display the animation of packet transmission/reception according to the log file, or it can translate the binary-format file into an ascii-format file so that a user can check the log using a text editor. Another example is that the changing of a network interface's output queue length over time can be drawn on a two-dimension graph. Besides, the graph can be captured into a file for further use.

- **Simulation Engine**

  The simulation engine is a user-space program. It functions like a small operating system. Through a set of defined API's, it provides useful and basic simulation services to protocol modules. Such services include virtual clock maintenance, timer management, event scheduling, etc. The simulation engine needs to be compiled with various protocol modules to form the "simulation server." When executed to service a job, the simulation server takes a configuration file suite as its input, runs the simulation, and generates log files and statistics as its output. Because the simulation server uses the kernel's tunnel interfaces that can not be shared with other processes during simulation, no more than one simulation server can run concurrently on a single machine.

- **Protocol Modules**

  A protocol module is like a layer of a protocol stack. It performs a specific protocol or function. For example, the address resolution protocol (ARP) or a first-in-first-out (FIFO) queuing mechanism is implemented as a protocol module. A protocol module is not an independent user-space program. Instead, It needs to be compiled with the simulation engine to form the simulation server. Usually, inside the simulation server, multiple protocol modules are linked into a chain to function as a protocol stack.

- **Dispatcher**

  The dispatcher is a user-space program responsible for simulation job management. It should be executed and remain alive all the time to manage multiple simulation machines on each of which a simulation server is run. The dispatcher is the key role to support the NCTUns distributed architecture. Like that shown in Figure 3.2, the dispatcher can operate between a large number

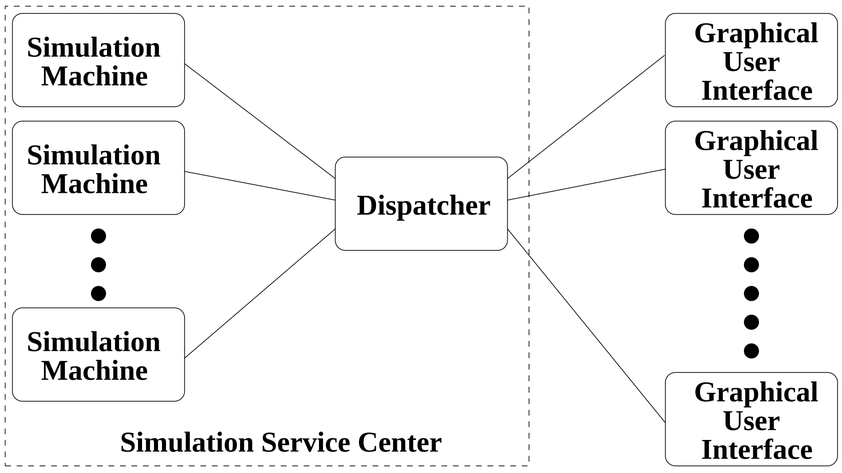


Figure 3.2: The distributed architecture of NCTUns

of GUI users and a large number of simulation machines. It behaves like the manager of a simulation service center. When a GUI user submits a simulation job to the dispatcher, the dispatcher will select an available simulation machine to service this job. If all simulation machines are busy on servicing at this time, the submitted job can be queued in the dispatcher waiting for a simulation machine to become available. Of course, in the case that only one GUI user exists and one simulation server is required, all the NCTUns components can run on a single machine.

- **Coordinator**

  The coordinator is a user-space program. It operates like the agent of the simulation server and deals with the coordination among the GUI program, the dispatcher, and the simulation server. On every simulation machine, a coordinator program needs to be executed and remain alive. Its main task is to let the dispatcher know whether the simulation machine is currently busy running a simulation or not. When executed, it immediately registers itself with the dispatcher to join the dispatcher's simulation service center. Later on, when its status (idle or busy) changes, it will notify the dispatcher of its new status. This enables the dispatcher to always choose an available simulation machine from its simulation service center to service a job. When the coordinator receives a job, it forks (executes) a simulation server to simulate the specified network and protocols. When the simulation server is running, the coordinator communicates with the dispatcher and the GUI program on behalf of the simulation server. For example, periodically the simulation server sends the current virtual time of the simulated network to the coordinator. The coordinator then forwards this information to the GUI program. This enables the GUI user to know the progress of the simulation.

- **Daemons or Agents**

  A daemon or an agent runs at the user space to provide some specific service. For example, the routing information protocol (RIP) routing daemons, as well as the open shortest path first interior gateway protocol (OSPF) routing daemons, exchange routing messages and set up system routing table during simulation. Besides, the home agents and foreign agents are executed to provide the mobile IP service.

- **Traffic Generators**

  A traffic generator runs at the user space to generate TCP or UDP network packets into the simulated network. Unlike a daemon or an agent, a traffic generator does not provide a specific service but just generates background network traffic. For one pair of traffic generators provided by NCTUns, named stg and rtg, they can be configured to produce network packets based on a specified packet transmission pattern. Moreover, they can generate packets following a real-life packet log to produce more realistic background traffic.

- **Kernel Modifications**

  The last component is the modifications that need to be made to the kernel so that a simulation server can correctly run on it. For example, during a simulation, the timers of TCP connections used in the simulated network need

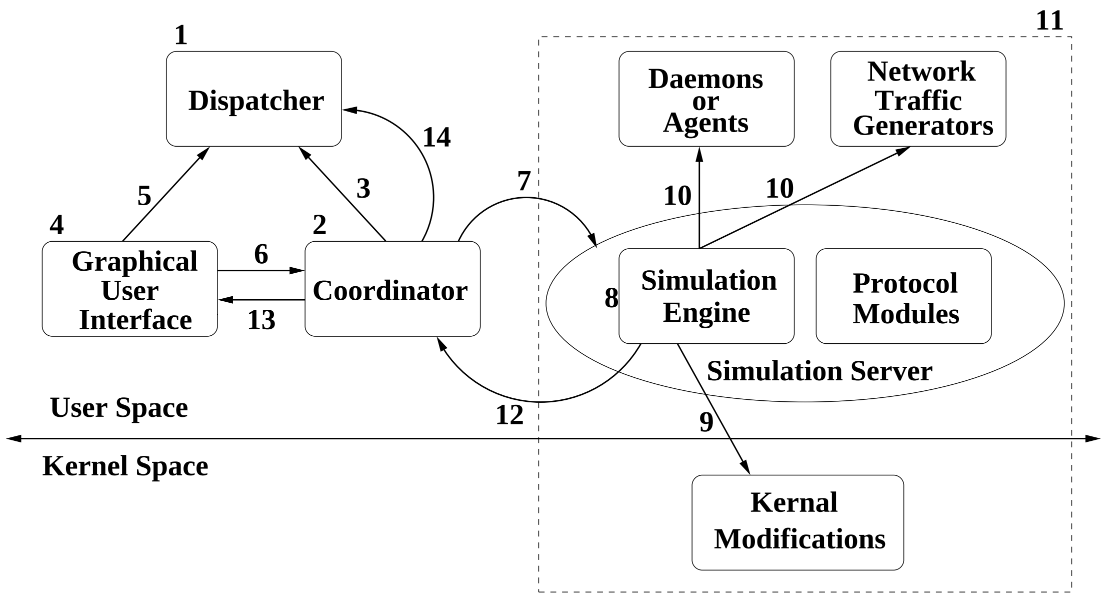


Figure 3.3: The execution procedure of NCTUns

to be triggered by the virtual time rather than by the real time. The kernel modifications will be elaborated later in this chapter.

3.1.2 Execution Procedure

The above-mentioned eight major components comprise the main NCTUns framework. They work together to complete network simulations. Here, we present the execution procedure among all of them to explain how they work with each other during simulation. Figure 3.3 shows the procedure step by step and each step is stated below.

1. First of all, a dispatcher should be ready (executed) for accepting the registration from a coordinator or the simulation job request from a GUI.

2. A coordinator is executed as the agent of the simulation server. Note that so far the simulation server has not been forked by the coordinator.

3. The coordinator builds a TCP connection to the dispatcher and registers its existence through the connection. Now, the dispatcher has the information

about the registered coordinator and treats it as an available agent that can provide the simulation server for service.

4. A user executes the GUI program to draw a network topology, configure network parameters, and let the GUI generate all the required configuration files. Finally, the user triggers a sending out of a simulation jog request.

5. The GUI asks the dispatcher for an available coordinator and waits for the response from it. If a coordinator is available at this time, the response is returned immediately and the status of that coordinator is set to busy by the dispatcher. Otherwise, the GUI has to wait until any busy coordinator becomes available again. If no coordinator has registered to the dispatcher, the response indicating this situation is returned immediately.

6. After obtaining the information of an available coordinator (e.g., the IP address and listen port number used by the coordinator's TCP passive socket), the GUI builds a TCP connection to that coordinator and sends all the configuration files to it to start the simulation.

7. The coordinator places the configuration files into a dynamically allocated directory and forks the simulation server, which includes the simulation engine and some required protocol modules.

8. The simulation engine reads in those configuration files to allocate and initialize the required object for each protocol module, to construct the protocol module stack for each network node, to generate events that will be triggered during simulation to execute daemons, agents, or traffic generators, to set up the required system routing entries, to set up the required tunnel interfaces, and to complete other initialization procedures.

9. The simulation engine resets the states of the NCTUns kernel data structures. For example, the simulated virtual time is reset to zero.

10. If any daemon, agent, or traffic generator has to be run at the beginning of a simulation, it is forked (executed) by the simulation engine now. Of course, a daemon, an agent, or a traffic generator can also be executed at any time during simulation.

11. When the simulation starts, daemons (if any), agents (if any), traffic generators (if any), the simulation engine, the protocol modules, and the NCTUns kernel parts work together to conduct the network simulation. The daemons or agents provide network services. The traffic generators generate background traffics. The kernel provides the operations of the real-life TCP/IP and UDP/IP protocols. The simulation engine and protocol modules provide the operations of various data link layer and physical layer protocols.

12. When the simulation is finished, the simulation engine informs the coordinator, kills all of the daemons (if any), agents (if any), or traffic generators (if any), and finally terminates itself.

13. The coordinator sends back all the log files and statistics files collected during simulation to the GUI.

14. Finally, the coordinator informs the dispatcher that it is available again. The dispatcher then set the coordinator's status to idle.

The current version of NCTUns can only run on the operating systems of Linux Fedora series. Because the kernel modification is required to let NCTUns' architecture work, the kernel source codes must be accessible, modifiable, and able to be re-compiled. Thus, the qualified Linux Fedora series are chosen.

3.2 Kernel Re-entering Simulation Methodology

The kernel re-entering simulation methodology is the key design that allows NCTUns using the real-life TCP/IP and UDP/IP protocol stacks and allows real-life application programs running on NCTUns. In this methodology, the tunnel interface is the key facility. Before explaining how this methodology is applied on NCTUns, we first introduce how a packet is transmitted over a one-hop network connection between two machines in the real world. Based on it, we then show how the tunnel interface is used to realize the simulation of the same one-hop network connection on a single machine.

3.2.1 A Real-world Host-to-host Connection

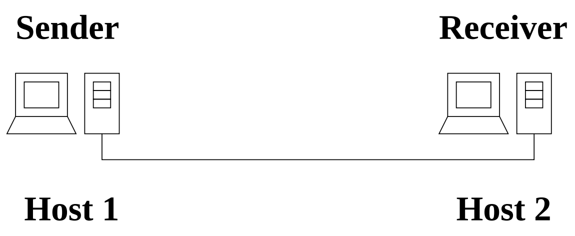
In Figure 3.4a, two real host computers are connected by a network line that could be any kind. A Linux Fedora operating system are run on each of them to provide networking services through the standard POSIX socket interface. We assume that Host 1 is the sender of a network packet and Host 2 is the receiver of that packet. Between being sent and being received, the packet goes through several steps that are depicted in Figure 3.4b and described as follows.

1. The application program run on Host 1 sends out a data segment through the socket. First, the data segment is copied from the user space to the kernel space. Then it reaches the TCP/UDP layer (which is defined as the transportation layer in the OSI reference model [31]).

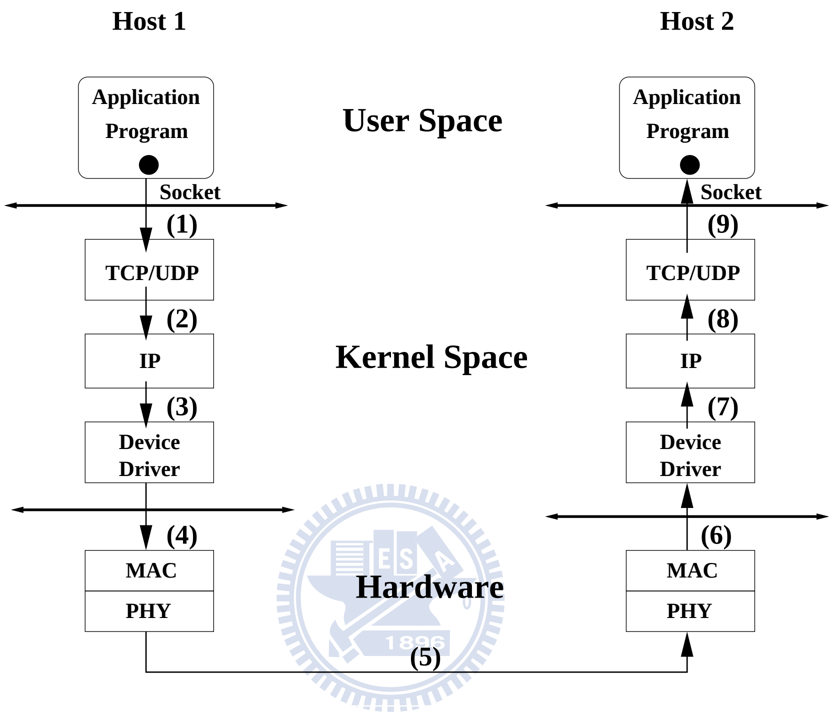
2. The data segment is encapsulated with a TCP/UDP header and becomes a TCP/UDP packet. It then is passed to the IP layer (which is defined as the network layer in the OSI reference model).

3. The TCP/UDP packet will be encapsulated again with an IP header. In addition, a MAC header could be added into this IP packet. The format of the MAC header depends on which kind of media access control protocol is used at the next step. Then this packet is put into a network device driver's output queue to wait for being fetched by the driver's associated network interface card, which is a hardware device.

4. The network interface card copies the queued IP packet from the kernel space to its memory space. Then it sends the packet out on the medium, which is



(a) Network topology



(b) Network packet transmission path

Figure 3.4: A host-to-host network in the real world

a network line in this case. The sending operation is controlled by the media access control (usually called MAC layer and defined as the data link layer in the OSI reference model) module, that is usually made as a firmware within a network interface card.

5. After going through the processes of encoding and modulation in the PHY layer (which is defined as the physical layer in the OSI reference model), the packet stored in digital data format is transformed into a series of analog signals. Finally, the signals are emitted by the radio frequency module within the

network interface card. When the signals reach the receiving radio frequency module, the process of transforming analog signals into digital data format is conducted to obtain the original IP packet.

6. The interface card copies the packet from its memory space to its associated network device driver's input queue and informs the kernel about this incoming packet by triggering a software interrupt.

7. The kernel fetches the packet and puts it to the IP layer. The MAC header and IP header are stripped off here and the resulting TCP/UDP packet is then sent up to the TCP/UDP layer.

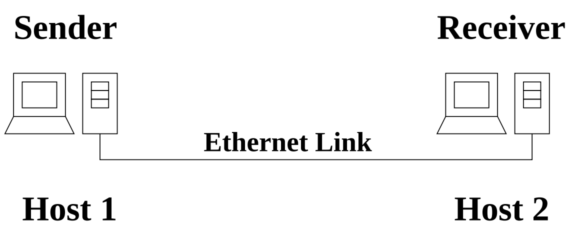
8. After stripping the TCP/UDP header, the kernel puts the data segment in the corresponding socket's receive buffer and informs the receiving application program to fetch the data segment.

3.2.2 A Simulated Host-to-host Connection

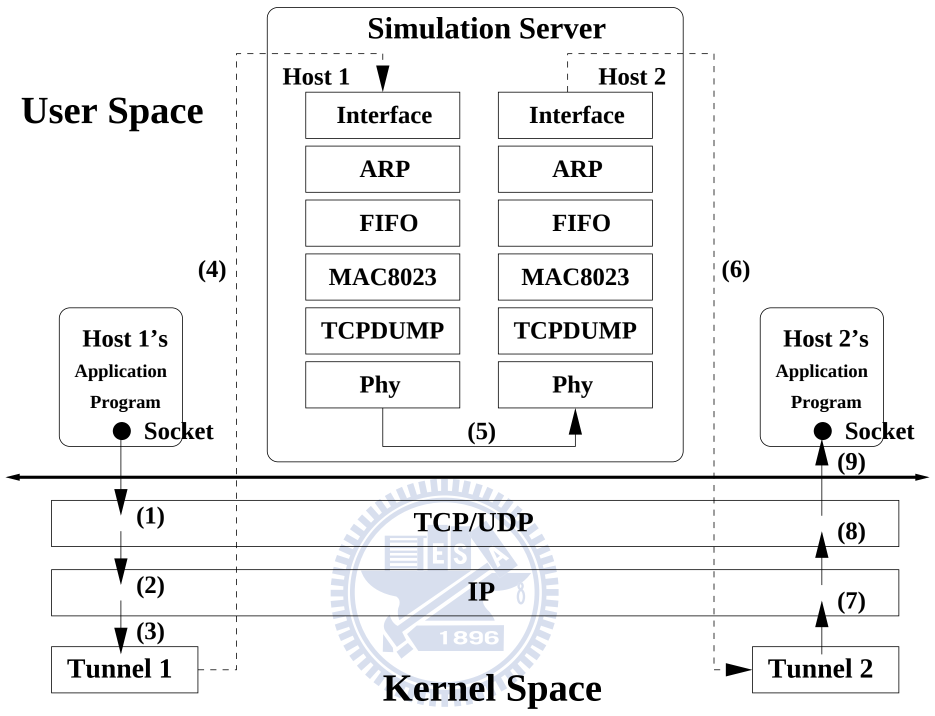
In Figure 3.4, one sees that the real-life TCP/IP or UDP/IP protocol stack is jointed with the real hardware network interface card by the network device driver. Usually, a single machine can support only a few number of network interface cards. This could fail a network simulator to simulate hundreds or thousands of hosts on a single machine and to use the real-life TCP/IP or UDP/IP protocol. NCTUns uses the tunnel interface to solve this problem.

The tunnel interface is the key facility in the kernel re-entering methodology. A tunnel interface is a pseudo network interface and supported by most Linux operating systems. It operates like a network device driver but no real hardware network interface card attaches to it. Figure 3.5 shows how NCTUns simulates the network packet transmission in the network depicted in Figure 3.4a by using tunnel interfaces.

The network topology depicted in Figure 3.5a is the same as that depicted in Figure 3.4a, except that we assume that the network connection is an Ethernet link.



(a) Network topology



(b) Network packet transmission path

Figure 3.5: An Ethernet host-to-host network in simulation

Note that, in Figure 3.5b, the simulation server, the Host 1's application program, and the Host 2's application program are all running on a single machine where the simulation is conducted. The application program can be a daemon, an agent, or a traffic generator. The network packet transmission steps are depicted in Figure 3.5b and described as follows.

1. The application program recognized as a Host 1's application programs sends out a data segment through the socket. The data segment is copied from the user space to the kernel space. Then it reaches the TCP/UDP layer.

2. The data segment is encapsulated with a TCP/UDP header and becomes a TCP/UDP packet. It then is passed to the IP layer.

3. The TCP/UDP packet will be encapsulated again with an IP header. Because the exit interface is a tunnel interface, no MAC header is added into the IP packet by the kernel. Then, instead of being put into a network device driver's output queue, the packet is put into the output queue of Tunnel 1, which is a tunnel interface. The simulation server sets this tunnel interface to be associated with Host 1 before the simulation starts. In order to let a tunnel interface imitate a real Ethernet interface, we modify the tunnel interface to add an Ethernet header into the IP packet before the packet is put into the queue. The field of destination MAC address in the header is set to the next hop's IP address instead of the next hop's MAC address. This is because the next hop's MAC address is unavailable here.

4. Later on, the simulation server reads the queued IP packet from Tunnel 1 to the Interface module that is associated with Tunnel 1. Within the simulation server, the ARP module performs the ARP protocol to find out the next hop's MAC address according to the next hop's IP address. It then fills all the fields of the Ethernet header. The FIFO module functions as an interface output queue. In this module, the queue length is synchronized with the associated tunnel interface's queue length so that in effect only one output queue exists for an interface. The MAC8023 module performs the IEEE 802.3 MAC protocol (usually known as CSMA/CD). The TCPDUMP module supports the tcpdump program provided by Linux operating systems.Finally, the Phy module simulates the transmission delay and the propagation delay of the packet and sends the packet to the receiving Phy module.

5. The sending Phy module and receiving Phy module are bound together by a wired link. The sent packet from the sending module will reach the receiving module after the time period of propagation delay that is specified by users. The received packet travels up the protocol modules to the Interface module

associated with Tunnel 2, which is the tunnel interface associated with Host 2.

6. The Interface module writes the packet to Tunnel 2's input queue and informs the kernel about this incoming packet by triggering a software interrupt. Note that the Ethernet header is not copied back to the kernel space.

7. The kernel fetches the packet and puts it to the IP layer. The IP header is stripped off here and the resulting TCP/UDP packet is then sent up to the TCP/UDP layer.

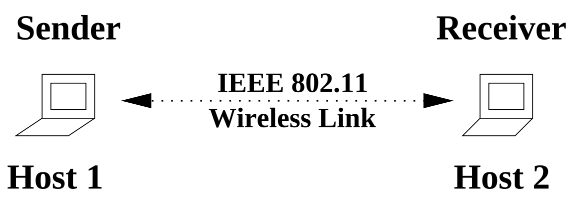
8. After stripping the TCP/UDP header, the kernel puts the data segment in the corresponding socket's receive buffer and informs the application program recognized as a Host 2' application programs to fetch the data segment.

During the simulation described above, the transmitted packet enters and exits the same kernel twice on a single machine. This is why this methodology is named "kernel re-entering simulation methodology." As stated before, using this methodology, NCTUns uses the real-life TCP/IP and UDP/IP protocol stacks and supports real-life application programs to run on itself.

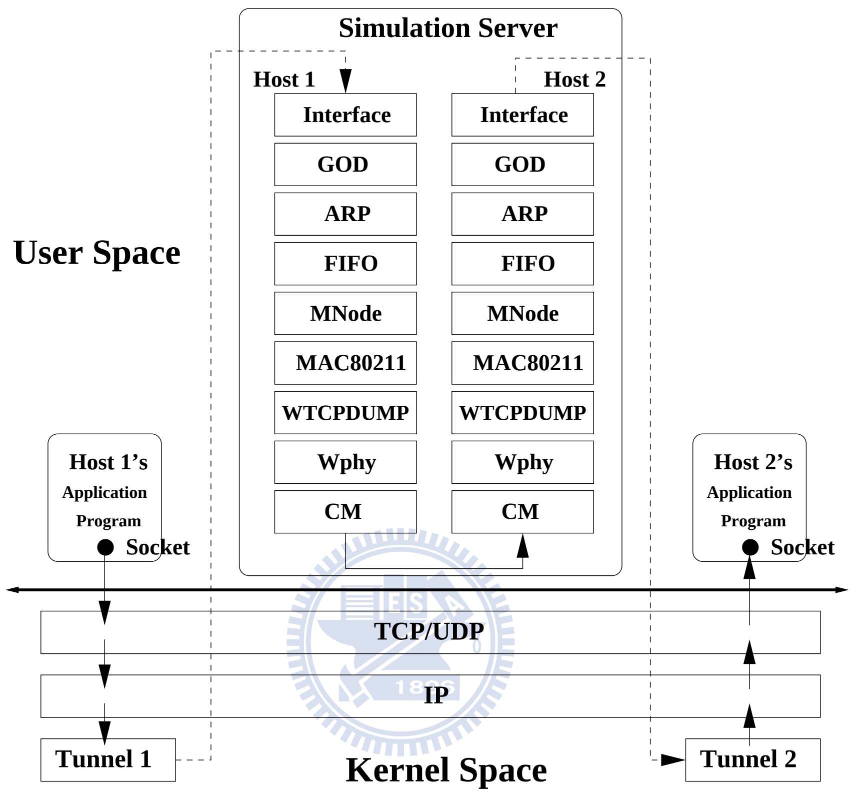
3.2.3 A Simulated Mobile-host-to-mobile-host Connection

Next, we show a simulated network containing two mobile hosts each of which is equipped with an IEEE 802.11b wireless interface. The network topology is shown in Figure 3.6a and the packet transmission path is depicted in Figure 3.6b. Comparing Figure 3.6b with Figure 3.5b, one can see that the only difference is the protocol module stack within the simulation server.

Regarding the protocol stack of a mobile host, the GOD module is the default routing daemon module for an IEEE 802.11b wireless interface. This module does not perform any routing protocol during simulation. Instead, it sets the routing entries according to a schedule that is provided by the GUI. When a user manipulates the GUI to set each mobile node's moving path before starting the simulation, the



(a) Network topology



(b) Network packet transmission path

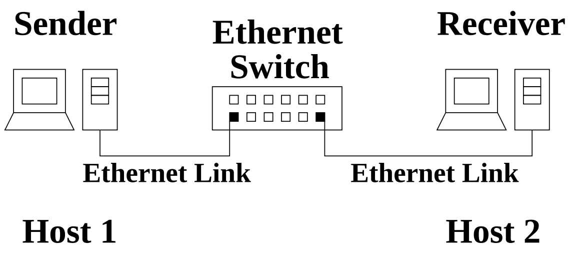
Figure 3.6: An IEEE 802.11 mobile-host-to-mobile-host network in simulation

GUI calculates each mobile node's routing entries according to the relative movements among mobile nodes. The resulting routing entries are written into a file by the GUI. Later on, when the simulation starts, the simulation server reads in these routing entries and schedules events to be triggered during simulation. When one of these events is triggered, the corresponding GOD module's routing table is updated. The GOD module represents an optimum routing path search result and is usually used to compare with other routing protocol implementations. In addition to the GOD module, NCTUns also provides AODV, ADV, DSDV, and DSR modules. Note that all of these routing protocols are implemented in the form of protocol module. Of course, they can be implemented as a user space daemon, like the RIP and OSPF daemons mentioned before.

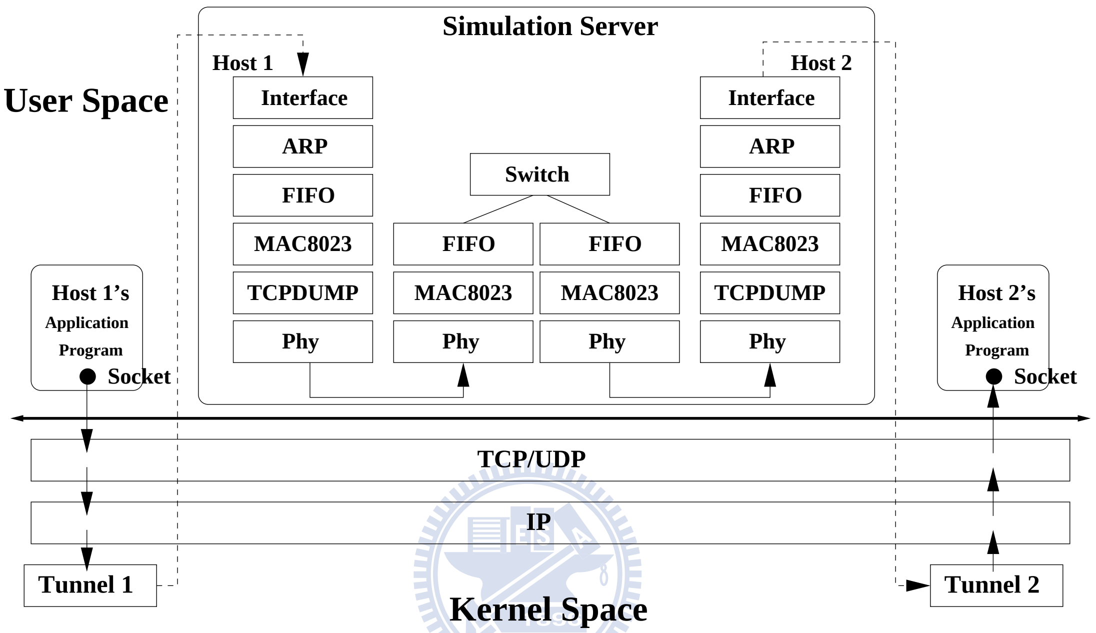
Besides, the MNode and MAC80211 modules perform the IEEE 802.11b MAC protocols (usually known as CSMA/CA). The WTCPDUMP module (W stands for wireless) supports the tcpdump program provided by Linux operating systems. Regarding the Wphy module, it is like the Phy module but it does not have the one-to-one connection relationship with another Wphy module. Because the wireless signals are media with broadcast nature, the Wphy module has the one-to-many connection relationship with other Wphy modules, and so do other counterpart modules used in different types of wireless networks. After the Wphy module determines those other Wphy modules to which a duplicated packet should be sent, an outgoing packet is duplicated based on the number of target Wphy modules. Next, each duplicate is sent to the CM module. In the CM module, several channel models can be selected, each of which is associated with different wireless signal transmission technologies and environments. The degradation of wireless signal power is calculated in the CM module based on the related physical layer parameters, such as the antenna gain patterns used in the sender mobile host and the receiver mobile host.

3.2.4 A Simulated Host-to-switch-to-host Connection

Next, we insert a layer-2 Ethernet switch into the connection between two hosts, as that shown in Figure 3.7a. Comparing Figure 3.7b with Figure 3.5b, one can see that the difference is the switch's protocol module stacks within the simulation server. In this example network, only two ports of the switch are used. Thus, only two protocol module stacks, each of which is for one port, are created by the simulation server. Because the switch is a layer-2 device, the Phy module (layer 1) and the MAC8023 module (layer 2) are put into the protocol stack. The Switch module determines which port an incoming packet should be sent to according to the destination MAC address recorded in that packet's Ethernet header. Besides,



(a) Network topology



(b) Network packet transmission path

Figure 3.7: An Ethernet host-to-switch-to-host network in simulation

instead of implementing additional output queue functions in the Switch module, we use the existing FIFO module. Thus, the FIFO module functions as each switch port's output queue here.

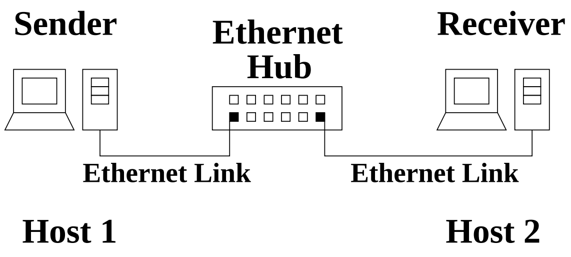
3.2.5 A Simulated Host-to-hub-to-host Connection

Next, we replace the layer-2 Ethernet switch with a layer-1 Ethernet hub on the connection between two hosts. The network topology is shown in Figure 3.8a. Comparing Figure 3.8b with Figure 3.7b, one can see that the difference is that each of the hub's ports does not have the MAC8023 module (layer 2). This difference determines the different simulation method on packet forwarding. Because an Ethernet switch is a layer-2 device, it simulates a store-and-forward way between a packet entering it from one port and leaving it from another port. A packet experiences the transmission delay when it travels through the switch. However, because an Ethernet hub is a layer-1 device, it simulates a pass-through way. That means a packet experiences no delay between entering a Ethernet hub and leaving it.

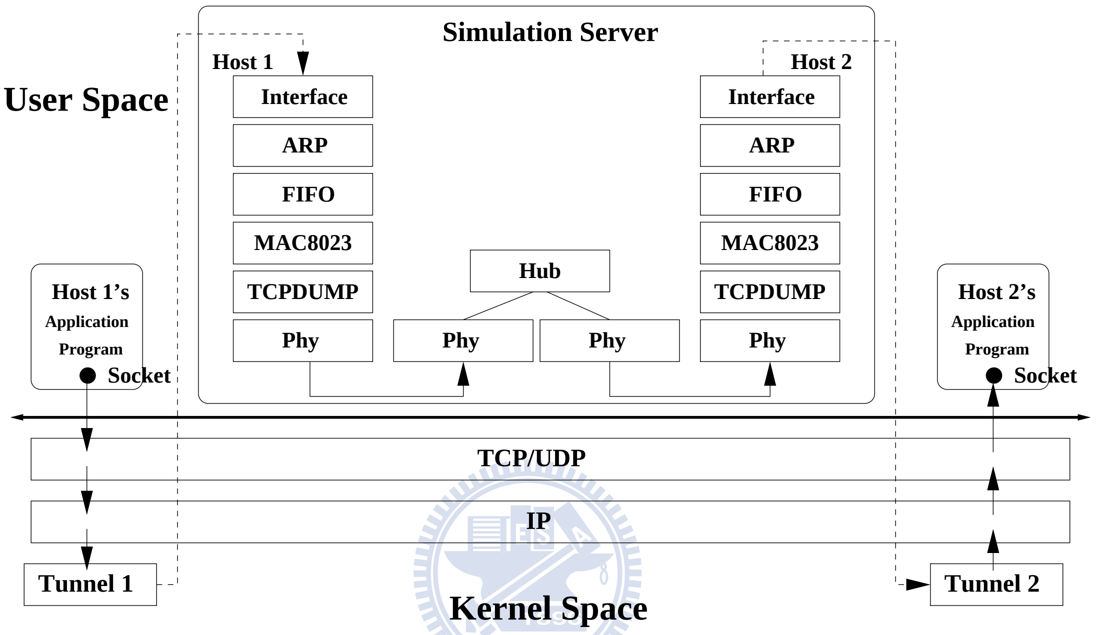
During simulation, the log of the packet transmission/reception from one device to another device is usually achieved by the layer-2 MAC modules located in the sending device and receiving device respectively. In the case of a switch device, its MAC modules deal with the log. However, in the case of a hub device, the Hub module has to do the log. For example, when a packet enters the hub, the Hub module has to trace the packet's source MAC module and destination MAC module. Besides, it has to keep monitoring the packet's reception status (success or failure) in the destination MAC module. When the packet's reception status is known, the Hub module has to do two logs. For the real source MAC module, the Hub module functions as its corresponding destination MAC module to complete the first log. For the real destination MAC module, the Hub module functions as its corresponding source MAC module to complete the second log. Lacking of the Hub module's log capability does not affect the simulation correctness but just loses some simulation detail for debugging (if needed) and for displaying the animation of packet transmission/reception.

3.2.6 A Simulated Host-to-router-to-host Connection

In this case, we add a layer-3 Ethernet router into the connection between two hosts, as that shown in Figure 3.9a. Because the Ethernet router is a layer-3 device, it performs the routing functions that is provided in the kernel space. In Figure 3.9b, one can see that the router device has two protocol stacks each of which is exactly the same as a host's protocol stack. Besides, the Interface module belonging to the router's first interface is associated with Tunnel 3 in the kernel space, while the



(a) Network topology

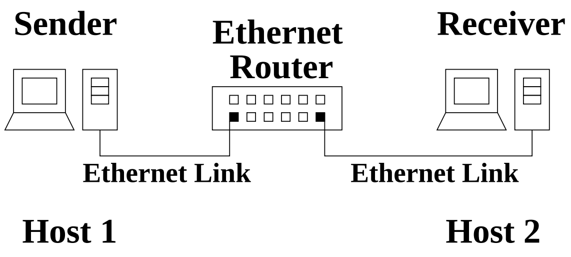


(b) Network packet transmission path

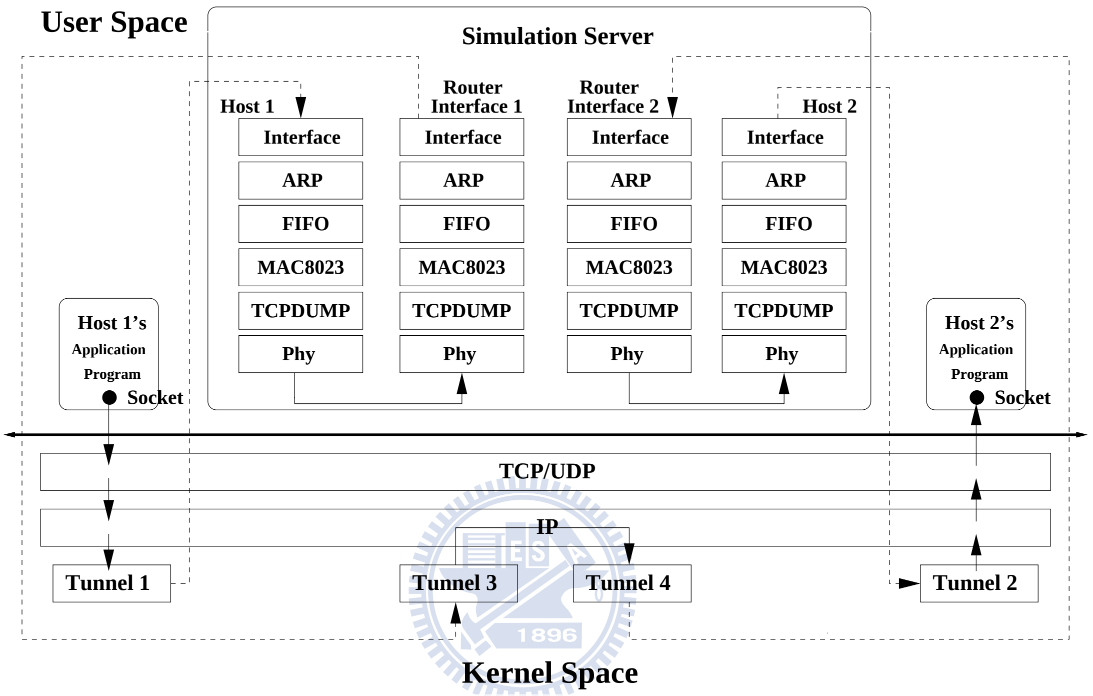
Figure 3.8: An Ethernet host-to-hub-to-host network in simulation

Interface module belonging to the router's second interface is associated with Tunnel 4 in the kernel space.

Regarding the simulation of packet transmission in this network, a packet has to travel a host-to-router connection first, that is equivalent to a host-to-host connection except that the router is not the destination of the packet. When it reaches the Interface module of the router's first interface, it is written to Tunnel 3 in the kernel space. Then, because the packet's destination IP address indicates that it has not yet reached the destination interface, it is forwarded to the Tunnel 4 by the kernel's forwarding function. Later on, the simulation server reads the packet from



(a) Network topology



(b) Network packet transmission path

Figure 3.9: An Ethernet host-to-router-to-host network in simulation

Tunnel 4 to the Interface module of the router's second interface. Then the packet continues traveling a router-to-host connection and reaches its destination host.

3.3 Routing Scheme

In the architecture of NCTUns, the real-life IP protocol stack in the kernel space is involved. The routing/forwarding functions in the kernel is directly used without modification. In other words, the real-life routing scheme has to be followed so that the kernel's routing/forwarding functions can work correctly to forward IP packets.

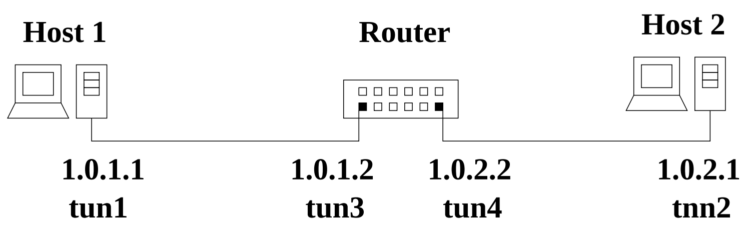
However, a routing entry conflict problem occurs when setting the routing entries for a simulated network using the original routing scheme. Next, we explain the problem and the solution to it.

Figure 3.10a shows the example network topology. Two class C subnets are jointed by a Ethernet router. One's subnet ID is 1.0.1.0/24 (referred as subnet 1 later) and the other's is 1.0.2.0/24 (referred as subnet 2 later). Host 1 is placed under subnet 1 and its associated tunnel interface is tun1. The IP address assigned to tun1 is 1.0.1.1. Host 2 is placed under subnet 2 and its associated tunnel interface is tun2. The IP address assigned to tun2 is 1.0.2.1. Regarding the router, it has two interfaces in this example network. One is attached to subnet 1 and its associated tunnel interface is tun3. The IP address assigned to tun3 is 1.0.1.2. The other interface is attached to subnet 2 and its associated tunnel interface is tun4. The IP address assigned to tun4 is 1.0.2.2.

Based on the original routing scheme, the required routing entries for Host 1, Host 2, and the router are listed in Figure 3.10b. Regarding the entries for Host 1, if the outgoing packet's destination IP address is 1.0.1.2, it is sent out through tun1 directly because the destination is located at the same subnet as Host 1 is located. If the outgoing packet's destination IP address is 1.0.2.1 or 1.0.2.2, it is sent to the gateway interface with the IP address 1.0.1.2 because the destination is located at different subnet. The routing entries for Host 2 and the router are set using the same rules to set Host 1's routing entries.

The individual set of routing entries of each device seems to work well to route packets in this network. However, because NCTUns runs on a single machine with a single kernel, only one kernel routing table exists to store the routing entries. If all of the routing entries listed in Figure 3.10b are added into the only routing table, a conflict situation occurs. That is, for any destination IP address, two routing entries with different gateway IP address and different output interface need to be set. This conflict situation fails the routing function in the simulated network.

When inspecting each pair of conflicted routing entries, one can see that each entry belongs to different network device. In other words, losing the information of



(a) Network topology

For Host 1

Dst	Gw	If
1.0.1.2	0.0.0.0	tun1
1.0.2.1	1.0.1.2	tun1
1.0.2.2	1.0.1.2	tun1

For Router

Dst	Gw	If
1.0.1.1	0.0.0.0	tun3
1.0.2.1	0.0.0.0	tun4

For Host 2

Dst	Gw	If
1.0.2.2	0.0.0.0	tun2
1.0.1.1	1.0.2.2	tun2
1.0.1.2	1.0.2.2	tun2

(b) Routing entries with conflicts

For Host 1

Dst	Gw	If
1.1.1.2	0.0.0.0	tun1
1.1.2.1	1.1.1.2	tun1
1.1.2.2	1.1.1.2	tun1

For Router

Dst	Gw	If
1.2.1.1	0.0.0.0	tun3
2.2.1.1	0.0.0.0	tun3
1.2.2.1	0.0.0.0	tun4
2.2.2.1	0.0.0.0	tun4

For Host 2

Dst	Gw	If
2.1.2.2	0.0.0.0	tun2
2.1.1.1	2.1.2.2	tun2
2.1.1.2	2.1.2.2	tun2

(c) Routing entries without any conflict

Figure 3.10: An example of the routing scheme used in NCTUns

the source device that sending packets leads to the conflict. Thus, the NCTUns' solution is to add the IP address of the tunnel interface from which a packet is sent out into the destination IP address in corresponding routing entries.

Figure 3.10c shows the modified routing entries for each devices in the simulated network. Those underscores indicate the modified portion in destination and gateway IP addresses. Regarding the entries for Host 1, the first two octets of each IP address is replaced by the last two octets of Host 1's output tunnel interface's IP address. That is 1.1. Regarding the entries for Host 2, the same change is applied based on the last two octets of its output tunnel interface's IP address. That is 2.1. For the router, because it has two tunnel interfaces, each tunnel interface's IP address is used to change the IP address in each routing entry. Thus, the number of routing entries for the router is doubled after the modification. Now, no conflict exists when storing all the routing entries into a single routing table. The new representation of IP address is called the "source-destination-pair" format. That is because the new IP address contains both the part of source interface's IP address and the part of destination interface's IP address.

When the source-destination-pair IP format is used, only the third and fourth octets of a tunnel interface's IP address are used to identify that tunnel interface. In other words, the first and second octets of a tunnel interface's IP address are useless. By default, NCTUns sets the first octet to 1 (0x01 in hex) and the second octet to 0 (0x00 in hex). Thus, all IP addresses assigned to tunnel interfaces begin with 1.0. Besides, because the third octet of IP address is used to denote the subnet ID and the fourth octet is used to denote the host ID, NCTUns can support up to 255 subnets and up to 255 IP addresses within each subnet.

3.4 IP Address Translation

When source-destination-pair IP addresses are used, those user space application programs have to use this IP format to indicate the destination IP address of a sent packet. For example, in Figure 3.10a, if an application program running on Host 1 wants to send a packet to another application program running on Host 2, the packet's destination IP address should be set as 1.1.2.1 by the sending application program. Reversely, the destination IP address should be set as 2.1.1.1.

Because the source-destination-pair IP format is unnatural to users who are familiar with the original definition of IP address format, NCTUns provides a solution, called IP address translation, to alleviate users' inconvenience. For those users who want to develop a user space routing daemon, the knowing of the source-destination-pair IP format is necessary because the routing daemon needs to manipulate the kernel's routing table. However, for those who want to develop/use a user space traffic generator or protocol module within the simulation server, they can be unaware of the source-destination-pair IP format and still conduct simulations using

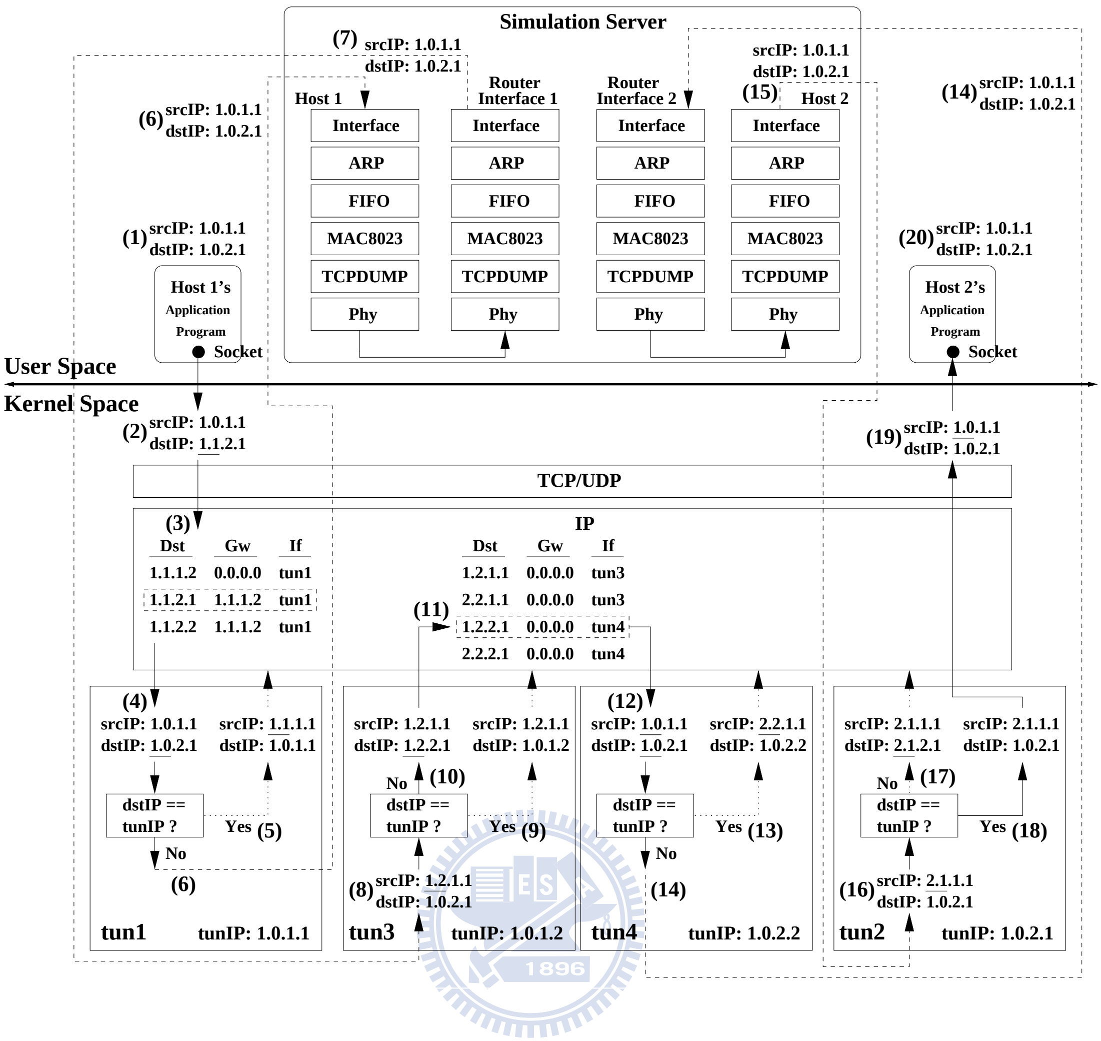

Simulation Server
(7) srcIP: 1.0.1.1 dstIP: 1.0.2.1
srcIP: 1.0.1.1 dstIP: 1.0.2.1
Host 1
Router Interface 1
Router Interface 2
(15) Host 2
(14) srcIP: 1.0.1.1 dstIP: 1.0.2.1
(6) srcIP: 1.0.1.1 dstIP: 1.0.2.1
Interface
ARP
FIFO
MAC8023
TCPDUMP
Phy
(1) srcIP: 1.0.1.1 dstIP: 1.0.2.1
(20) srcIP: 1.0.1.1 dstIP: 1.0.2.1
Host 1's Application Program
Host 2's Application Program
Socket
User Space
Kernel Space
(2) srcIP: 1.0.1.1 dstIP: 1.1.2.1
(19) srcIP: 1.0.1.1 dstIP: 1.0.2.1
TCP/UDP
IP
(3)
Dst Gw If
1.1.1.2 0.0.0.0 tun1
1.1.2.1 1.1.1.2 tun1
1.1.2.2 1.1.1.2 tun1
(11)
1.2.1.1 0.0.0.0 tun3
2.2.1.1 0.0.0.0 tun3
1.2.2.1 0.0.0.0 tun4
2.2.2.1 0.0.0.0 tun4
(4) srcIP: 1.0.1.1 dstIP: 1.0.2.1
srcIP: 1.1.1.1 dstIP: 1.0.1.1
dstIP == tunIP ?
Yes (5)
No
(6)
tun1 tunIP: 1.0.1.1
srcIP: 1.2.1.1 dstIP: 1.2.2.1
srcIP: 1.2.1.1 dstIP: 1.0.1.2
No (10)
Yes (9)
(8) srcIP: 1.2.1.1 dstIP: 1.0.2.1
tun3 tunIP: 1.0.1.2
(12) srcIP: 1.0.1.1 dstIP: 1.0.2.1
srcIP: 2.2.1.1 dstIP: 1.0.2.2
Yes (13)
(14)
tun4 tunIP: 1.0.2.2
srcIP: 2.1.1.1 dstIP: 2.1.2.1
srcIP: 2.1.1.1 dstIP: 1.0.2.1
No (17)
Yes (18)
(16) srcIP: 2.1.1.1 dstIP: 1.0.2.1
tun2 tunIP: 1.0.2.1

Figure 3.11: An example of IP address translation

the original IP address format.

An example of IP address translation is demonstrated in Figure 3.11. The network topology, IP assignment, and tunnel interface assignment are the same as those used in Figure 3.10a. Thus, the routing entries used in this example network are also the same as those listed in Figure 3.10c. In Figure 3.11, a packet is sent from the application program belonging to Host 1 to the application program belonging to Host 2. The packet's traveling path is traced to show how the IP address translation is applied to the source and destination IP addresses of the packet's IP header. Each step of the trace is described below.

1. When the data segment is sent out by the Host 1' application program through

the socket, the source and destination IP addresses are recorded in some related socket data structures. In this example, the source IP address is 1.0.1.1 and the destination IP address is 1.0.2.1.

2. When the data segment is copied from the user space to the kernel space, it is stored in a sk_buff data structure as well as the output interface information and destination information. Here, the first two octets of the destination IP address are replaced with the last two octets of the source IP address. Note that the source IP address is still the output interface's IP address. The underscore indicates the changed portion of the destination IP address.

3. Referring to the destination IP address, the kernel finds out the output interface by searching the routing table. Then the packet is sent to tun1.

4. Once the packet arrives at tun1 from the IP layer, the first two octets of the source and the destination IP addresses are changed to 1.0. Then it undergoes a condition check.

5. If the destination IP address is equal to tun1's IP address, then the packet has reached its destination and should be looped back to the upper layer. If this is the case, the first two octets of the source IP address are replaced with the last two octets of tun1's IP address. Thus, the source IP address should be 1.1.1.1 and destination IP address should be 1.0.1.1. However, in this example, this is not the case.

6. Because the destination IP address is unequal to tun1's IP address, the packet is sent to the simulation server for going through the data link layer and physical layer simulations. Note that when the simulation server gets the packet, the source IP address is 1.0.1.1 and the destination IP address is 1.0.2.1.

7. The source and destination IP addresses keep unchanged when the packet stays in the simulation server. Thus, when the packet is copied from the simulation

server to another tunnel interface tun3. The source IP address is still 1.0.1.1 and the destination IP address is still 1.0.2.1

8. Once the packet arrives at tun3 from the simulation server, the first two octets of the source IP address are replaced with the last two octets of tun3's IP address. Here the destination IP address keeps unchanged. Thus, the source IP address is 1.2.1.1 and the destination IP address is 1.0.2.1. Then the packet undergoes a condition check.

9. If the destination IP address is equal to tun3's IP address, then the packet has reached its destination. If this is the case, then both the source and destination IP addresses are unchanged and the packet is sent to upper layer. In this case, the source IP address should be 1.2.1.1 and the destination IP address should be 1.0.1.2. Of course, this is not the case in this example.

10. Because the destination IP address is unequal to tun3's IP address, the first two octets of the destination IP address are replaced with the last two octets of tun3's IP address. Now, the source IP address is 1.2.1.1 and the destination IP address is 1.2.2.1. Then the packet is sent to IP layer for being forwarded.

11. Referring to the destination IP address, that is 1.2.2.1, the kernel finds out the output interface by searching the routing table. Then the packet is sent to tun4.

12. Once the packet arrives at tun4 from the IP layer, the first two octets of the source and the destination IP addresses are changed to 1.0. Then it undergoes a condition check.

13. If the destination IP address is equal to tun4's IP address, then the packet has reached its destination and should be looped back to the upper layer. If this is the case, the first two octets of the source IP address are replaced with the last two octets of tun4's IP address. Thus, the source IP address should

be 2.2.1.1 and destination IP address should be 1.0.2.2. Obviously, this is not the case in this example.

14. Because the destination IP address is unequal to tun4's IP address, the packet is sent to the simulation server for going through the data link layer and physical layer simulations. Note that when the simulation server gets the packet, the source IP address is 1.0.1.1 and the destination IP address is 1.0.2.1.

15. Again, the source and destination IP addresses keep unchanged when the packet stays in the simulation server. Thus, when the packet is copied from the simulation server to another tunnel interface tun2. The source IP address is still 1.0.1.1 and the destination IP address is still 1.0.2.1

16. Once the packet arrives at tun2 from the simulation server, the first two octets of the source IP address are replaced with the last two octets of tun2's IP address. Here the destination IP address also keeps unchanged. Thus, the source IP address is 2.1.1.1 and the destination IP address is 1.0.2.1. Then the packet undergoes a condition check.

17. If the destination IP address is unequal to tun2's IP address, the first two octets of the destination IP address should be replaced with the last two octets of tun2's IP address. Thus, the source IP address should be 2.1.1.1 and the destination IP address should be 2.1.2.1. Then the packet should be sent to IP layer for being forwarded. However, this is not the case in this example.

18. Because the destination IP address is equal to tun2's IP address, the packet has reached its destination. Both the source and destination IP addresses are unchanged and the packet is sent to upper layer. In this case, the source IP address is 2.1.1.1 and the destination IP address is 1.0.2.1. Note that unlike the destination address, the source IP address keeps in the source-destination-pair format. This is because some kernel protocol implementations need this information to send back responding packets. For example, an ICMP echo reply should be sent back when the kernel receives an ICMP echo request

(network layer). Yet another example is that a TCP acknowledge has to be sent when the kernel receives a TCP data segment (transportation layer).

19. When the data segment, stored in a sk_buff data structure, is put into the socket receive buffer by the kernel, the first two octets of the source IP address are changed to 1.0. Thus, the source IP address is 1.0.1.1 and the destination IP address is 1.0.2.1 here.

20. When the Host 2's application program refers to the IP address information from some related socket data structures, it gets that the source IP address is 1.0.1.1 and the destination IP address is 1.0.2.1.

Note that in Figure 3.11, when any user space program wants to access the IP address information, it just uses the format of 1.0.*.* instead of the source-destination-pair format. This is how the IP address translation provides the original view of IP address format to those who need not manipulate the IP routing functions.

3.5 Discrete-event Simulation Engine

NCTUns is an event-driven simulator. In Figure 3.3, the daemons, the agents, the traffic generators, the simulation server, and the kernel modifications are separated components in NCTUns' architecture. When a simulation is running, the virtual time synchronization has to be achieved among them. Besides, the events or timers generated by these separated components need to be scheduled in order based on each event's or timer's triggered time. In NCTUns, the main component taking care of these jobs is the simulation engine. Next, we present how the simulation engine deals with the virtual time synchronization and the event/timer scheduling on the NCTUns platform.

3.5.1 Virtual Time Synchronization

The simulation engine has to advance the virtual time during simulation and pass the virtual time into the kernel. This is required for some reasons that affect the accuracy of simulation results. First, the timers of TCP connections used in the simulated network need to be triggered based on the virtual time rather than the real time. Second, for those application programs run in the simulated network, the timer-related system calls issued by them must be serviced based on the virtual time rather than the real time. For example, if a ping program is executed to send out an ICMP echo request packet every one second, the sleep() system call issued by the ping program must be triggered based on the virtual time. Third, the in-kernel packet logging mechanism (i.e., the Berkeley packet filter scheme used by the tcpdump program) needs to use timestamps based on the virtual time to log packets transferred in a simulated network. Moreover, the NCTUns kernel events need to use timestamps based on the virtual time to let the simulation engine know their trigger time.

The simulation engine can pass the current virtual time into the kernel by periodically calling a customized system call or calling the customized system call whenever the virtual time changes. If the former way is used, the virtual time granularity in the kernel depends on the simulation engine's update period. Infrequent update may lead to inaccurate simulation results. For instance, the in-kernel packet logging mechanism needs a microsecond-resolution clock to generate timestamps. If the simulation engine's update period is longer than 1 microsecond, the timestamps in the log is inaccurate. If the latter way is used, the very frequent update of the virtual time causes very high system call overhead that slows down the simulation speed. To solve this problem, we use a memory-mapping technique to allow the simulation engine accessing a 64-bits integer data structure whose memory address space is located in the kernel space. In other words, only one integer data structure is used to store the current virtual time during simulation and it can be accessed simultaneously by the simulation engine and the kernel. With this technique, the virtual time obtained in the kernel is always the same as that obtained in the simulation engine during simulation. Besides, during simulation no system call is needed to continually update the virtual time in the kernel space.

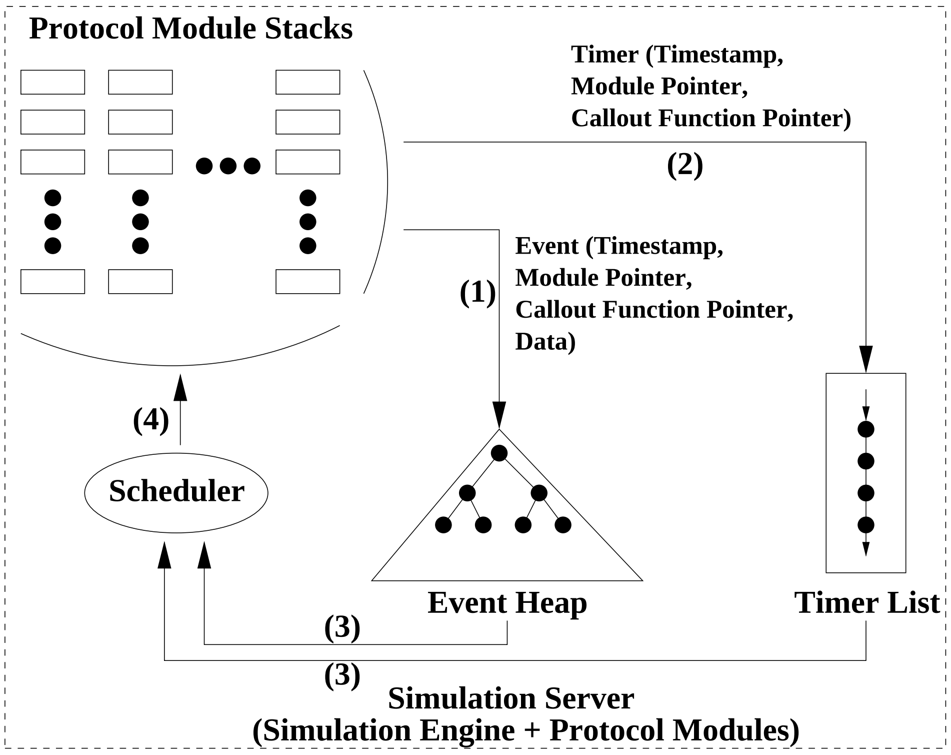

Protocol Module Stacks
Timer (Timestamp, Module Pointer, Callout Function Pointer)
(2)
Event (Timestamp, Module Pointer, Callout Function Pointer, Data)
(1)
(4)
Scheduler
Event Heap
Timer List
(3)
(3)
Simulation Server
(Simulation Engine + Protocol Modules)

Figure 3.12: The execution procedures of the protocol module event and timer

3.5.2 Protocol Module Event and Timer

The protocol module event and timer are generated by protocol modules. When an event or a timer is triggered, a function belonging to some protocol module is called by the scheduler located in the simulation engine. Figure 3.12 shows the generation and the execution of this kind of event or timer. Each step is described below.

1. A protocol module event is generated from some function within a protocol module. Usually, it is used to deliver some sort of data (e.g., a network packet) to another function within the same protocol module or to some function belonging to another protocol module. For example, one can use this event to purposely hold a network packet for a period of time within a protocol module. The effect is like that the packet experiences a period of delay somewhere on its journey in the network. Another example is to simulate a packet's transmission on the medium. In this case, the packet leaves from one physical

layer protocol module first, experiences the propagation delay on the medium, and finally reaches another physical layer protocol module. When generating a protocol module event, the protocol module has to specify the timestamp when the event should be triggered, the module pointer pointing to itself or another protocol module, the function pointer pointing to the callout function, and the data to be delivered. The event is then inserted into a heap data structure called event heap. The event with smallest timestamp is always placed on the root of the event heap. During simulation, the simulation engine continually retrieves the event at the root to execute.

2. A protocol module timer is also generated from a protocol module. Unlike the protocol module event, a timer is usually used to call a routine function periodically or just call a function after a period of time from when the timer is started. Thus, no delivery data is carried by the timer. Actually, the protocol module timer is implemented based on the protocol module event by setting a timer object as the delivery data within the protocol module event. Thus, a protocol module timer only has the information of the timestamp, the module pointer, and the callout function pointer. The timer is inserted into the timer list, within which timers are sorted according to their timestamps, instead of the event heap. This is because a timer may be suspended, resumed, or canceled during simulation. Supporting these operations to the event heap increases a lot of time and space complexity and even breaks the original operation principle of a heap data structure. Thus, the timer list is used to deal with the timer operations.

3. During simulation, the simulation engine's scheduler keeps checking the event at the heap root and the timer at the list head. The one with the smallest timestamp is retrieved to execute. Besides, the scheduler advances the virtual time based on that timestamp. Thus, the virtual time advancing may not be continuous. This is the working principle of discrete-event simulation engine.

4. According to the module pointer and the callout function pointer, the scheduler

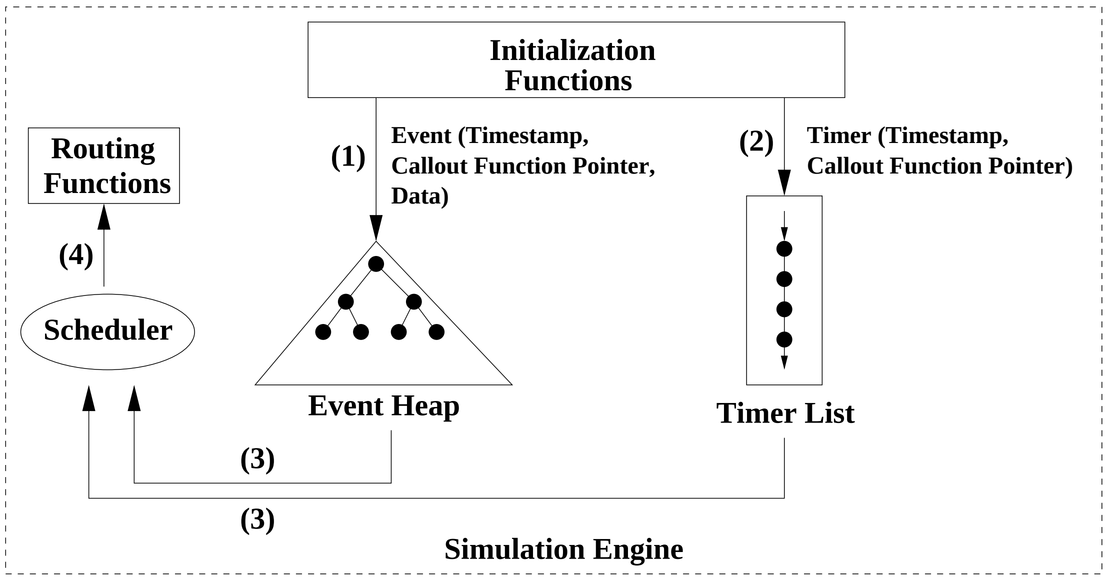


Figure 3.13: The execution procedures of the routine function event and timer

invokes the corresponding function within the specified module. Besides, the event which carries a network packet or a timer object is delivered as an input argument to the corresponding function.

3.5.3 Routine Function Event and Timer

The routine function event and timer are generated by the initialization functions within the simulation engine. When this kind of event or timer is triggered, some corresponding routine function is called by the scheduler. Figure 3.13 shows the generation and the execution of this kind of event or timer. Each step is described below.

1. A routine function event is generated during the initialization procedure at the beginning of a simulation. It is usually used to execute a routine function within which some specified command is executed during simulation, such as forking an application program by invoking the execv() system call. Within a routing function event, the triggered timestamp, the callout function pointer, and the data in which the specified command is recorded are carried. As well

as the protocol module event, the routing function event is inserted into the event heap.

2. A routine function timer is also generated during the initialization procedure at the beginning of a simulation. It is used to execute some routing functions during simulation, such as displaying the progress of the virtual time, sending the virtual time to the GUI, checking the run-time command from the GUI, and so on. Within a routing function timer, only the triggered timestamp and the callout function pointer are carried. As well as the protocol module timer, the routing function timer is inserted into the timer list.

3. The retrieve of this kind of event or timer is the same as that of the protocol module event or timer except that the callout function belongs to the simulation engine, not a protocol module. Besides, the scheduler advances the virtual time based on the triggered timestamp of the event or timer.

4. According to the callout function pointer, the scheduler invokes the corresponding function to execute some routine function.

3.5.4 Process Timer Event

As stated before, for application programs run in the simulated network, their timer-related system calls must be serviced based on the virtual time. This requirement is achieved by the process timer events. Figure 3.14 shows the generation and the execution of this kind of event. Each step is described below.

1. In Linux, each running user space process has a corresponding data structure in the kernel space, called task_struct. The task_struct is the identification of a user space process in the kernel space and is used in many kernel work, such as process scheduling and kernel resource allocation. In NCTUns' implementation, some additional information are added into the task_struct, one of which is the identification number of a simulated node/device (e.g., host, router, hub, switch, etc.). During simulation, when a user space application program

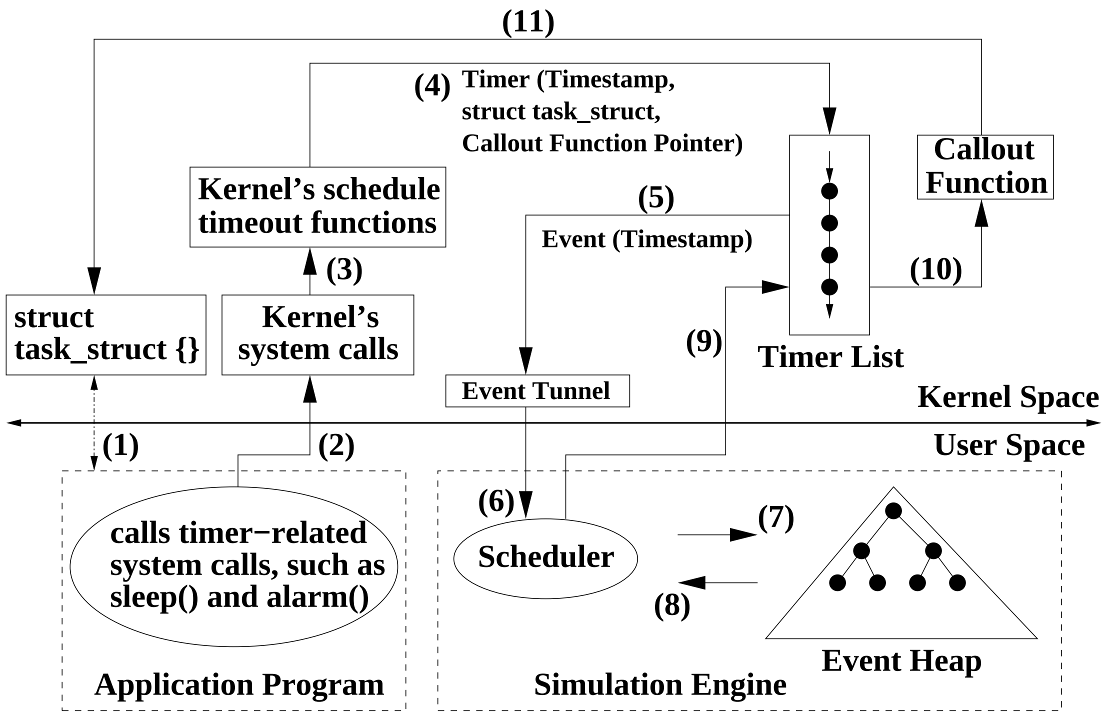


Figure 3.14: The execution procedure of the process timer event

is forked to run on a node by the simulation engine, the simulation engine calls a cutomized system call to fill the forked program's corresponding task_struct with that node's identification number. Each simulated node's identification number is greater than zero. Thus, in the kernel space, a task_struct with a non-zero node identification number represents a NCTUns process that must be executed based on the virtual time.

2. When a user space application program calls a timer-related system call, such as sleep() and alarm(), the corresponding kernel function is invoked to service the call.

3. In general, the timer-related kernel function calls some kernel's schedule time-out function to insert a timer into the kernel's original timer list that operates based on the real time. However, a timer list based on the virtual time is required by NCTUns. Thus, an additional timer list is created in the kernel space for those timers created by those NCTUns processes. In other words,

if a process' task_struct has a non-zero node identification number, all of the kernel timers originated from the process are inserted into the timer list that operates based on the virtual time.

4. A process timer carries the triggered timestamp based on the virtual time, the task_struct representing the calling process, and the function pointer pointing to the callout kernel function, such as process_timeout() and hrtimer_wakeup().

5. When a process timer is inserted into the timer list, a corresponding process timer event is created and inserted into the event tunnel's output queue. The event tunnel is a tunnel interface that is dedicated to deliver the kernel events to the simulation engine. The timer's triggered timestamp is carried by the process timer event so that the simulation engine can know when to trigger the timer.

6. Later on, the scheduler in the simulation engine checks the event tunnel's queue length (using memory-mapping technique) to see if any kernel event needs to be read from the event tunnel to the simulation engine. The scheduler does this check immediately after the virtual time is advanced, a packet is written into some tunnel interface from some interface module, a process timer event is triggered, or a TCP socket timer event is triggered. These timings are chosen because the above occurrences may cause consequent kernel events.

7. The process timer event retrieved from the event tunnel is inserted into the event heap and sorted by the triggered timestamp.

8. When the process timer event's triggered time is reached, it is retrieved by the scheduler. The scheduler also advances the virtual time based on the triggered timestamp of the event.

9. The scheduler calls a customized system call to the kernel. This system call makes the kernel to trigger those timers whose timestamps are equal to the current virtual time.

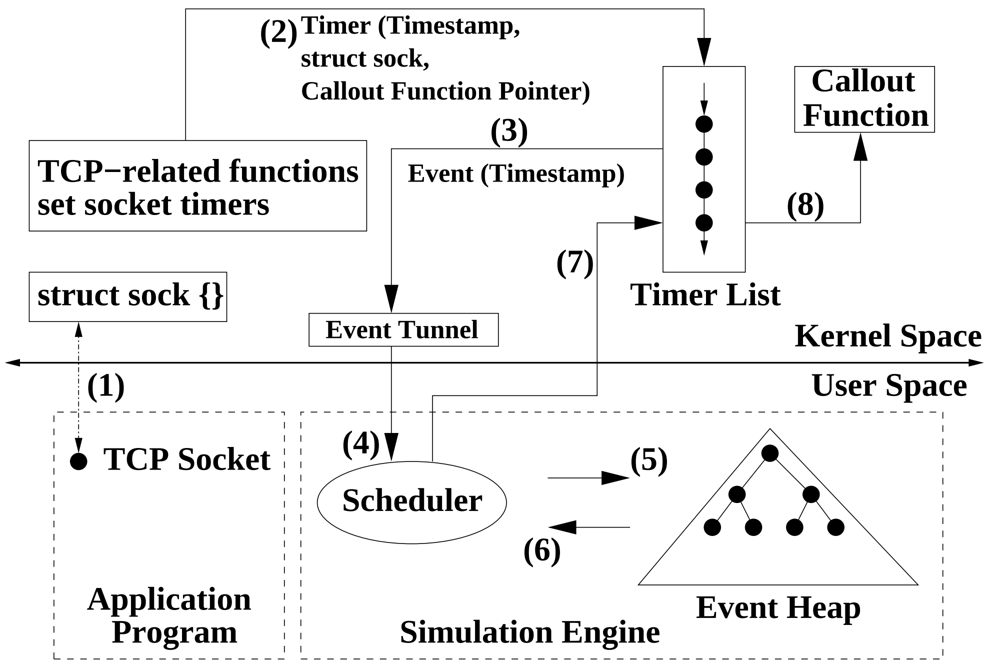


Figure 3.15: The execution procedure of the TCP socket timer event

10. The kernel invokes the function pointed by each timer's callout function pointer.

11. Within the callout function, a sleeping process represented by the task_struct is waked up, or a signal is sent to the process represented by the task_struct.

3.5.5 TCP Socket Timer Event

As mentioned before, the timers of TCP connections used in the simulated network need to be triggered based on the virtual time. This requirement is achieved by the TCP socket timer events. Figure 3.15 shows the generation and the execution of this kind of event. Each step is described below.

1. Within a user space application program, each opened socket has a corresponding data structure in the kernel space, called sock. The sock is the identification of a user space socket in the kernel space and is used in many network protocol work, such as specifying which network protocol is applied on the socket and providing socket send buffer and receive buffer to store

outgoing and incoming data segments. In NCTUns' implementation, some additional information are added into the sock, one of which is the identification number of a simulated node/device. During simulation, when a user space application program opens a socket, the kernel fills the node identification number in the socket's corresponding sock according to the application program's corresponding task_struct. Thus, in the kernel space, a sock with a non-zero node identification number represents a NCTUns socket that must be operated based on the virtual time.

2. In the kernel space, when some TCP-related functions set socket timers, these socket timers are inserted to the NCTUns timer list if the corresponding socket is a NCTUns socket. A socket timer carries the triggered timestamp based on the virtual time, the sock representing the corresponding socket, and the function pointer pointing to the callout kernel function, such as tcp_write_timer(), tcp_delack_timer(), and tcp_keepalive_timer().

3. When a TCP socket timer is inserted into the timer list, a corresponding TCP socket timer event is created and inserted into the event tunnel's output queue. The timer's triggered timestamp is carried by the TCP socket timer event so that the simulation engine can know when to trigger the timer.

4. Later on, the scheduler in the simulation engine checks the event tunnel's queue length to see if any kernel event needs to be read from the event tunnel to the simulation engine.

5. The TCP socket timer event retrieved from the event tunnel is inserted into the event heap and sorted by the triggered timestamp.

6. When the TCP socket timer event's triggered time is reached, it is retrieved by the scheduler. The scheduler also advances the virtual time based on the triggered timestamp of the event.

7. The scheduler calls a customized system call to the kernel. This system call

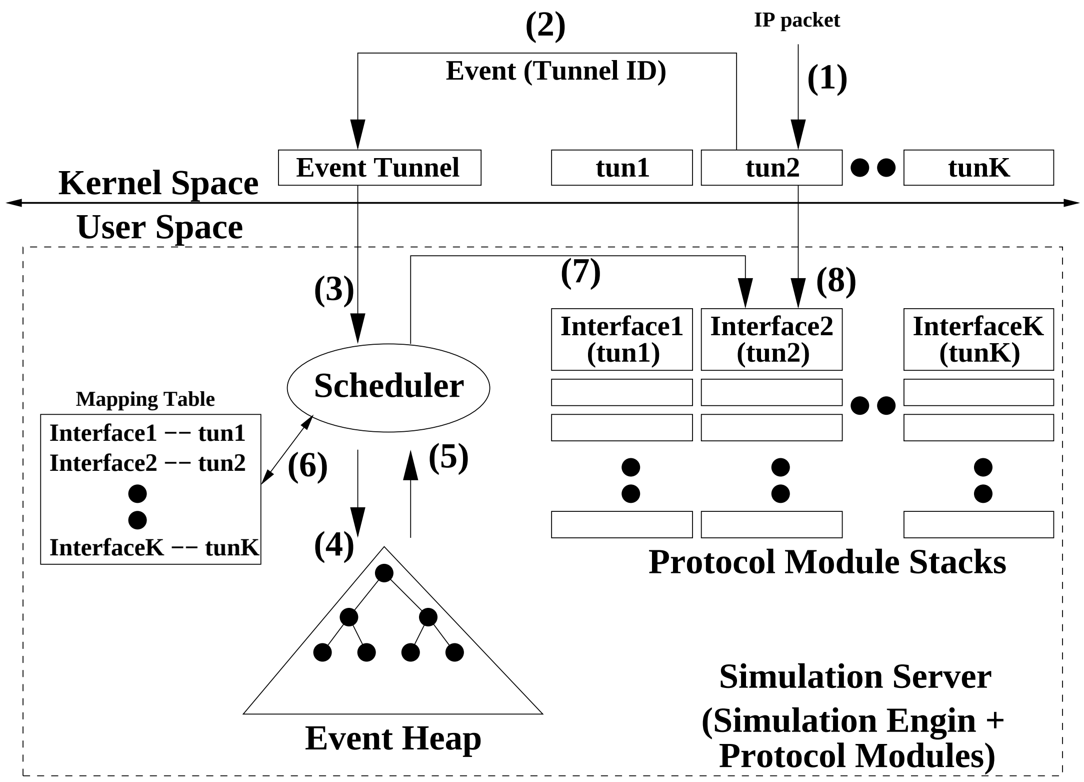


Figure 3.16: The execution procedure of the tunnel packet event

makes the kernel to trigger those timers whose timestamps are equal to the current virtual time.

8. The kernel invokes the function pointed by each timer's callout function pointer. Usually, in the case that a TCP socket timer expires, more TCP control packets (keep-alive timer), acknowledgment packets (delay-ack timer), or data packets (retransmission timer) are sent out by the kernel. These packets then cause more tunnel packet events to be inserted into the event tunnel.

3.5.6 Tunnel Packet Event

A tunnel packet event indicates exactly which tunnel interface has packet to be read by the simulation engine. Figure 3.16 shows the generation and the execution of this kind of event. Each step is described below.

1. Based on the kernel re-entering simulation methodology, an outgoing IP packet is sent to a tunnel interface.

2. Once an IP packet enters a tunnel interface, a tunnel packet event is created and inserted into the event tunnel's output queue. The identification number of that tunnel interface is carried by the event so that the simulation engine can know which tunnel interface generates the event.

3. Later on, the scheduler in the simulation engine checks the event tunnel's queue length to see if any kernel event needs to be read from the event tunnel to the simulation engine.

4. The scheduler sets the current virtual time to the tunnel packet event's timestamp and inserts it into the event heap.

5. After the scheduler inserts all other kernel events into the event heap. The inserted tunnel packet event is retrieved immediately (no virtual time progress) by the scheduler.

6. According to the tunnel identification number carried by the event, the scheduler searching a mapping table to find out the Interface module associated with the tunnel interface.

7. The scheduler calls the tunnel-reading function belonging to the found Interface module.

8. The queued IP packet is read from the tunnel interface to the associated interface module.

3.6 Port Number Translation

On a single machine, when multiple application programs try to bind their sockets to the same port number simultaneously, only the one who executes the bind() system call first will succeed and the others will fail. This is normal because one port number can only be used by one socket at any time on a single machine. However, this characteristic may fail some simulation scenarios. For example, when several Web servers want to bind to the port number 80, which is assigned by the Internet Assigned Numbers Authority (IANA) for providing web services, this simulation scenario can not be performed successfully. One solution to this problem is to let each Web server bind to different port number. Although it does not affect the simulation result, it makes a simulated network unnatural to the users. Thus, NCTUns provides another solution, called port number translation, to allow multiple application programs binding to the same port number. The idea of the port number translation is to translate the original port number used by a NCTUns socket to an unused port number in the kernel.

To achieve this goal, the kernel maintains a bitmap to record which port number has been used and which has not. During a simulation, suppose that an application program (say A) running on node i wants to bind to port number j, the kernel will find an unused port number (say k) and instead let application program A bind to port number k. The kernel then creates an association (node identification = i, original port number = j, remapped port number = k) and inserts it into a hash table.

With this arrangement, if an application program (say B) wants to send packets to application program A, application program B can use the port number originally used by application program A (i.e., j) as the destination port number. The port number translation process occurs at the destination node(s), not at the source node. When application program B sends a packet to application program A, before the packet reaches the destination node, the destination port number carried in the packet remains j, not k. Only after the packet reaches the destination node is its destination port number translated to k. Finding k is achieved by searching the hash table using the key pair (i, j), where j is readily available from the packet header. As for the value of i (the destination node's identification number), the kernel can obtain this information from the sock data structure that is mentioned in Section 3.5.5.

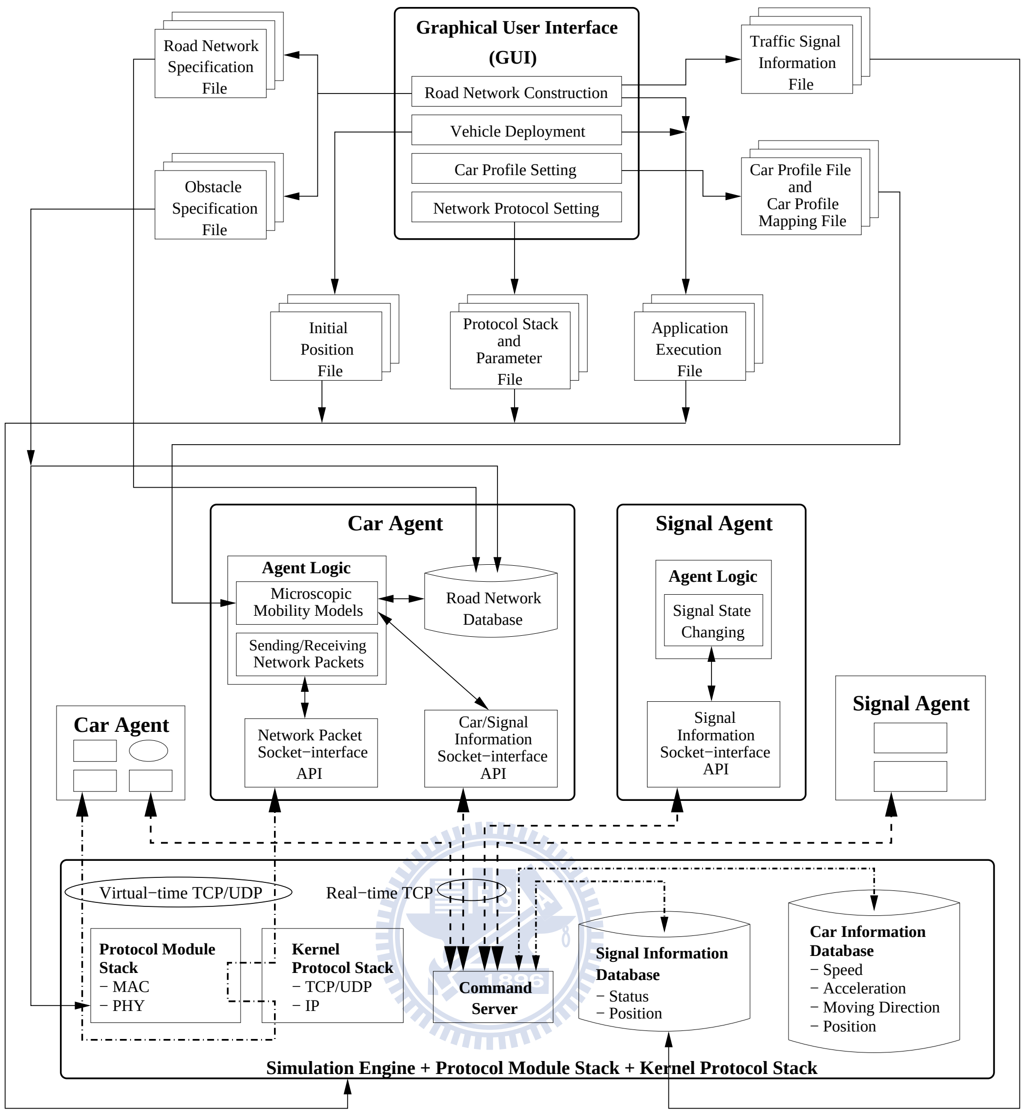
Translating the port number at the destination node(s), not at the source node, has two advantages. The first advantage is that it supports broadcast transfers on a subnet. If the translation is performed at the source node, only unicast transfers can be supported. Broadcasting a packet on a subnet to multiple application programs that bind to the same port but run on different nodes (e.g., routing daemons) will be impossible. The second advantage is that the tcpdump program can filter and capture packets based on the original port number instead of the translated one in a simulated network. This offers the users a natural way to use the tcpdump program in the simulated network.

Chapter 4

NCTUns Extension for Vehicular Network

In this chapter, we present the NCTUns extension for support the vehicular network simulation [32, 33, 34]. Two major directions are focused on the extension. The first one is the vehicular movement simulation. In the previous versions of NCTUns, a mobile node's moving path is generated automatically by the random waypoint mobility model or drawn manually by a user. No road network is provided for mobile nodes to move on it. Besides, no acceleration and deceleration is applied to a mobile node's movement. All of these need to be improved to support more realistic vehicular movement simulation. The other direction is to develop more wireless communication protocols for providing vehicular communication services, such as the IEEE 802.11p [1] and IEEE 1609 draft standards [2, 3, 4, 5].

In Figure 4.1, the extended architecture of NCTUns is depicted. From it, one can see that, based on the original architecture of NCTUns, several extensions are made to some NCTUns components to support the wireless vehicular communication network simulation. The extensions include the GUI extension, the simulation server extension, the car agent program, and the signal agent program. The functionalities and the design and implementation issues of these extensions are presented below.



Road Network Specification File
Obstacle Specification File
Graphical User Interface (GUI)
Road Network Construction
Vehicle Deployment
Car Profile Setting
Network Protocol Setting
Traffic Signal Information File
Car Profile File and Car Profile Mapping File
Initial Position File
Protocol Stack and Parameter File
Application Execution File
Car Agent
Agent Logic
Microscopic Mobility Models
Sending/Receiving Network Packets
Road Network Database
Network Packet Socket−interface API
Car/Signal Information Socket−interface API
Signal Agent
Agent Logic
Signal State Changing
Signal Information Socket−interface API
Car Agent
Signal Agent
Virtual−time TCP/UDP
Real−time TCP
Protocol Module Stack
− MAC
− PHY
Kernel Protocol Stack
− TCP/UDP
− IP
Command Server
Signal Information Database
− Status
− Position
Car Information Database
− Speed
− Acceleration
− Moving Direction
− Position
Simulation Engine + Protocol Module Stack + Kernel Protocol Stack

Figure 4.1: The extended architecture of NCTUns

4.1 Graphical User Interface Extension

In terms of the wireless vehicular communication network simulation, the GUI provides four major functions to help users easily generate the configuration files required by a simulation case. These files will be read by other components, that will be explained later, at the beginning of a simulation. In addition to the four major functions, the GUI can also play the animations of packet transmission/reception and vehicle movement, either during simulation or after simulation. This visual display of simulation results helps a user check the correctness of their network protocol

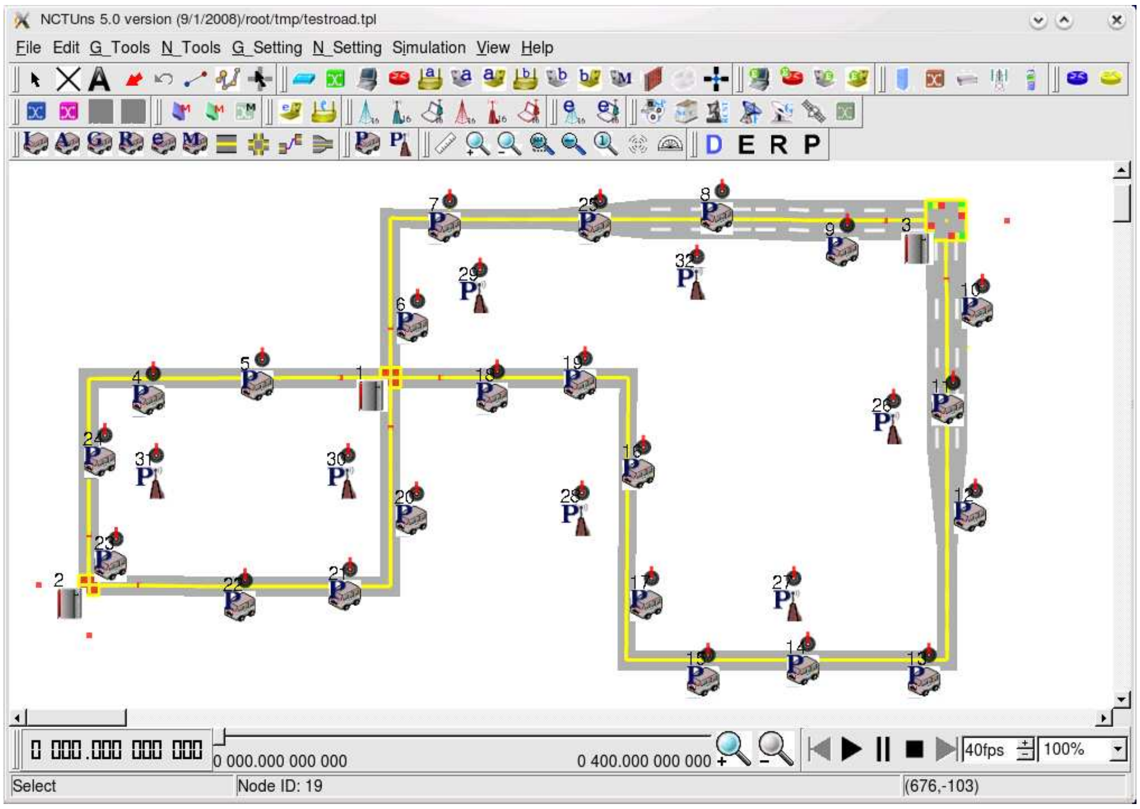

NCTUns 5.0 version (9/1/2008)/root/tmp/testroad.tpl
File Edit G_Tools N_Tools G_Setting N_Setting Simulation View Help
D E R P
0 000.000 000 000
0 400.000 000 000
40fps
100%
Select
Node ID: 19
(676,-103)

Figure 4.2: A snapshot of the GUI

designs and vehicle movement behavior.

4.1.1 Road Network Construction

The GUI provides many convenient facilities for users to construct their desired road network in a user-friendly environment. For instance, road deployment can be accomplished through a few operations of mouse clicking and drawing instead of editing a complex road network specification script file. The GUI supports several types of roads, including single-lane roads, multi-lane roads, crossroads, T-shape/L-shape roads, and lane-merging roads. A snapshot of the GUI with different type of roads is shown in Figure 4.2. Currently, the GUI constructs a T-shape/L-shape road from a crossroad by closing unused entrances. Four traffic lights are automatically placed at the four corners of a crossroad when a user puts a crossroad on the GUI.

A road network can be drawn manually by users or generated automatically by

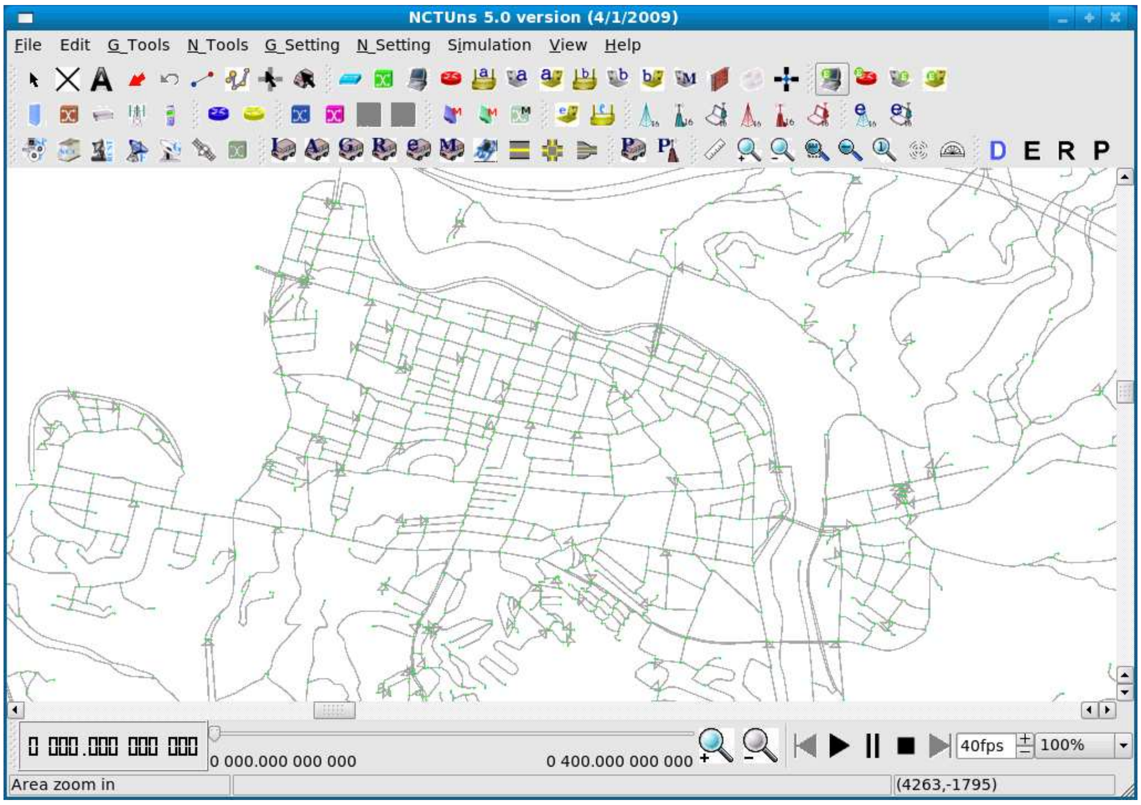

NCTUns 5.0 version (4/1/2009)
File Edit G_Tools N_Tools G_Setting N_Setting Simulation View Help
D E R P
0 000.000 000 000
0 000.000 000 000
0 400.000 000 000
40fps
100%
Area zoom in
(4263,-1795)

Figure 4.3: The facility of importing a real-world map

the GUI based on a shapefile [35], which is a digital vector storage format for storing geometric location and associated attribute information. The GUI can import a real-world map stored in a shapefile to construct a road network, like that shown in Figure 4.3. This capability can save much time required to manually construct a large-scale road network. Besides, because using real-world maps provides more realistic vehicular moving environments during simulation, the simulation results are more convincing.

After a user finishes his/her desired road network, the GUI automatically exports the corresponding road network information into four configuration files. The first one is the traffic signal information file. This file contains the information of each traffic signal deployed in the road network, including the type of each traffic signal (e.g., a traffic light or a speed-limit sign), the coordinate of each traffic signal, the facing direction of each traffic signal, and some type-specific data. For example, the

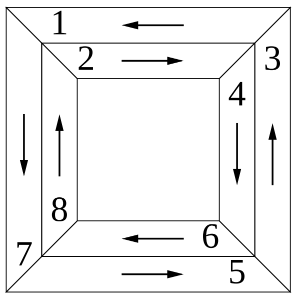


Figure 4.4: An example of road network representation

type-specific data of a traffic light include the initial state (e.g., green, yellow, or red) and the group identification number, which is assigned to the four traffic lights around a crossroad.

The second configuration file is the road network specification file. This file contains the information of each road block, including the coordinates of the four corners of a road block, the moving direction for vehicles to follow on a road block, which road blocks are chained to form a lane, which lanes are joined side by side to form a multi-lane road. Take the road network shown in Figure 4.4 as an example. In this square road network, road block 1, 3, 5, and 7 are chained to form a lane while road block 2, 4, 6, and 8 are chained to form another lane with reverse moving direction. These two lanes are joined side by side to form a square road.

The third configuration file is the application execution file. This file contains the information about when and on which node (e.g., a vehicle or a traffic light controller) to launch a specific application program. For example, for the set of four traffic lights around a crossroad, the GUI automatically arranges a signal agent application program to be run up on a traffic light controller to control the state changing of these traffic lights during simulation. The application's execution time and its corresponding traffic light controller node's identification number are exported by the GUI into the application execution file.

The fourth configuration file is the obstacle specification file. In the GUI, the view/radio obstacle is supported. An obstacle is a rectangular object that can block a vehicle driver's view, and/or totally block wireless signal or just reduce the power of the wireless signal passing through the obstacle. The information of each obstacle, including the coordinates of its two diagonal corners, the ability of blocking view, the ability of blocking wireless signal or reducing wireless signal power, and the user-specified value of signal power attenuation (if this ability is enabled), are exported by the GUI into the obstacle specification file.

4.1.2 Vehicle Deployment

After a user constructs his/her desired road network, he/she can put a vehicle at any place on the road network as the initial position of that vehicle. In case a user does not care about the initial position of each vehicle, the GUI also provides a facility for him/her to automatically deploy a specified number of vehicles on the road network at a specified average distance between two neighboring vehicles on the same lane. Through this facility, a user can also select which kind(s) of wireless radio(s) should be equipped with a vehicle. Currently, the provided choices include the IEEE 802.11b ad hoc mode radio, the IEEE 802.11b infrastructure mode radio, the IEEE 802.16e radio, the IEEE 802.11p radio, the GPRS radio, and the DVB-RCST satellite radio. The GUI exports the initial positions of all vehicles into the initial position file.

During simulation, a car agent application program is responsible for controlling its associated vehicle's movement. A car agent dynamically makes its moving decisions based on the vehicle's surrounding traffic and road conditions. When a vehicle is deployed on a road network, the GUI automatically arranges a car agent to be run up on the vehicle to control its movement. Each car agent's execution time and its corresponding vehicle node's identification number are exported into the application execution file by the GUI.

4.1.3 Car Profile Setting

A car profile defines the vehicular moving characteristics, including a vehicle's maximum speed, maximum acceleration, and maximum deceleration. Currently, five different car profiles are supported and each of them is recorded in a separate car profile file. The GUI provides a facility for a user to edit each car profile and assign the percentage of vehicles that will use a particular car profile during simulation. According to the assignment, the GUI automatically maps each vehicle to a specific car profile to match the percentage distribution. Finally, before a simulation starts, this mapping information is exported into the car profile mapping file. At the beginning of a simulation, a car agent first gets its assigned car profile by referring to the car profile mapping file. Then it opens the corresponding car profile file to read in the vehicular moving characteristic parameters.

4.1.4 Network Protocol Setting

When we presented the function of vehicle deployment before, we mentioned that several types of wireless radios can be selected to be equipped on a vehicle. NCTUns realizes the simulation of different kinds of wireless radios by simulating different network protocol stacks. In other words, a vehicle equipped with a wireless radio is associated with that radio's corresponding protocol stack. Each layer of a protocol stack is implemented as a protocol module in NCTUns. Thus, a protocol stack can be viewed as a series of protocol modules linked together. A useful facility is provided to help users easily select/replace protocol modules, such as mobile ad hoc routing protocol modules or buffer/queue management modules. In addition, the value of each parameter associated with each protocol module can be modified by users through this facility. The information about the used protocol stacks and module parameter values is exported by the GUI into the protocol stack and parameter file.

4.2 Simulation Server Extension

Before a simulation starts, all required configuration files are exported by the GUI for the simulation engine to read. At the beginning of the simulation, the simulation engine reads in the traffic signal information file, the initial position file, the protocol stack and parameter file, the obstacle specification file, and the application execution file. In addition, the simulation engine also needs to read in other configuration files. However, because they are not directly related to the settings for a wireless vehicular network simulation, they are not explained in this chapter.

The traffic signal information file is read by the simulation engine to build the signal information database. The attributes of each traffic signal are recorded in this database for car agents and signal agents to access during simulation. The initial position file is read by the simulation engine for setting each vehicle's initial position. During simulation, the latest positions of all vehicles are maintained in the car information database. The protocol stack and parameter file is read by the simulation engine for constructing each vehicle's network protocol stack(s) by linking a series of protocol modules and for initializing the parameters associated with each protocol module. In addition, the simulation engine reads the obstacle specification file to set up radio obstacles, which can block or reduce wireless signal power during simulation.

When the simulation engine reads the application execution file, it creates the application-executing events and inserts them into the event heap. These events are routine function events mentioned in Section 3.5.3. When an application-executing event is triggered, the simulation engine forks a car agent process or a signal agent process, depending on which application is specified. Like other applications forked by the simulation engine, a car agent process or a signal agent process can be forked at any time during simulation and then be killed at any time before the end of a simulation. The GUI allows a user to freely specify the start/end time of an application. When a car agent or a signal agent is created, its corresponding node identification number is recorded in its corresponding task_struct in the kernel space. The simulation engine calls a customized system call to set the task_struct. The node identification number can be retrieved from the kernel through another customized system call during simulation. We will present how a car agent process exploits this system call later.

In the simulation engine, a TCP-based command server is set up to periodically receive the request commands issued from a car agent or a signal agent. The simulation engine schedules a routine function timer to periodically read the command server's TCP socket(s) for any incoming request command during simulation. According to the type of a request command, the command server may store/retrieve data into/from the signal information database or the car information database. The operations between the command server and the car agent or signal agent will be described later.

Regarding the network protocol simulation, the network protocol stacks simulated in NCTUns include the direct use of the real-life TCP/UDP/IP protocol stacks in the Linux kernel and the simulations of MAC and PHY-layer protocols in the simulation engine. From the bottom-left part in Figure 4.1, one sees that different car agents exchange messages with each other over real-life TCP or UDP connections. These TCP/UDP connections are set up by the car agents using the standard POSIX socket-interface API's. The procedure of TCP/UDP packet transmission is described in Section 3.2. Besides, in order to support the wireless vehicular communication, NCTUns provides the 802.11p-related protocol modules for advanced ITS studies. These protocol modules are implemented based on the IEEE 802.11p [1] and IEEE 1609 draft standards [2, 3, 4, 5].

4.3 Car Agent

When a car agent is forked by the simulation engine, it reads the road network specification file to build its own road network database. This database will be queried by the agent logic during simulation. Instead of accessing a common road network database built by the simulation engine, each car agent builds its own road network database even though the content of each car agent's database is the same. This design is based on the performance consideration that the road network information query is very frequent and needs a fast response from the database. Thus, we trade the memory space for the simulation speed. In addition, the car agent reads the obstacle specification file to construct visual obstacles. The obstacle information is also stored in the road network database. Moreover, the car agent also reads the car profile mapping file to know which car profile is assigned to it and then reads the corresponding car profile file. As stated before, different car profiles define different moving characteristics with respect to maximum speed, maximum acceleration, and maximum deceleration. The difference and change on vehicular moving speed result in the changing of the vehicular ad hoc network topology during simulation. The topology changing affects the operations of network communication protocols and the performance of application programs. Generally speaking, the dynamics of topology is one of the considerable parameters when studying ad hoc network issues. A user can change this parameter by modifying the five car profiles and changing the car profile mapping ratio.

A car agent is an independent process that communicates with the command server located within the simulation engine through its car/signal information socket-interface API's, which are TCP-based. Besides, a car agent exchanges messages with other car agents through its network packet socket-interface API's, which are TCP- or UDP-based. The operations of packet exchange and the content carried in each packet are defined by a user to satisfy the requirements of an ITS application. NCTUns provides users with the flexibility to add or modify any operation in a car agent.

In Section 3.5.5, we mentioned that a socket opened by a NCTUns process is a NCTUns socket that operates based on the virtual time. That means, by default, both the car/signal information socket and the network packet socket operate based on the virtual time within a car agent. However, the former must operate based on the real time. This is because when a car agent starts a request-reply communication with the simulation engine through the TCP-based car/signal information socket, the virtual time maintained in the simulation engine is frozen to avoid unexpected virtual time advance. Until the car agent finishes its request-reply communication and signals the simulation engine about the finish, the simulation engine continues advancing the virtual time. During the request-reply communication, if the car/signal information socket is based on the virtual time, all the TCP timers malfunction because no timer expiration can be expected. In other words, a dead lock situation may occur when the request-reply communication relies on the virtual time advancing to finish and the virtual time advancing depends on the completion of the request-reply communication. In order to support the real time socket within a car agent, NCTUns provides a customized system call. When a car agent opens a socket, it calls the system call to clear (set to zero) the node identification number in the socket's corresponding sock data structure in the kernel space. After that, the kernel does not recognize the socket as a NCTUns socket and the socket starts operating based on the real time.

Figure 4.5 shows the communication between the car agent and the simulation server, including the virtual time based network packet transmission path and the real time based request-reply communication. The operations over network packet transmission path are the same as those presented in Section 3.2.3. For example, if Car Agent 1 (belonging to Mobile Host 1) wants to send a TCP/UDP network packet to Car Agent 2 (belonging to Mobile Host 2), it first sends out the packet through its network packet socket. The packet is then sent to Tunnel 1, which is Mobile Host 1's associated tunnel interface. After the simulation server moves the packet from Tunnel 1 to Mobile Host 1's Interface module, the packet goes through the MAC- and PHY-layer protocol simulations. Then, the packet is moved from Mobile Host 2's Interface module to Tunnel 2, which is Mobile Host 2's associated tunnel interface. Finally, Car Agent 2 obtains the packet through its network packet socket.

Regarding the real time based request-reply communication, it is a normal TCP connection built between a car agent and the command server located within the simulation server. Because the car agent and the simulation server are run on a single machine, the TCP connection between them passes through the loop back interface provided by most Linux operating systems. The loop back interface's IP address is usually 127.0.0.1. When a request packet is sent out by a car agent over the TCP connection, the kernel recognizes it as a normal TCP packet instead of a NCTUns TCP packet. Thus, the kernel does not translate this packet's IP address to the

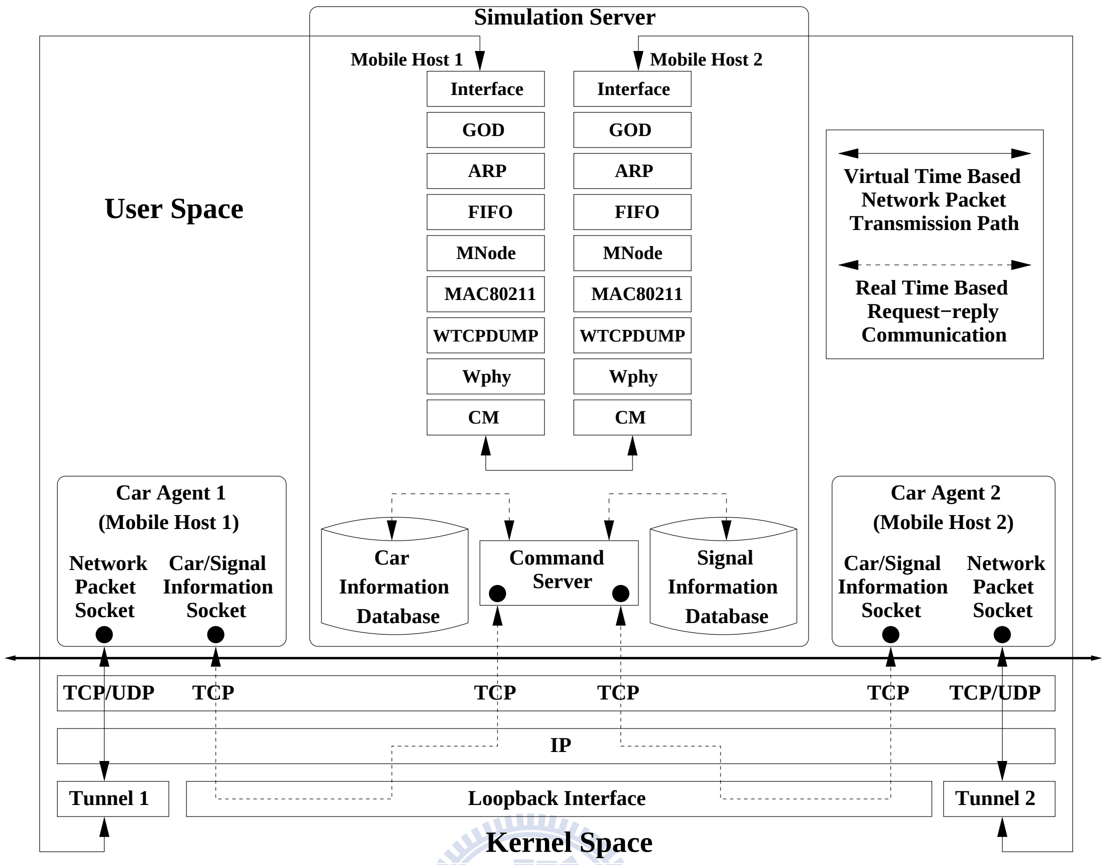


Figure 4.5: The communication between the car agent and the simulation server

source-destination-pair IP format and operates its corresponding TCP operations based on the real time. Because the packet's destination IP address is 127.0.0.1, at the IP layer the kernel routes it to the loop back interface and it is looped back immediately to the IP layer. When the command server obtains the request packet, it queries the car or signal information database for requested data and then sends a reply packet back to the car agent. The kernel treats the reply packet the same way as the request packet.

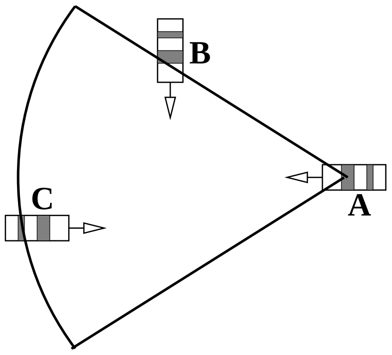
A car agent is the mobility controller of its associated vehicle simulated in the simulation engine. The agent logic in a car agent is the decision maker that determines when to make an action. After setting up the connection with the command server located within the simulation engine, the agent logic first retrieves the node identification number of the associated vehicle through the customized system call that is mentioned in Section 4.2. With the node identification number, the agent logic informs the command server to activate its associated vehicle in the simulation engine. This validates that vehicle's corresponding data within the car information database. Next, the agent logic queries for its initial/current position from the command server. The command server retrieves the queried position from the car information database and then sends it back to the agent logic. Before moving a vehicle from its initial position, the agent logic sets the current speed and current acceleration of the vehicle to zero.

During simulation, the agent logic periodically updates/accesses the car and signal information databases through the car/signal information socket-interface API's. For example, the agent logic may store/retrieve the current moving direction, current speed, current acceleration, current position of its associated vehicle, or retrieve the state of a traffic light that is nearest in front of its associated vehicle. The command server provides not only update/access services but also data-analyzing services. In the case that an agent logic tries to retrieve the state of a traffic light through the command server, the command server first retrieves the current position of the vehicle controlled by the agent logic from the car information database and all traffic lights' positions from the signal information database. Because the command server does not have the road network information, the nearest traffic light in front of the vehicle is obtained by doing some mathematical calculations based on the information provided by the agent logic. For example, the agent logic specifies the distance within which the nearest traffic light ahead of the vehicle should be returned by the command server. After the command server identifies the nearest traffic light, it sends the state of the traffic light to the agent logic. With this information, the agent logic can now control the vehicle to either drive across a crossroad if the traffic light is green or stop it at a crossroad if the traffic light is red.

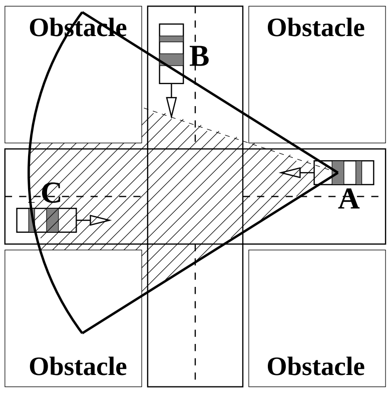
Moreover, the agent logic can retrieve the positions of the vehicles that is located within a specified area to its associated vehicle's position. Because the command server does the analysis considering only the vehicular positions, the agent logic has to analyze the retrieved positions again considering the view obstacles. If the straight line between the associated vehicle's position and one retrieved vehicle's position crosses any obstacle's rectangular area, then in the agent logic's viewpoint the retrieved vehicle should not be seen. For example, in Figure 4.6, the agent logic on behave of vehicle A's driver tries to find out the nearest vehicle that is located within a sector area in front of vehicle A. The agent logic has to specify the the angle of the sector (say, 60 degrees) and the radius of the sector (say, 50 meters) so that the command server can search those qualified vehicles and return them to the agent logic. Like that shown in Figure 4.6a, vehicle B and vehicle C are qualified vehicles. After the agent logic obtains the returned vehicles' positions, it checks if any obstacle locates between its associated vehicle (say, vehicle A) and those returned vehicles. Like that showed in Figure 4.6b, because the obstacles around the crossroad blocks the viewpoint from vehicle A, the visual area from vehicle A does not include the whole sector area but only the gray part within the sector area. As a result, vehicle B should not be seen by vehicle A and the nearest vehicle in front of vehicle A is finally determined to be vehicle C instead of vehicle B.

Although the command server can obtained the obstacle information from the obstacle specification file that is read by the simulation engine, it does not use the information when dealing with the above-mentioned request. This is because the request is a general-purpose request. A car agent usually issues the request to obtain those vehicles located within a specified area. Then, the car agent may combine the retrieved result with additional information to do advanced analyses, such as finding the nearest vehicle located on the right or left lane in front of or in the back of itself. The additional information may be the obstacle positions, the road block positions, and/or other information. Which kinds of information are required depend on what operation an agent logic wants to do. Thus, the command server does not do that for an agent logic but only provides general-purpose functions for an agent logic to call.

Other request commands are also provided for the agent logic to collect comprehensive information to make driving decisions. For example, the agent logic obtains the direction of a road ahead of the vehicle so that the vehicle can move in the



(a) The simulation engine returns a driver's viewpoint to the car agent only considering the relationship of vehicular position.



(b) The car agent determines a driver's viewpoint considering additionally the locations of all view obstacles based on the simulation engine's return.

Figure 4.6: Two phases to simulate a driver's viewpoint

correct direction on the road. Another example is that the agent logic obtains the information of neighboring lanes so that the vehicle can safely change lanes and/or overtake other vehicles. Yet another example is that the agent logic obtains the information of the crossroad ahead of the vehicle so that the vehicle can make a turn smoothly. On top of the default autopilot intelligence, a user can freely add more intelligence into the agent logic of a car agent. A user can also replace the default autopilot intelligence with more advanced autopilot intelligence.

4.4 Signal Agent

The signal agent is responsible for controlling the state of a traffic signal or a set of traffic signals. Many types of traffic signals can be supported in NCTUns. We take the traffic light as an example here. The signal agent for controlling a set of traffic lights is forked by the simulation engine with its group identification number. As stated before, the four traffic lights surrounding a crossroad are grouped together by assigning all of them a unique group identification number. A traffic-light signal agent with a given group identification number is responsible for controlling the state changes of the four traffic lights within the same group. Using the group identification number as an index, the agent logic of a traffic-light signal agent uses the signal information socket-interface API's to retrieve the type-specific data of each traffic signal, such as the initial state of a traffic light, from the command server. The command server retrieves the queried data from the signal information database and sends it back to the agent logic. During simulation, the agent logic periodically swaps the states of the two pairs of the traffic lights surrounding a crossroad. It uses the signal information socket-interface API's to update each traffic light's state in the signal information database. Like the car/signal information socket used within a car agent, the signal information socket used within a signal agent is also TCP-based and operates based on the real time. The communication between the signal agent and the command server is the same as the real time based request-reply communication depicted in Figure 4.5.

Chapter 5

Validation of Vehicular Mobility Control

A car agent is the mobility controller of its associated vehicle simulated in the simulation engine while a traffic-light signal agent is the status controller of its associated traffic light. The agent logic in a car agent communicates with the command server frequently during simulation to update its information, retrieve other vehicle's information, or retrieve the signal status of a traffic light. The agent logic in a traffic-light signal agent communicates with the command server periodically during simulation to update the signal statuses of a group of traffic lights. The information delivery delay is one potential factor that may affect the synchronization between a car/signal agent and the simulation engine. For example, a car agent may not obtain the latest locations of its neighboring vehicles because those vehicles' latest locations are not updated to the car information database in time. This problem may cause unexpected collisions while the car agent tries to overtake its neighboring vehicles or keep following its front vehicle. Another example is that the car agent may fail to stop its associated vehicle before the stop line at a crossroad because the traffic-light signal agent does not update the signal statues in time.

The results generated by a simulator should be validated with either the results derived from mathematical models (when the simulated system is simple enough to be modeled) or real-world data (when the simulated system is too complicated to be modeled) before it can be trusted. In this chapter, we test the car agent's mobility control with respect to two fundamental driving behaviors: the reaction to traffic light signal and the car following. In each test, we employ a vehicular mobility scenario that is composed of some uniform motions and uniform accelerated motions. In other words, one can easily verify the vehicular mobility with respect to the velocity and the displacement by using two fundamental formulas based on Newton's laws of motion:

$$\begin{cases} v_2 = v_1 + a(t_2 - t_1) \\ s = v_1(t_2 - t_1) + \frac{1}{2}a(t_2 - t_1)^2 \end{cases}$$

where v_1 is a vehicle's speed at time t_1, v_2 is the vehicle's speed at time t_2, a is the vehicle's acceleration/deceleration, and s is the vehicle's displacement from t_1 to t_2. By the formulas, we first derive the changes of position and velocity over time. Then, we record the same information during simulation. Finally, we compare these two results to observe if the theoretical driving behavior is carried out correctly by the simulation. The scenario and comparison results for each test are described below respectively.

5.1 Reaction to Traffic Light Signal

In the first test, we want to observe the car agent's reaction to the traffic light signal. Figure 5.1 illustrates the vehicular movement scenario adapted in this test. At the beginning, a vehicle is static and a traffic light stands in front of that vehicle. The distance between the vehicle and the traffic light is 200 m. Next, the vehicle starts moving with fixed acceleration of 1 m/s^2. When the vehicle reaches its maximum speed of 10 m/s, it keeps at this speed for a while. Later, once the vehicle sees that the traffic light signal turns to red and the distance between itself and the traffic light is equal to 30 m, it starts slowing down with fixed deceleration of 2 m/s^2 until it stops before the stop line that is drawn at where the traffic light stands.

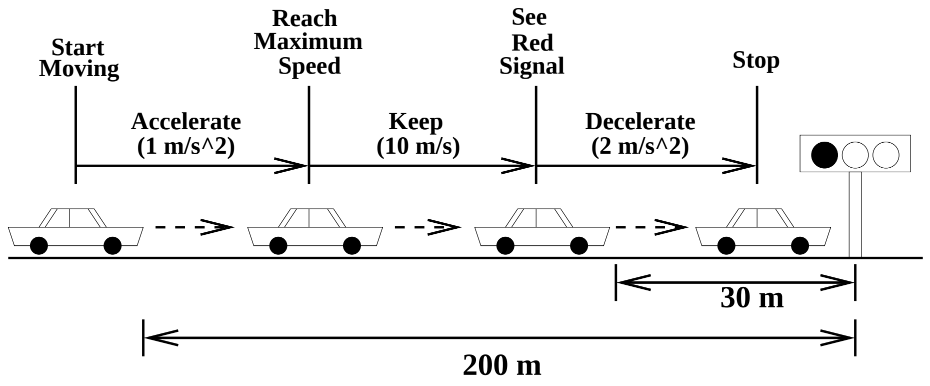


Figure 5.1: The scenario of mobility control test on the reaction to traffic light signal

Figure 5.2 shows the changes of the tested vehicle's velocity over time. The x-axis represents the elapsed time while the y-axis represents the tested vehicle's velocity. When comparing the two curves depicted in this figure, one of which is derived from the formulas and the other is obtained from the simulation, one can see that the controlled mobility curve is quite consistent with the theoretical mobility curve. Note that the simulated maximum speed is a little higher than the theoretical maximum speed of 10 m/s. This is because by default the car agent process periodically wakes up to check its surrounding road environment and make a moving decision every 100 ms. The default wake-up interval may cause the deviation if the car agent does not wake up just at some particular time. This deviation can be alleviated by reducing the wake-up interval at the cost of slowing down the simulation speed. In the test case, until the car agent wakes up and finds that its speed is larger than 10 m/s, it stops accelerating and keeps at the current speed. The tiny deviation also occurs at the time when the car agent finds that the distance between the traffic light and itself is less than 30 m and starts slowing down. Except the above-mentioned tiny deviation, the consistency shown in Figure 5.2 confirms that the car agent can correctly control its associated vehicle simulated in the simulation engine to react to seeing a red traffic light signal. In other words, it shows that the signal agent updates its signal status to the signal information database in time. Also, the car agent's position retrieval and signal status retrieval are completed in time.

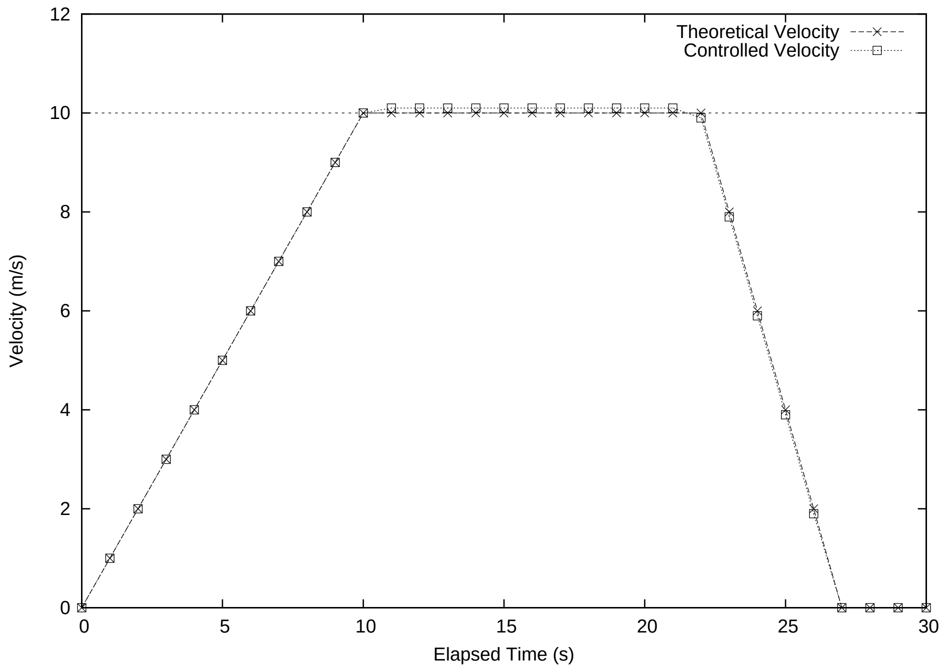


Figure 5.2: The changes of the tested vehicle's velocity over time

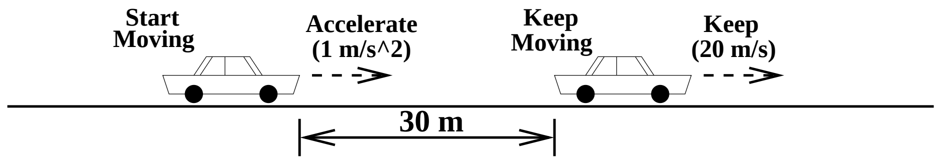
5.2 Car Following

In the second test, we want to observe the car agent's car following behavior. Figure 5.3 and Figure 5.4 illustrate the vehicular movement scenario adapted in this test. Each movement change is described as follows.

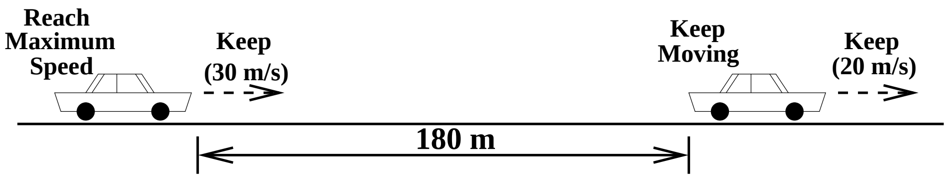
1. Figure 5.3a: At the beginning, the rear vehicle starts moving from static and the front vehicle keeps moving at 20 m/s. The initial distance between the rear vehicle and the front vehicle is 30 m. The rear vehicle's acceleration is 1 m/s^2 and its maximum speed is 30 m/s.

2. Figure 5.3b: After 30 s from the beginning, the rear vehicle reaches its maximum speed (say, 30 m/s) and stays at that speed. Currently, the distance between the two vehicles is 180 m. Because the rear vehicle's speed is higher than the front vehicle's, the rear vehicle will gradually approach the front vehicle when time goes by.

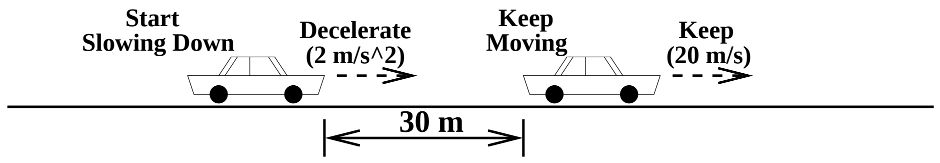
3. Figure 5.3c: After 45 s from the beginning, the rear vehicle finds that the



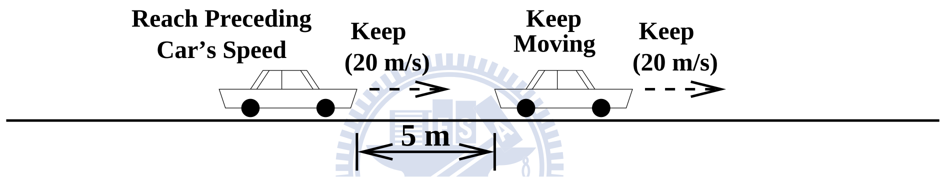
(a) At the beginning



(b) After 30 s from the beginning



(c) After 45 s from the beginning



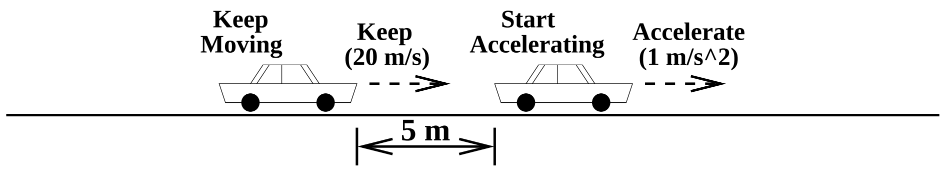
(d) After 50 s from the beginning

Figure 5.3: The scenario of mobility control test on the car following behavior (part 1)

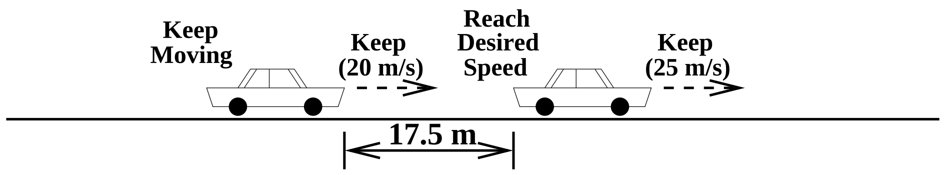
distance between itself and the front vehicle is 30 m and starts slowing down to avoid colliding with the front vehicle. Its deceleration is 2 m/s^2 and its desired speed is the front vehicle's current speed (say, 20 m/s).

4. Figure 5.3d: After 50 s from the beginning, the rear vehicle reaches its desired speed (say, 20 m/s) and keeps at that speed. Currently, the distance between the rear vehicle and the front vehicle is 5 m.

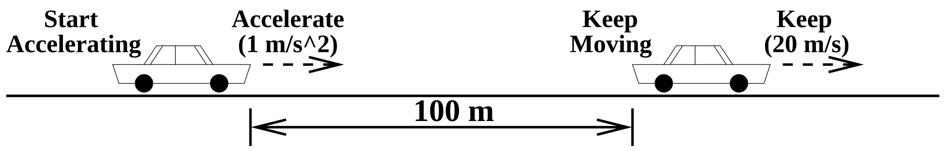
5. Figure 5.4a: After 60 s from the beginning, the front vehicle starts speeding up. Its desired speed is 25 m/s and its acceleration is 1 m/s^2. The current distance between the rear vehicle and the front vehicle is still 5 m. This



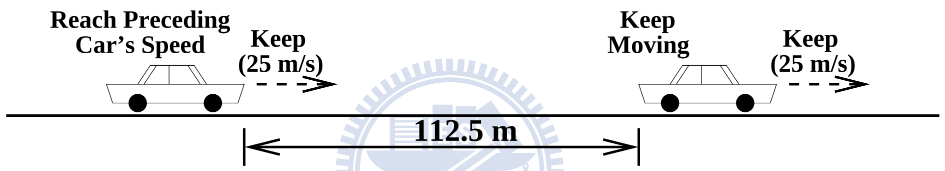
(a) After 60 s from the beginning



(b) After 65 s from the beginning



(c) After 81.5 s from the beginning



(d) After 86.5 s from the beginning

Figure 5.4: The scenario of mobility control test on the car following behavior (part 2)

speed-up will gradually increase the distance between the two vehicles when time goes by.

6. Figure 5.4b: After 65 s from the beginning, the front vehicle reaches it desired speed (say, 25 m/s) and keeps at that speed. Currently, the distance between the two vehicles is 17.5 m.

7. Figure 5.4c: After 81.5 s from the beginning, the rear vehicle finds that the distance between itself and the front vehicle is 100 m and starts speeding up. Its desired speed is the front vehicle's current speed (say, 25 m/s) and its acceleration is 1 m/s^2.

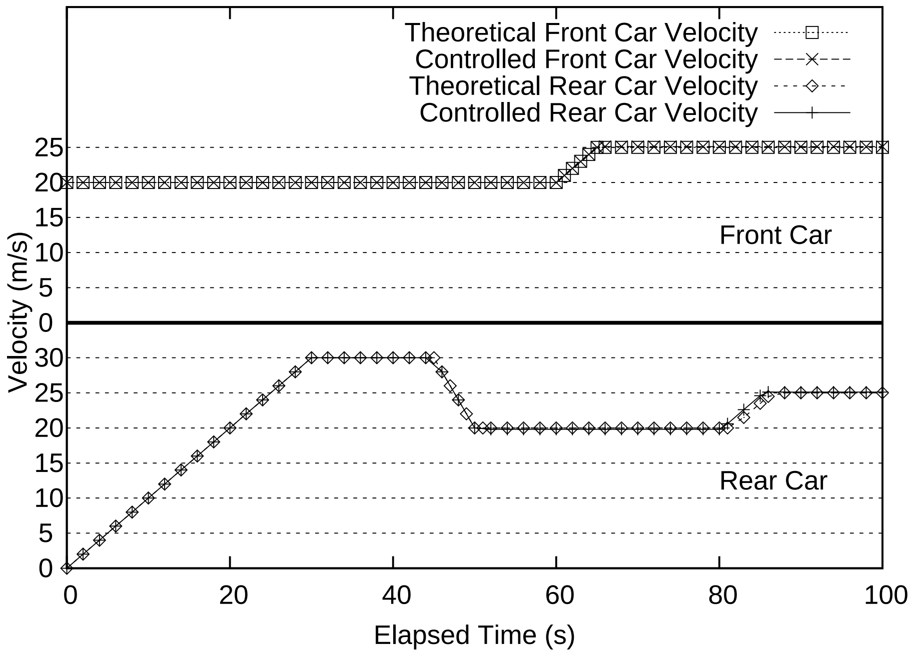


Figure 5.5: The changes of the tested vehicles' velocities over time

8. Figure 5.4d: After 86.7 s from the beginning, the rear vehicle reaches its desired speed (say, 25 m/s) and stays at that speed. Currently, the distance between the two vehicles is 112.5 m.

Figure 5.5 shows the changes of the front vehicle's and the rear vehicle's velocities over time. The x-axis represents the elapsed time while the y-axis represents the tested vehicle's velocity. From this figure, one can first see that the front vehicle's theoretical velocity curve and controlled velocity curve are quiet consistent, and so are the rear vehicle's. Next, one can see that, for either the front vehicle or the rear vehicle, the velocity changes are consistent with the vehicular movement scenario described above. The tiny deviation mentioned in Section 5.1 also exists here. Except the deviation, the consistency shown in Figure 5.5 shows that the car agent can correctly control its associated vehicle simulated in the simulation engine under a car following situation. In other words, the vehicular position update and retrieval are completed in time during simulation.

Chapter 6

Performance Evaluation

In this chapter, we evaluate the simulation performance of NCTUns with respect to the elapsed time and the physical memory usage of each run-time component, including the simulation server, the car agent, and the signal agent. Two important system parameters are investigate: the number of road blocks and the number of vehicles deployed in a simulated wireless vehicular communication network. The simulation machine used in the evaluations is a desktop computer equipped with a P4 2.53 GHz CPU and 1 GB RAM. The total time to be simulated for each simulation case is set to 500 seconds. Each simulation case is run 10 times to collect the average results.

The topology of the road network is a 6x6 grid network, as shown in Figure 6.1. The edge length of each grid is 1 Km. Thus, the simulated field covers an area of 36 Km^2. The edge of a grid is a road that is formed by four lanes. Each lane is in turn formed by a single or multiple road block(s). In the grid road network, a crossroad is positioned at each intersection.

The car profile settings and distribution shown in Table 6.1 are applied to all simulation cases. Currently, we use the maximum (allowable) speed of a vehicle as the desired maximum speed for its driver. The maximum speed settings and distribution reflect the normal driving speeds (from 40 Km/hr to 80 Km/hr) in a urban area, where the road network is similar to the used grid topology. The range

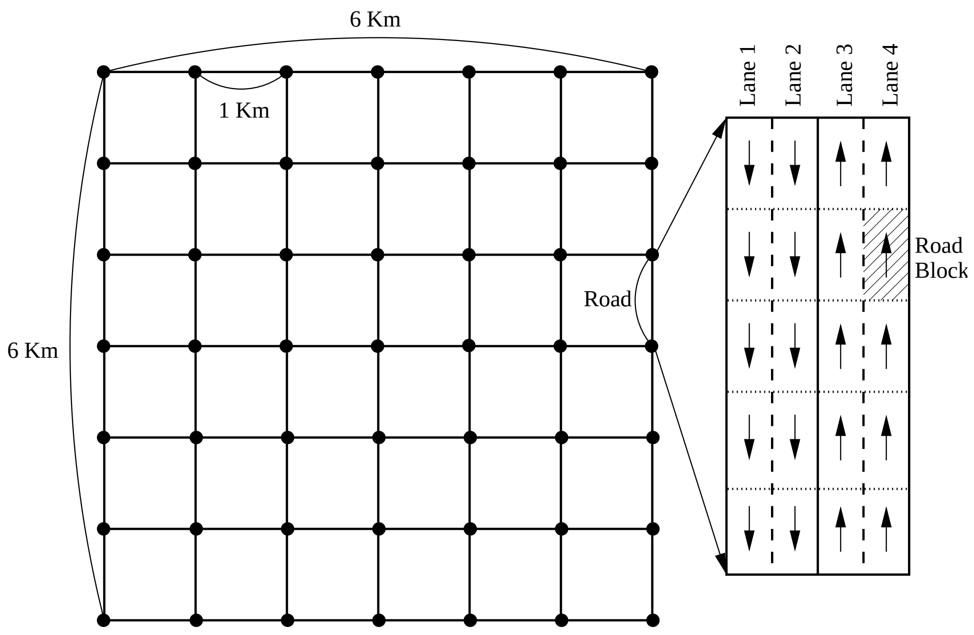


Figure 6.1: The topology of the grid road network used for performance evaluation

Table 6.1: Car profile settings and distribution

Profile Number	1	2	3	4	5
Percentage	10%	35%	35%	15%	5%
Max Speed (m/s)	11	14	17	19	22
Max Acceleration (m/s^2)	1.1	1.4	1.7	1.9	2.2
Max Deceleration (m/s^2)	3.67	4.67	5.67	6.33	7.33

of the used maximum accelerations reflects the characteristic of a normal vehicle that can accelerate from 0 Km/hr to its maximum speed in about ten seconds. Finally, the range of the used maximum decelerations reflects the characteristic of a normal vehicle that can decelerate from its maximum speed to 0 Km/hr in about three seconds.

Regarding the network communication scenario, the car agent running on each vehicle is programmed to broadcast a 1084-byte UDP packet (1056 bytes for the data payload, 20 bytes for the IP header, and 8 bytes for the UDP header) once per second to the vehicles located within its wireless transmission range. A simplified wireless PHY-layer module is used, which uses 250 and 550 meters as the wireless transmission range and interference range, respectively. The used wireless MAC-layer module is based on the IEEE 802.11b standard. No routing protocol is adopted because all packet transmissions are based on broadcast.

6.1 Number of Road Blocks

In the first evaluation, in total 200 vehicles are deployed in each of the five simulation cases. Without changing the topology shown in Figure 6.1, we vary the number of road blocks on each lane from 1 to 5 in the five cases by purposely using different sizes of road blocks. Therefore, we deploy 385, 721, 1,057, 1,393, and 1,729 road blocks in these cases, respectively. The total number of road blocks deployed in a case can be calculated by the formula: [$Number\ of\ Crossroad$ + ($Number\ of\ Roads$ $*$ $Number\ of\ Lanes\ on\ each\ Road$ $*$ $Number\ of\ Road\ Blocks\ on\ each\ Lane$)].

Because the size of the area of the road network is kept the same in all cases, the density of vehicles on the road network is the same in all cases. This keeps the simulation overhead of broadcasting UDP packets about the same in all cases. Increasing the number of road blocks will increase the size of the road network database. This may increase the memory space usage of each car agent as it needs to store every road block information into its road network database. This may also increase the number of query in a car agent to searches for the information of a road block. For example, every time when a vehicle reaches the end of a road block, it needs to get the information of the new road block ahead of it. Thus, when the number of road blocks increases, we expect to see increased physical memory usage for a car agent and increased time for running a simulation (i.e., the elapsed time).

The results shown in Table 6.2 confirm the above conjectures. One sees that the elapsed time increases slightly as the number of road blocks increases. In addition, as expected, the physical memory usage of a car agent increases slightly as the number of road blocks increases. The slight performance change between two adjacent cases indicates that the number of road block has little impact on the simulation performance of NCTUns.

Table 6.2: Elapsed time and physical memory usage in each case with different number of road blocks

Simulation Settings					
Number of Vehicles	200				
Number of Crossroads	49				
Number of Roads	84				
Number of Lanes on each Road	4				
Number of Road Blocks on each Lane	1	2	3	4	5
Total Number of Road Blocks	385	721	1,057	1,393	1,729
Simulation Results					
Elapsed Time (min)	17.3	20.1	20.6	21.3	21.9
Physical Memory Usage of the Simulation Server (MB)	15.22	15.85	15.87	15.85	15.85
Physical Memory Usage of a Car Agent (MB)	2.14	2.20	2.27	2.34	2.75
Physical Memory Usage of a Signal Agent (MB)	1.08	1.08	1.08	1.08	1.08

6.2 Number of Vehicles

In the second evaluation, in total 1,729 road blocks are deployed in each of the five simulation cases. We deploy 250, 350, 450, 550, 650, 750, and 850 vehicles in these cases, respectively.

Because the total number of road blocks is the same in all cases, a car agent's run-time overhead on the road network database, such as the number of queries to the database and the memory consumption for storing the database, is the same in all cases. Increasing the number of vehicles will increase the vehicle density on the road network. Therefore, the simulation server needs to spend more time on broadcasting and receiving more UDP packets. Also, the simulation server needs to consume more memory space for storing these UDP packets during simulation.

Table 6.3: Elapsed time and physical memory usage in each case with different number of vehicles

Simulation Settings							
Number of Road Blocks	1,729						
Number of Vehicles	250	350	450	550	650	750	850
Simulation Results							
Elapsed Time (min)	55.8	108.7	173.2	244.6	341.9	456.1	525.3
Ratio of Simulated Time to Elapsed Time	1:7	1:13	1:21	1:29	1:41	1:55	1:63
Physical Memory Usage of the Simulation Server (MB)	29	37	46	57	70	85	110
Physical Memory Usage of a Car Agent (MB)	2.75	2.75	2.75	2.75	2.75	2.75	2.75
Physical Memory Usage of a Signal Agent (MB)	1.08	1.08	1.08	1.08	1.08	1.08	1.08

Thus, it is expected to see increased time for running the simulation and increased physical memory usage for the simulation server.

The results shown in Table 6.3 confirm the above conjectures. One sees that the elapsed time and the physical memory usage of the simulation server increase with the number of vehicles deployed on the road network. In addition, the case with 850 vehicles requires about 525 minutes to complete the 500-second simulation. The ratio of simulated time to elapsed time is about 1:63. In other words, in this case, the advance of the simulated virtual time is 63 times slower than that of the real time.

Chapter 7

Future Work

In order to increase the usability of NCTUns in terms of the support for wireless vehicular communication network, several improvements are still underway and will be available in the future releases.

- **Traffic Signal**

  Currently, the only supported traffic signal type is traffic light. Other different types of traffic signals should also be provided to support more realistic road network environments, such as the stop sign and speed limit sign. This improvement includes not only the GUI functionalities for setting up these traffic signals in a road network, but also the car agent's driving logic that has to react to these newly supported traffic signals.

- **Centralized Traffic Light Control**

  Regarding the current implementation of the signal agent that controls a group of traffic lights, a signal agent does not cooperate with other signal agents during simulation. However, in the real life, the centralized traffic light control is common, especially on those roads with heavy traffic load. The goal of the centralized traffic light control is to control the signal statuses of a set of traffic light groups to efficiently direct heavy traffic flows. To support this kind of control, NCTUns may involve a new communication channel among signal agents or a new type of signal agent that controls a set of traffic light groups instead of just one group. Besides, more complex control mechanism has to be provided for users to specify their desired operations on the traffic light control.

- **Road Infrastructure**

  Currently, only some fundamental road types are supported for users to construct road networks. Although many road network environments can be constructed by these fundamental roads, other road infrastructures are still required for more accurate road network representation and reflecting the real-world environments, such as the circular roads, the left-turn dedicated lane at intersections, the toll stations on freeways, and so on. These road infrastructures are required because they affect the vehicular movement during simulation. With more complex road networks being supported, the microscopic vehicle mobility model also has to be improved to support new road network environments. For example, when approaching a toll station on a freeway and finding that there is a long waiting line ahead, some drivers not only slow down their vehicles but also change to an adjacent lane with a shorter waiting line. It is clear that a more intelligent microscopic vehicle mobility model is required to simulate these behaviors.

- **Platoon of Vehicle**

  According to the studies published in [36] and [37], one finding shows that in VANET a viable multi-hop routing path usually has limited length in hop count (say, less than ten hops). The result indicates that some VANET applications are only viable when they are applied on the moving vehicles always close to each other. A platoon of vehicle fits this requirement. Thus, the simulation of platoon movement is required. Within a platoon, a vehicle has to follow its front vehicle (if any) and also wait its rear vehicle (if any). Besides, all vehicles belonging to the same platoon have to move along the same path.

The support of the platoon simulation can provide those platoon applications a very useful testing environment.

- **Network Protocol and Channel Model**

  As of writing this disseration, the IEEE 802.11p/1609 communication technology proposed for wireless access in vehicular environments has been supported in the NCTUns platform. Besides, the IEEE 802.16e communication standard proposed for mobile WiMAX environments has also been supported. Based on the support of vehicular communication protocols, we have conducts some research and the results have been published [38, 39]. Currently, the IEEE 802.16j standard proposed for multi-hop relay in WiMAX environments is under development, including the implementation of new MAC- and PHY-layer modules. The requirements for emerging network protocols and wireless channel models are endless. NCTUns will keep updating its simulation capabilities with new communication technologies.

- **Module-based Vehicular Mobility Control**

  As that shown in Figure 4.5, a car agent is an independent user space process. As the number of vehicles deployed in the road network increases, the number of car agent processes also increases during simulation. Thus, during simulation the CPU context switching between the simulation server process and the car agent processes becomes more frequent. This is one considerable overhead with respect to the simulation speed. Besides, the increasing frequency of database accessing from the car agents to the command server results in more communication overhead. This considerable overhead includes the request/reply data copy between the user space and the kernel space and the TCP protocol operations to deliver the request/reply packets.

  In order to eliminate the overhead, a new design to replace the car agent is proposed, called the module-based vehicular mobility control. The concept of the new design is to transplant a car agent into a protocol module located in

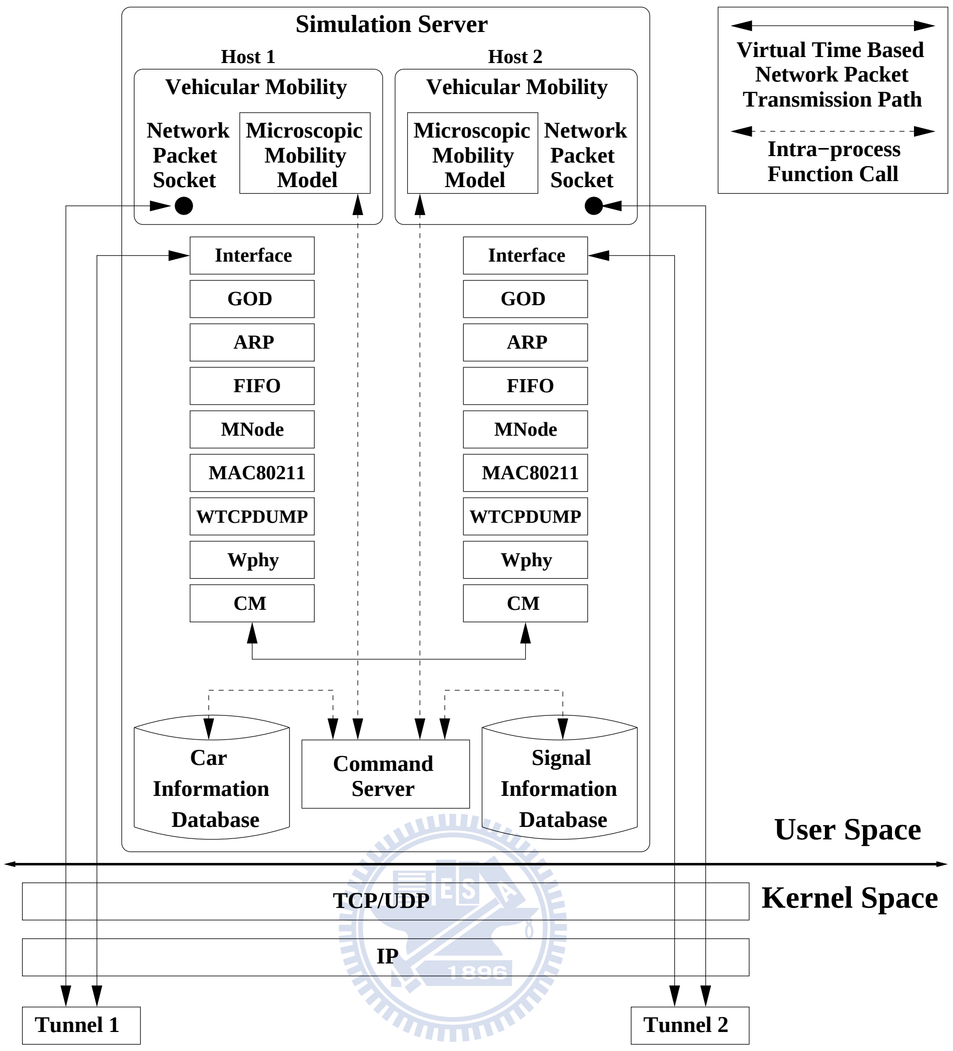


Figure 7.1: The module-based vehicular mobility control

the simulation server. In Figure 7.1, one can see that no car agent exists in this new architecture. Instead, a new type of protocol module, called Vehicular Mobility module, is placed on the top of each mobile host's Interface module. Comparing Figure 7.1 with Figure 4.5, the virtual time based network packet transmission path is almost the same except that the packet sender/receiver is now a protocol module instead of a car agent process. In addition, no real time based request-reply communication is needed. Instead, it is replaced with the intra-process function call within the simulation server.

Using the module-based design, we can expect significant speed-up on the simulation speed. Besides, this design can be applied on a signal agent. In this case, a signal agent is transplanted into a protocol module, called Signal module, located in the simulation server. The module-based signal control can facilitate the above-mentioned centralized traffic light control because it is convenient to synchronize a set of traffic light groups when they are all located within the simulation server process.

Because the simulation server process does not represent any simulated node, the node identification number in its corresponding kernel-space task_struct is zero. As stated in Section 3.5.5, when a socket is opened within the simulation server process, the node identification number in the socket's corresponding kernel-space sock is also zero. In other words, the kernel does not recognize the simulation server as a virtual time based NCTUns process. Thus, by default, any socket opened in the simulation server process is based on the real time. However, in Figure 7.1, the network packet sockets should be based on the virtual time. To solve this problem, the simulation server should masquerade as if it belongs to some simulated node before opening a network packet socket. For example, in Figure 7.1, before the simulation server opens the network packet socket in Mobile Host 1's Vehicular Mobility module, it first calls a customized system call to set its tasks_struct's node identification number to Mobile Host 1's node identification number. The system call is the same as that mentioned in Section 3.5.4. When the network packet socket is opened, its corresponding sock's node identification number is equal to Mobile Host 1's node identification number. The kernel then treats it as a NCTUns socket and operates it based on the virtual time. After the socket is opened, the simulation server has to call the same system call to set its tasks_struct's node identification number back to zero.

In addition, the protocol module programming is a bit different from the independent process programming. For example, a process can call a blocking read() or write() system call to read from or write to an opened socket, but this kind of system call can not be used within a protocol module. This is because when a blocking system call is called within a protocol module, the whole simulation server may be blocked on that socket and be put to sleep by the kernel. The blocked socket may be waiting for some incoming data that is supposed to be sent from another socket that is also opened in another protocol module. This results in a dead lock situation. Another example is that calling the sleep() system call, based on the real time, is useless within a protocol module. Because when the whole simulation server is put to sleep by the kernel, no one is responsible to advance the virtual time. Thus, when the simulation server wakes up, the simulated virtual time is still the same as that when it was put to sleep. Yet another example is that no infinite loop can be used within a protocol module. The infinite loop makes the whole simulation engine get stuck in the module forever.

The above-mentioned forbidden usages of some function calls or programming syntax can be solved by using some replacement. For example, one can use the non-blocking system calls to replace the blocking system calls. One can use the protocol module event or timer, mentioned in Section 3.5.2, to replace the sleep() system call. In addition, one can also use the protocol module timer to imitate the infinite loop. Although in some case the imitation of the agent's processing procedure can not be exactly the same as that of the agent process, we think these alternative implementations are still acceptable for obtaining significant simulation speed improvement.

Chapter 8

Conclusion

In this dissertation, we present a software simulator, called NCTUns. NCTUns is an open source tool that integrates communication/network simulation with road/traffic simulation for wireless vehicular communication network research.

We first introduce the original architecture of NCTUns, which is designed to support communication protocol simulations. In addition to the NCTUns' major components and the execution procedure of these components, several design and implementation issues are illustrated, including the kernel re-entering simulation methodology, the specific routing scheme, the IP address translation, the discrete-event simulation engine, and the port number translation.

Based on the original architecture, we then present the extended architecture of NCTUns, which is developed to further support vehicular network simulations. The design and implementation issues on these extensions are illustrated, including the GUI extension, the simulation server extension, the car agent, and the signal agent. Besides, we also explain the utilities provided to facilitate the operations of conducting simulations, such as road network construction, vehicle deployment, car profile setting, and network protocol setting.

Next, we test the vehicular mobility control with respect to the reaction to traffic light signal and the car following behavior. Our tests show that the simulated vehicular moving behavior is consistent with that derived from the theoretical formulas.

This confirms that the information update between a car agent and the simulation engine or between a signal agent and the simulation engine are completed in time.

Finally, we evaluate the simulation speed and the physical memory usage under different simulation scales. Our evaluations shows that the increase of the number of deployed roads does not affects the simulation speed significantly, while the increase of the number of deployed vehicles does. The simulation results show that in the case with 850 vehicles, the advance of the simulated virtual time is 63 times slower than the advance of the real time.

Since NCTUns was first released in 2002, we have continually updated its capabilities to support new communication protocols and more user-friendly GUI facilities. Besides, we have kept maintaining the NCTUns forum [41] to interact with NCTUns users, such as answering the questions about the usage of NCTUns, inspecting the reported bugs and fixing them if they do exist, etc. As of May 25, 2009, according to the download user database, more than 14,603 people from 132 countries have registered at the NCTUns Web site [9] and downloaded it to use. Some of these users also have published their research results on international journal or conference papers [40]. The above achievement shows that NCTUns has become a reliable and useful research tool for many researchers around the world.

Regarding the support of the vehicular network simulation, we have expanded the usability of NCTUns to the wide-ranging ITS research fields. After the capabilities of the vehicular network simulation were released, several users have contacted us asking the usage and functionalities of the NCTUns vehicular network simulation. The users' requirements and feedback give us stronger motivation and practical developing directions to keep improving NCTUns. We believe that in the future NCTUns will be more and more useful for users to conduct wireless vehicular communication network research.

Bibliography

[1] "IEEE 802.11p/D3.0: Draft Standard for Information Technology - Telecommunications and information exchange between systems - Local and metropolitan are networks - Specific requirements - Part 11, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications," IEEE Standards Activities Department, July 2007.

[2] "IEEE 1609.1 Trial - Use Standard for Wireless Accesses in Vehicular Environments (WAVE) - Resource Manager," IEEE Vehicular Technology Society, October 2006.

[3] "IEEE 1609.2 Trial - Use Standard for Wireless Accesses in Vehicular Environments (WAVE) - Security Services for Applications and Management Messages," IEEE Vehicular Technology Society, October 2006.

[4] "IEEE 1609.3 Trial - Use Standard for Wireless Accesses in Vehicular Environments (WAVE) - Networking Services" IEEE Vehicular Technology Society, October 2006.

[5] "IEEE 1609.4 Trial - Use Standard for Wireless Accesses in Vehicular Environments (WAVE) - Multi-channel Operation," IEEE Vehicular Technology Society, October 2006.

[6] "IEEE 802.16e-2005 - Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems - Amendment 2: Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands," IEEE Standard for Local and metropolitan area networks, February 2006.

[7] S.Y. Wang, **C.L. Chou**, C.H. Huang, C.C. Hwang, Z.M. Yang, C.C. Chiou, and C.C. Lin, "The Design and Implementation of the NCTUns 1.0 Network Simulator," Computer Networks, Vol. 42, Issue 2, June 2003, pp. 175-197.

[8] S.Y. Wang, **C.L. Chou**, C.C. Lin, "The Design and Implementation of the NCTUns Network Simulation Engine," Simulation Modelling Practice and Theory, 15 (2007) 57-81.

[9] NCTUns Network Simulator and Emulator, available for download at `http://NSL.csie.nctu.edu.tw/nctuns.html`.

[10] The Network Simulator - ns-2, available at `http://www.isi.edu/nsnam/ns`.

[11] The QualNet software, available at `http://www.scalable-networks.com/`.

[12] The OPNET modeler, available at `http://www.opnet.com/`.

[13] R. Barr, "Java in Simulation Time / Scalable Wireless Ad hoc Network Simulator," available at `http://jist.ece.cornell.edu`.

[14] The ptv simulation - VISSIM, whose reference link is `http://www.english.ptv.de/cgi-bin/traffic/traf_vissim.pl`.

[15] The TransModeler traffic simulator, whose reference link is `http://www.caliper.com/transmodeler/`.

[16] The SUMO traffic simulation package, available at `http://sumo.sourceforge.net/index.shtml`.

[17] A real-time freeway traffic simulator - FreeSim, available at `http://www.freewaysimulator.com`.

[18] A microscopic traffic simulation model - CORSIM, whose reference link is `http://www-mctrans.ce.ufl.edu/featured/TSIS/Version5/corsim.htm`.

[19] H. Wu, J. Lee, M. Hunter, R. M. Fujimoto, R. L. Guensler, and J. Ko, "Simulated Vehicle-to-Vehicle Message Propagation Efficiency on Atlanta's I-75 Corridor," in Transportation Research Board Conference Proceedings, Washington D.C., 2005.

[20] Multiple Simulator Interlinking Environment (MSIE) for C2CC in VANETs, available at `http://www.cn.uni-duesseldorf.de/projects/MSIE`.

[21] C. Schroth, F. Dotzer, T. Kosch, B. Ostermaier, and M. Strassberger, "Simulating the traffic effects of vehicle-to-vehicle messaging systems," in Proc. of the 5th International Conference on ITS Telecommunications, Brest, France, 2005.

[22] The TraNS (Traffic and Network Simulation Environment), available at `http://wiki.epfl.ch/trans`.

[23] B. Khorashadi, A. Chen, D. Ghosal, C.N. Chuah, and M. Zhang, "Impact of Transmission Power on the Performance of UDP in Vehicular Ad Hoc Networks," ICC 2007. IEEE International Conference on Communications.

[24] R. Vuyyuru and K. Oguchi, "Vehicle-to-Vehicle Ad Hoc Communication Protocol Evaluation using Simulation Framework," in Proc. of the 4th IEEE/IFIP Wireless On demand Networks and Services, pp. 100-106, Austria 2007.

[25] L. Bononi, M. Di Felice, M. Bertini, E. Croci, "Parallel and Distributed Simulation of Wireless Vehicular Ad Hoc Networks," in proc. of the ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM), Torresmolinos, Spain, 2006.

[26] C. Gorgorin, V. Gradinescu, R. Diaconescu, V. Cristea, L. Ifode, "An Integrated Vehicular and Network Simulator for Vehicular Ad-Hoc Networks," in Proc. of the European Simulation and Modelling Conference (ESM), Bonn, Germany, May 2006.

[27] Jungkeun Yoon, Mingyan Liu, and Brian Noble, "Random Waypoint Considered Harmful," IEEE INFOCOM 2003, March 2003.

[28] S.Y. Wang, H.T. Kung, "A simple methodology for constructing extensible and high-fidelity TCP/IP network simulators, IEEE INFOCOM'99," March 21-25, New York, USA, 1999.

[29] S.Y. Wang, H.T. Kung, "A New Methodology for Easily Constructing Extensible and High-Fidelity TCP/IP Network Simulators," Computer Networks 40 (2) (2002) 257-278.

[30] Harvard TCP/IP network simulator 1.0, available at `http://www.eecs.harvard.edu/networking/simulator.html`.

[31] Hubert Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," IEEE Transactions on Communications, Vol. 28, No. 4, April 1980, pp. 425 - 432.

[32] S.Y. Wang, **C.L. Chou**, Y.H. Chiu, Y.S. Tseng, M.S. Hsu, Y.W. Cheng, W.L. Liu, and T.W. Ho, "NCTUns 4.0: An Integrated Simulation Platform for Vehicular Traffic, Communication, and Network Researches," 1st IEEE WiVec 2007 (International Symposium on Wireless Vehicular Communications, colocated with VTC 2007 Fall), September 30 - October 1, 2007, Baltimore, MD, USA.

[33] S.Y. Wang and **C.L. Chou**, "NCTUns Simulator for Wireless Vehicular Ad Hoc Network Research," a chapter of the "Ad Hoc Networks: New Research" book, 2008, (ISBN: 978-1-60456-895-0, published by Nova Science Publishers)

[34] S.Y. Wang and **C.L. Chou**, "NCTUns Tool for Wireless Vehicular Communication Networks Research," Simulation Modelling Practice and Theory," (accepted and to appear) [SCI]

[35] "ESRI Shapefile Technical Description," An ESRI white paper, July 1998.

[36] S.Y. Wang, **C.L. Chou**, "On the characteristics of Information Dissemination Paths in Vehicular Ad Hoc Networks on the Move," International Journal of Computer Systems Science and Engineering, Vol. 23, No. 5, 2008 (SCI)

[37] S.Y. Wang, **C.L. Chou**, and C.C. Lin, "On the Characteristics of Routing Paths and the Performance of Routing Protocols in Vehicle-Formed Mobile Ad Hoc Networks on Highways," Wiley Wireless Communications and Mobile Computing (accepted and to appear, already published online on March 24, 2009) (SCI)

[38] S.Y. Wang, H.L. Chao, K.C. Liu, T.W. He, C.C. Lin and **C.L. Chou**, "Evaluating and Improving the TCP/UDP Performances of IEEE 802.11(p)/1609 Networks," IEEE ISCC 2008 (IEEE Symposium on Computers and Communications 2008), July 6-9, 2008, Marrakech, Morocco.

[39] S.Y. Wang, **C.L. Chou**, K.C. Liu, T.W. Ho, W.J. Hung, C.F. Huang, M.S. Hsu, H.Y. Chen, and C.C. Lin, "Improving the Channel Utilization of IEEE 802.11p/1609 Networks," IEEE WCNC 2009 (Wireless Communications and Networking Conference), April 5-8, 2009, Budapest, Hungary.

[40] The collection of the NCTUns-based papers is available at `http://nsl.csie.nctu.edu.tw/NCTUnsReferences/`.

[41] The NCTUns forum is located at `http://nsl10.csie.nctu.edu.tw/phpBB/`.