Undergraduate Project Report
2016/17

[Extending the TCP Socket Interface for the Data Centre]

Name:	[Yuqi Li]
Programme:	[E-Commerce]
Class:	[2013215115]
QM Student No.	[130806550]
BUPT Student No.	[2013213384]

Date [15th, May 2017]

Table of Contents

Abstract

A data centre is a set of communication systems and a large number of servers which are working co-ordinately. Data centres are important carriers of cloud computing, data-intensive applications, and big data. As the rapid increase in IT technology, people have more and more requirements of data centres. Although many countries and global recognised companies have paid significant attention to the development of data centres, data centres still have problems of routing, bandwidth, and electric power system, etc. Scheduling of large flows is one of them.

This project aims to create a simple system to supply a possible solution to scheduling problems. This system creates a manager between TCP senders and receivers with C under Linux. Normally, the manager can collect information for data flows of terminals before terminals start to communicate. Besides, the manager can display some information which may be useful for data centres to manage data flows. As a result, future study can use this manager to re-configure or manage flows of networks, and this could avoid blocking and congestion.

Keywords: data centre, scheduling, manager, TCP, C, Linux

[RN_3384 Extending the TCP Socket Interface for the Data Centre]

摘要 (Chinese translation of the Abstract)
一个数据中心是一组工作协调的通信设备和大量服务器的集合。数据中心是云计算、密级数据应用和大数据的重要载体。随着如今 IT 技术的飞速发展，人们对数据中心的需求越来越多。尽管许多国家、世界级的著名公司已经越来越重视数据中心的发展，数据中心仍然存在许多问题，例如，欠缺足够先进的路由技术，带宽和电力系统负载问题未得到妥善解决等等。大型数据流的调度就是其中的问题之一。

本次毕业设计的目的在于对数据中心的网络问题，尤其是大型数据流调度问题进行研究。然后，用 C 语言建立一个基于 Linux 系统的简单系统，用来提出一个数据流调度方面可能解决方案。这个系统会在 TCP 的发送端和接收端之间，建立一个“管理员”。在这些终端正常通信之前，“管理员”会收集它们即将发送的数据流的基本信息。除此之外，“管理员”会展示一些可能对数据中心管理数据流有用的信息。对于未来的相关研究，人们可以使用“管理员”来重新配置或管理流，让“管理员”起到尽可能避免阻塞等作用。

关键词：数据中心，数据流调度，管理员，传输控制协议，C 语言，Linux 系统

Chapter 1: Introduction

1.1 The Problem Statement

As the rapid development of cloud computing and big data, data centres [Definition 3], the vital carriers, play more and more important roles. Data centres should be capable of high abilities to support data-intensive works. Usually, data centres need to supply nonstop service. Once a data centre went wrong or crashed down, there would be tens of thousands of valuable data being messy, even lost. Especially, for data centres run by national organisations, global companies or banks, there will be tremendous losses when serious problems occur. Thus, it is vital for a data centre to guarantee its stability. A data centre should change or scale out smoothly as the changes of data and demands of customers.

Data centres are not mature enough, especially distributed data centres which are applied more and more. Nowadays, it is difficult for programmers to debug date centres instantly. Besides, because of data-intensive services in data centres, it is almost impossible for programmers to replay scenarios to analyse bugs. Therefore, data centres should avoid bugs as much as possible. On the aspect of data flows, a paper (Chowdhury, Zhong and Stoica, 2014) [1] states that when the resource is determinate, large data flows may occupy resources of small flows, causing small flows to miss their deadlines. In a set of communications, if any of related flows does not finish its tasks, the whole communications cannot terminate. Thus, large flows are possible to cause some problems in data centres (e.g. if this communication is based on First-Input-First-Output (FIFO)[1], Head-of-line blocking[2] is easy to happen. As a result, the efficiency of bandwidths of data centres may decrease, and resources will be wasted. Even worse, blocking and congestion may occur).

1.2 The Solution by this Project

One possible solution to this problem is to set a central manager among terminals in data centres. Currently, most typical data centres have independent control planes [Definition 1] and data planes [Definition 2]. They are part of the routing architecture in data centres. A paper (Zamfir, Altekar, and Stoica, 2013)[2] shows that bugs of data centres usually occur in control planes.

[1] FIFO is a traditional sequential execution method. The flow which enters the process first will be processed first.
[2] Head-of-line blocking is a blocking occurs in buffer communication. When a set of flows ingresses by FIFO, maybe the first flow is process while ports used for flows behind it have already been available. This will cause wasting.

Definition 1 A control plane is responsible for planning the overall network structure and arranging the orders of ingress of packets.

Definition 2 A data plane determines how to deal with packets which have entered a data centre.

Hypothetically, there is a manager. The manager can record what happens to data flows among those terminals in a data centre. As an underlying supporter, the manager can be used to coordinate with transmission situations. It may be helpful to solve scheduling or routing problems in data centres.

In this paragraph, I will give an example to describe the idea. Assume that, there is a central manager in a data centre. Client A wants to send several large data flows to Server B. However, B is extremely busy. Under this circumstance, Client A is likely to fail to send these flows. Blocking and congestion may occur. However, if Client A could acquire the working conditions of Server B, re-arranging the time of transmission or the destination of this communication, it is possible to avoid some scheduling problems of data flows. Or, perhaps the set of data flows of Client A is very urgent. Server B is informed of this communication in advance; it can redeploy its resources to accept those flows. If so, the data centre can keep running stable, guaranteeing the efficiency of working. Moreover, by long-term and volume data collections and analyses, according to a paper (Stoica, Abdel-Wahab, Jeffay, Baruah, Gehrke, Plaxton, 1996) [3], the manager can inform a data centre to supply relatively more resources to specific servers depending on using weights. The methods of utilising the manager should be various. The takeaway is the manager can collect pieces of information for data flows.

It should be emphasised that the goal of this project is establishing a simple system with a simple manager. The manager only can collect information for flows of terminals. The other advanced functions of the manager (e.g. monitoring, re-configuring, and managing the entire network) are out of the scope of the project.

Figure 1 is the simple system in the project. In this system, there are some simple clients and a server. They are used for simple communications. Besides, there is a manager which is responsible for collection information for data flows of these terminals. Because I assume the manager only can collect information of data flows, the manager does not need to interact with those terminals. Meaning, clients and servers do not report to the manager intuitively. Thus, I assume they do not need to know the existence of the manager. Information for data flows is collected secretly.

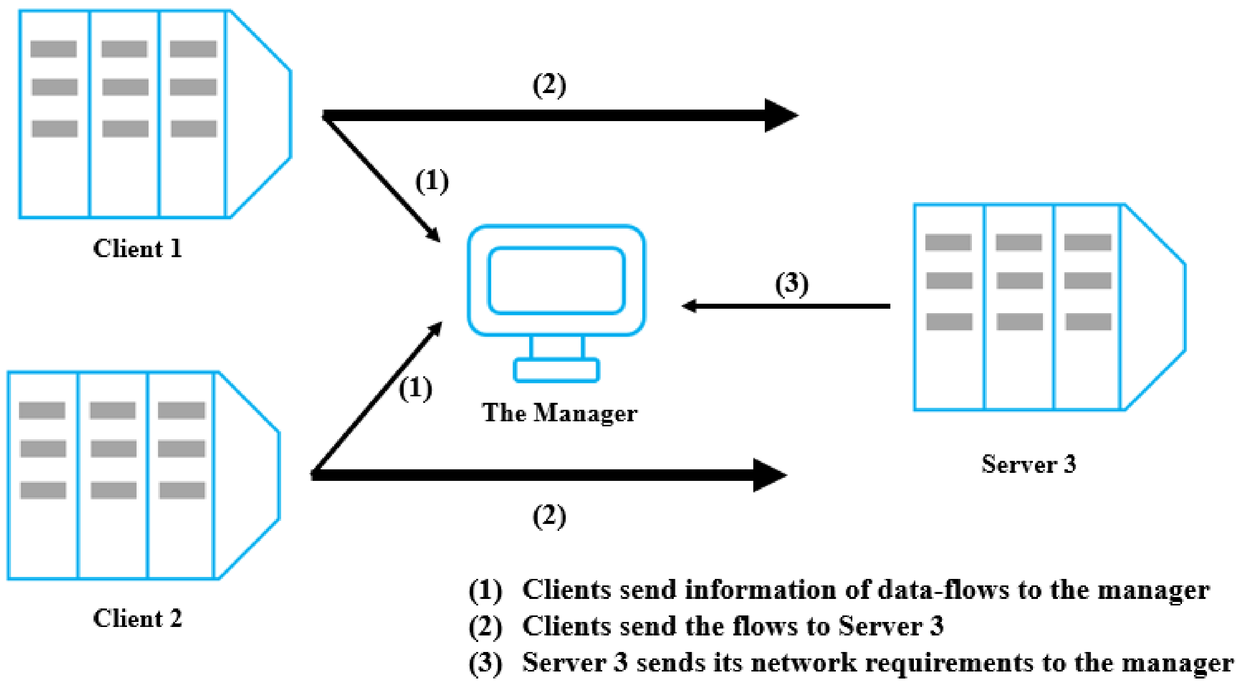

(2)
(1)
Client 1
(3)
(1)
The Manager
Server 3
Client 2
(2)
(1) Clients send information of data-flows to the manager
(2) Clients send the flows to Server 3
(3) Server 3 sends its network requirements to the manager

Figure 1 The Simple System in the Project

1.3 The Novelty of the Solution

In a data centre, there is a large plenty of one-to-many communications and many-to-many communications. Sizes of data flows are different. If I want to collect all of flows and analyse them one by one, this is time-consuming and wasteful. According to researches (Habibi, Mokhtari, Sabaei, 2016) [4][5] on practical data centres, extremely large data flows only take about 10% of all flows. However, they can occupy almost 80% resources of all flows. Meaning, I can monitor large flows by the manager to synchronize information for flows and prevent some scheduling problems (e.g. congestion). Also, I can analyse network conditions of date centres by data collected by the manager, locating problems and concluding possible solutions.

This project simplifies procedures of collecting information for data flows by overriding existing communicating C functions. Without overriding the function, collecting information for flows will be complicated; terminals need to send information to the manager beforehand and then communicate. To simplify the progress, I overrode two traditional socket programming functions **connect()** **[Appendix C]** and **accept()** **[Appendix D]** under Transmission Control Protocol (TCP)[Definition 10]. In the system, terminals only need to collect or set information they want to inform the manager, then they call **connect_init()** or **accept_init()**[3]. As a result, they can send the information to the manager before they establish communicating connections. This attempt is successful and proves that, out of the scope of the project, it is possible to override more TCP socket programming functions to meet different demands (e.g. every time a server receives a flow by a function overridden by **recv()**[4], it judges whether the flow is in the receiving plan and in the range of acceptable sizes. If not, the server can inform the manager).

In the project, the manager can do some simple analyses depending on received information. To keep a data centre stable, the manager displays some useful variables of flows (e.g. the number of flows whose sizes are bigger than 5 GB). Also, the manager stores analytic data and received information into certain files, which I will discuss in 3.2.3. Admins can locate a particular flow according to the place where stores the flow. Then, admins can do detailed analyses (e.g. Client A informs the manager that it will send an urgent flow with 20 GB. In this data centre, there is no specific approach to deal with such kind of flows automatically yet. The admin can get the information by a report displayed by the manager per minute, locate the flow by the place where stores the piece of information, then redeploy Client A or its destination).

The rest of the paper is structured as follows: Firstly, I will introduce some backgrounds of this project. Then, I will discuss the design, the implementation, and testing of the project. These chapters will focus on important parts of the project. Thirdly, I will show some results of the project and discuss them. Eventually, I will give conclusions and further work.

[3] The two function are the overridden functions of **connect()** and **accept()**. They will be discussed in detail in Chapter 3:.
[4] A traditional TCP socket programming function. This function is for receiving flows.

Chapter 2: Background

2.1 Data Centres

Definition 3 Data centres are resources pools combined by a large number of servers[5] and communication systems. They can provide a large number of virtual infrastructures, available software, and business computing solutions to customers.

A large quantity of networks and communicating infrastructures are carriers of data centres. Robust power systems and bandwidth are their supporters. Data centres links servers and other systems by complicated architectures and structures to do data-intensive operations efficiently.

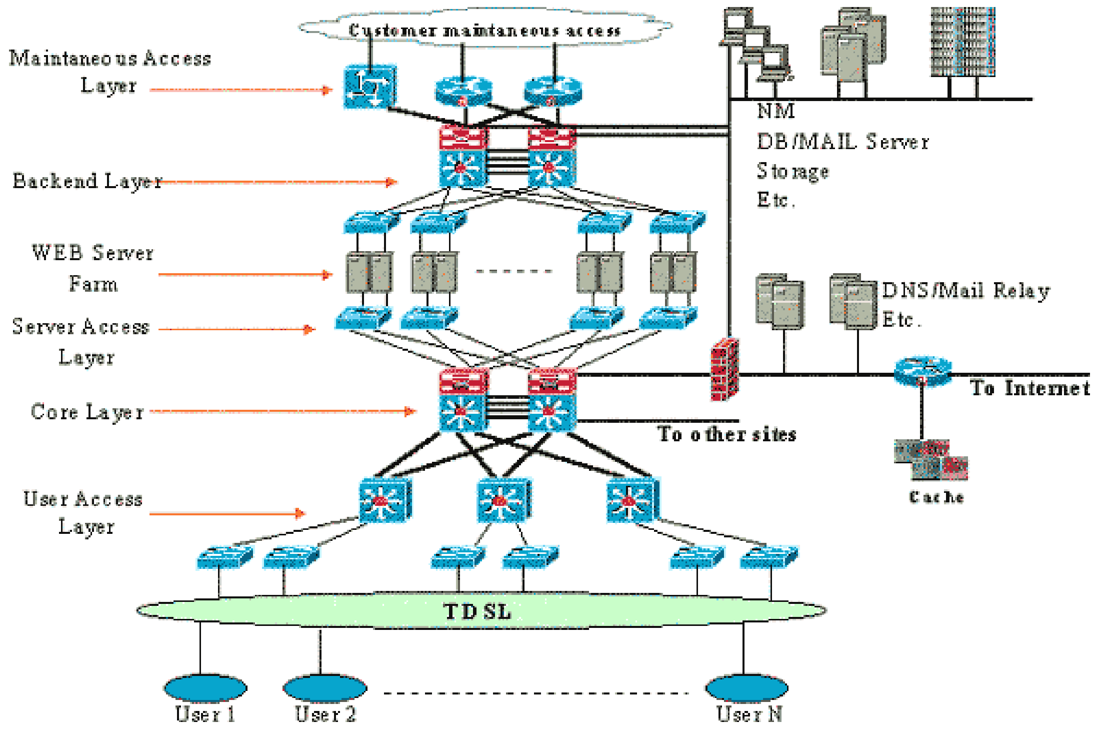


Figure 2 A Traditional Structure of a Data Centre by Cisco [6]

Figure 2 is a typical topology of traditional data centres by Cisco. Core computers in such kind of data centres are the minority. Because cloud computing, data-intensive applications and big data are more and more accessible, this kind of topologies cannot meet current increasing demands for dealing with a huge number of data. Thus, data centres have developed various topologies gradually

[5] A small data centre usually has about 500 servers. A medium data centre has less than 2000 servers. A large data centre usually has less than 10 thousand servers.

for different intentions (e.g. cubic topologies). A powerful data centre now is capable of accomplishing tasks of dealing with data, data query, and data storage. By data centres, service providers can acquire accurate and professional analyses to meet various demands of customers.

Data centres are widely-used. Many global recognised IT companies have built up their data centres. According to a magazine article (Metz, C., 2012)[7], Google has a set of data centres in serval countries whose Internet technologies are well-developed, especially in Asian-Pacific region. It uses these data centres to develop their business or outsource. According to an article (Miller, R., 2013)[8], to meet the growing office application demands, Microsoft also started to establish data centres in 2008. Besides, in recent decades, many countries also begin to develop data centres for studies of aerospace, biology, and government affairs, etc.

Although data centres have gained attention globally, they still are not perfect. Nowadays, there are three main problems of data centres. Firstly, the way of dealing with "elephant flows" [Definition 4] is not well-developed. To discuss scheduling problems, a paper (Liang, Gao. and Matta, I., 2001)[9] has put forward a pair of interesting definitions. They are "elephant flows" [Definition 4] and "mice flows" [Definition 5]. The definitions are not very clear because they are just a kind of metaphors used widely. They are a pair of relative concepts. Usually, "mice flows" are easy to fail of the competition in traffic with "elephant flows". A recent paper (Wei, W., Yi, S., Salamatian, K. and Zhongcheng, L., 2016)[10] propose a new flow scheduling, "Freeway", for data centres. To make "Freeway" easy to be used, the authors give more clear examples of the two kinds of flows. This shows that researchers can define the two kinds of flows in detail to meet their demands.

Definition 4 An Elephant is a connection which can carry a large percentage of traffic. An Elephant Flow is a flow with colossal bytes in an Elephant connection.

Definition 5 A Mice is a connection which is very small in traffic. A Mice flow is a flow in a Mice connection.

In a data centre, a research (Habibi, Mokhtari, Sabaei, 2016) [4] shows that, in every ten data flows, one "elephant flow" may occur once. However, "elephant flows" takes about 80% of the overall traffic capacity. According to Service Level Agreement (SLA)[6], data centres should coordinate different kinds of flows with different routing technology and protocols. Most bugs occurring in data centres are because people use technologies which are designed for "mice flows" to manage

[6] SLA is a contract between Internet Service Providers and users. It defines a set of terms (e.g. service types, Quality of Service and Receive Payments).

“elephant flows” (e.g. using Equal-Cost Multipath Routing (ECMP)[7] to deal with all mixed flows). Secondly, requirements of power systems and bandwidth are strict in data centres. If a company makes the structure and the architecture of power systems and bandwidth conditions of a data centre excessive excellent, this will cause massive economic loads. On the contrary, it is also infeasible. Thirdly, depending on inherent characters of data centres, it is difficult to debug and replay bugs.

To solve problems above, serval researchers have come up with different innovative solutions. However, it is difficult for most of them to implement their solutions because of massive costs of data centres.

2.2 Existing Solutions to Data Centres

Generally, to solve network problems of data centres, there are two main views. One is to set some kinds of central controllers in a data centre. A controller usually focus on a certain problem and work like an “administrator”. Another is to improve inherent algorithms, architectures or scheduling of data centres.

In terms of flow scheduling problems of a data centre, a research (Chowdhury, Zhong, and Stoica, 2014) [1] give a possible solution named “Varys” [Definition 6]. This is the most vital solution I refer to in my project, although it is not widely-used in data centres. Materials of “Varys” are limited on the Internet.

Definition 6 Varys is a set of complicated APIs which can assist servers in a data centre to coordinate flows and manage scheduling. Mainly, Varys have four functions: **register(), unregister(), put(), and get().**

Definition 7 “A coflow is a set of flows which are sharing the same performance goals”[1].

At the beginning, the authors define a “coflow” [Definition 7]. The solution is based on dealing with “coflows”. “Varys” is used for making impractical central coflow scheduling feasible. It works as a daemon to assist servers to coordinate time-coupled data coflows. It mainly focuses on managing “elephant flows” [Definition 4]. Firstly, “Varys” need to conduct on “an admission control”: it regulates deadlines and transmission speeds of flows to decrease Coflow Completion Time (CCT).[8] After that, clients call **register().** It submits clients to a “Varys master” and gets “coflow id” for these clients. The “Varys master” will record information for the coflow and judge

[7] A routing technology for small flows.
[8] The completion time of a coflow. Only the last flow of a coflow arrives, the coflow can finish.

whether the client can send the coflow or not. If the coflow is not planned to send this time, the client will use **unregister()**. Then, the client waits a submission next time. If permitted, the client will use **put()** and a server use **get()** to initialise. Besides, "Varys" APIs can send "measurements of the network usage at each machine to varys master" [[1]] to help analyse bandwidth usage.

Another solution to optimise network scheduling is widely-used. It is called Software-Defined Network (SDN). According to 1.2, a data centre usually has a control plane [Definition 1] and a data plane [Definition 2]. SDN for a data centre is to create a "flexible and programmable network" (Habibi, Mokhtari, Sabaei, 2016) [4] to separate the two planes. Its core is a network switching model called "OpenFlow". By SDN, a network is separated into switches, a controller and "OpenFlow". Figure 3 is the layer diagram of "OpenFlow". It has 3 layers. Infrastructure layer is responsible for switching. Control Layer controls the whole network. In a data centre, an Applications Layer can help virtualization work.

Besides creating controllers in data centres, researchers are seeking for other ways to make data centres more robust. A paper (Zamfir, Altekar, and Stoica, 2013)[2] also comes up with a solution to debug data centres, called "ADDA". It synchronises data in a data plane when a data centre is working. Thus, when bugs occur, "ADDA" can replay them to the control plane. By this way, quick debugging is realised.

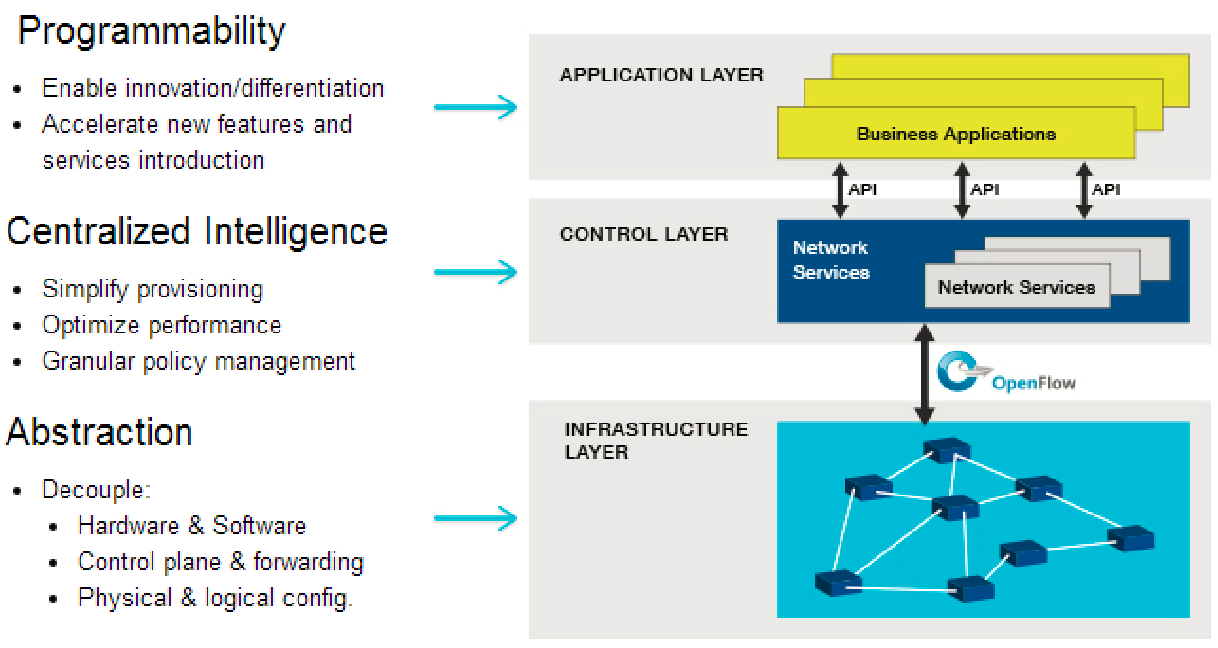


Figure 3 OpenFlow Diagram [12]

2.3 Linux and C

Linux is an operating system which is very suitable for programmers. It can run UNIX[9] tools, applications, and network protocols stably.

C is an object-based computer programming language which is applied extensively. It is very historical and fundamental. Operating systems (OSs) were compiled by C and assembly languages. C is equipped with excellent extension. Its compilers and integrated development environment (IDE) have been well-developed. Different from relatively advanced languages (e.g. Java and Python), C does not need to run with supports of virtual machines. C can generate executive files directly. However, as a kind of old computer programming languages, C lacks abundant interfaces or packages (e.g. Java packages). Thus, it is a little uncomfortable for programmers to use C comparing ones who use Java or more new languages like Python. In a C file, the **main()** function is the core thread in this process.

Similar with Java, C also has its own variable types. However, it does not have the concept of "object". Its operations are object-based, not object-orientated. In C, structures [Definition 8] can play part of role of "objects".

Definition 8 In C, a structure is a group of "objects" contains some variables (e.g. a structure "students" can have an object student "Tom". "Tom" can have variables like age, grade, major, etc.). Structures are the most efficient way to deal with a large number of variables in C.

Definition 9 An API is an application programming interface. It can be sets of functions implemented beforehand. Programmers can use opensource interfaces or write interfaces by themselves to reuse code.

In C, interfaces [Definition 9] are implemented by C header files and their library (e.g. to write a C file *hello.c*, a programmer wants to use a function: **extern int printf (const char *format)**. This function is in library *stdio.c*. He needs to ensure that *hello.c*, *stdio.c*[10], and *stdio.h*[11] are under the same directory[12]. Then, he declares **#include<stdio.h>** in hello.c then calls **printf()**. *stdio.h* begins to search whether there is stdio.c. If *stdio.c* exists, *stdio.h* maps **printf()** in *hello.c* to **printf()** in *stdio.c*. So, the programmer can use **printf()** successfully).

[9] A powerful and multitask operating system.
[10] The C file realises **printf()**.
[11] The C header file of **printf()**.
[12] Usually, C compilers bring many universal interfaces.

[RN_3384 Extending the TCP Socket Interface for the Data Centre]

It is necessary for programmers to write TCP socket programs with C under Linux or UNIX. During the stage of establishing underlying network communicating programs, many programs were established by C. Thus, there are a lot of available codes or APIs [Definition 9] of socket programming under Linux/Unix.
The following contents about programming are all based on C under Linux.

2.4 TCP Socket Programming

2.4.1 TCP

Figure 4 is the TCP/IP Model. It is a network communicating model. It involves a set of protocols, named TCP suites. It is the fundamental communication architecture of Internet. TCP is the core protocol of TCP suites. It is between the Network Layer and the Application Layer. It supplies various complicated services between networks and applications.

Before two terminals start to communicate, TCP will build up a “pipe” between the terminals. This “pipe” does not exist physically. It is just a virtual link. The two terminals must transfer byte streams through this “pipe”. When they want to finish this communication, the “pipe” must be closed.

Definition 10 TCP is a stream-oriented protocol which provides connection-oriented services.

TCP is a reliable transport protocol. It improves its quality of services by the flow control, the error control, and the congestion control. To start with, the goal of the flow control is to protect receivers from being overrun by senders. A TCP sender will control the capacity of the receiver. At the meantime, the TCP receiver will control how many bytes the sender can send. Secondly, the error control is used to decrease impacts of losing packets. This control is byte-oriented. The point is retransmission of packets. When a packet will be sent, it will be stored in a queue until the sender receives an acknowledgement character (ACK). When the timer of retransmission is time-out, or the sender receives three ACKs of the first packet in this queue, retransmission will start. Thirdly, to know about the congestion control, I should explain what congestions are at first. In the field of networks, congestion means the capacity of the network cannot burden the practical loads of all sending packets. The congestion control is designed to prevent these overloading situations. Usually, under TCP, resources will set the capacities of themselves to ensure the network is available.

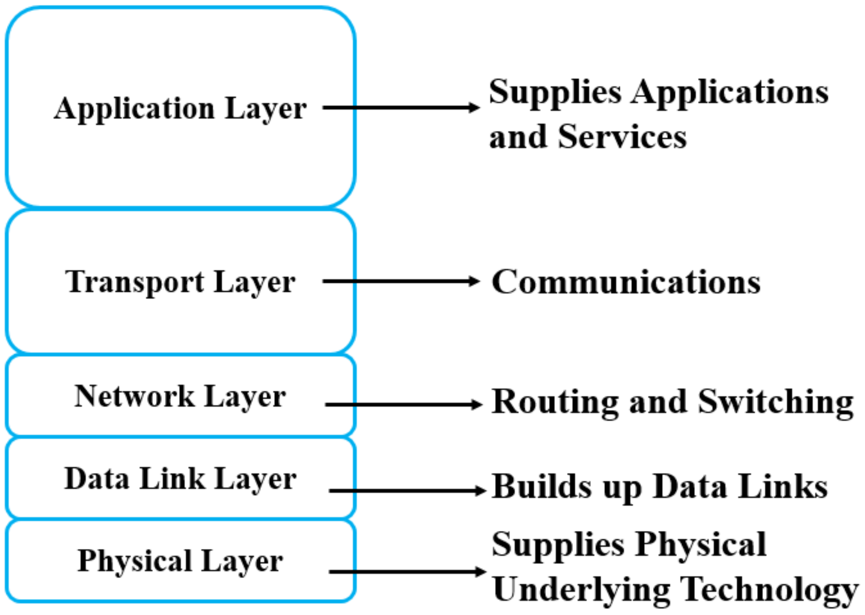


Figure 4 TCP/IP Model

2.4.2 Simple TCP Socket Programming

In communications based on TCP/IP suites, processes use sockets [Definition 11] to establish connections then transfer data. Based on sockets, TCP/IP suites have client-server communications similar with most network communications. **Figure 5** is a client-server communication under TCP.

Definition 11 A socket contains the IP address and the port number used for communication. Inherently, a socket is an encapsulated interface of TCP/IP.

Figure 6 is a flow chat of a simple socket communication under TCP. Most functions in Figure 6 are from **<sys/socket.h>**. **<sys/socket.h>** maps a C library under Linux. It contains abundant functions and structures used for TCP socket programming. **struct sockaddr [Appendix A]** and **struct sockaddr_in** [Appendix B] are a pair of structures in the library. Usually **sockaddr** is used for communications by functions. **sockaddr_in** is used for operations to **sockaddr**.

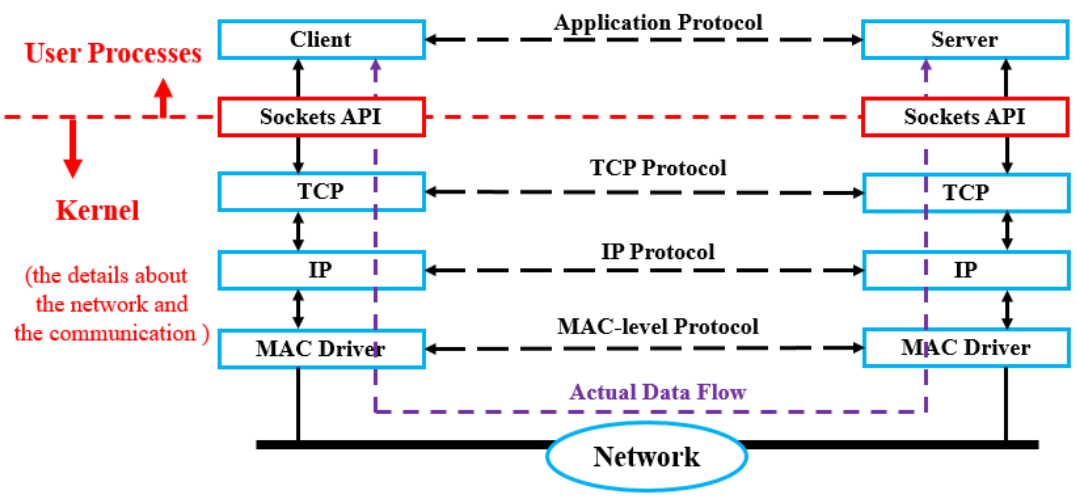


Figure 5 Client-Server Communication Based on TCP/IP [11]

I will describe the TCP socket programming in Figure 6 in this paragraph. Firstly, Server B needs to build up a socket used for listening coming connections. “sock2” is the descriptor of the socket. All following steps are based on this socket descriptor. Secondly, because the socket is used for listening to connections to the server, it should be bound to the local address. Then, Server B uses “sock2” to listen to whether there is connection request. The maximum of connections which are queuing is 100[13]. By the character of servers, programs of servers must start beforehand. Fourthly, Server B starts to accept coming connections. Interestingly, Server B does not use “sock2” to communicate to clients directly because “sock2” is concentrated on listening. Always, **accept()**[Appendix D] handles a new connection and creates a new socket “sock3” for this connection silently. Thus, “sock2” is only responsible to listen while new sockets like “sock3” are for communications. Next, Client A can start its program. Also, Client A should create its own socket used for the communication. Then, it calls **connect()** [Appendix C] to try to connect to Server B. A return value will be given after **connect()**. Programs will judge whether **connect()** successes or not. If succeeds, Client A begins to communicate with Server B. In this project, functions **connect()** and **accept()**are the most vital ones in a TCP communication.

[13] The biggest number of queuing connections is 65,536 theoretically. However, according to different OSs, the max value could be “1024”.

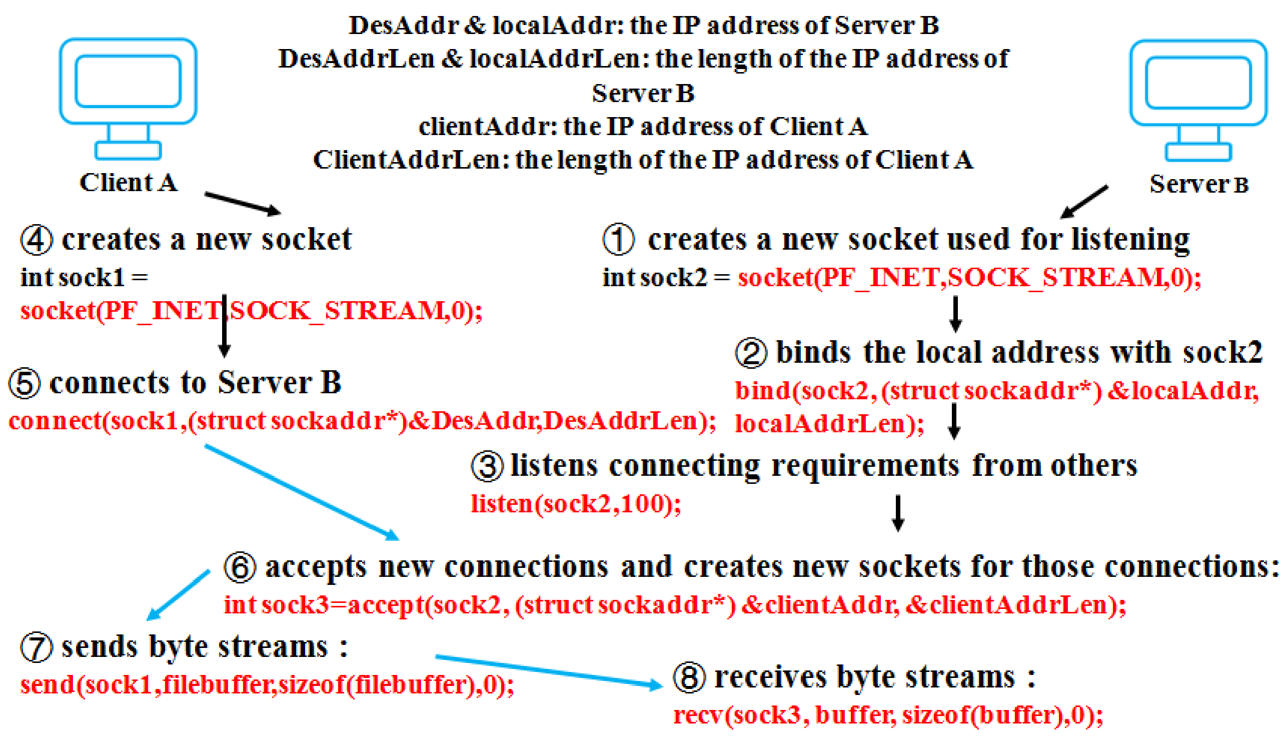

DesAddr & localAddr: the IP address of Server B
DesAddrLen & localAddrLen: the length of the IP address of Server B
clientAddr: the IP address of Client A
ClientAddrLen: the length of the IP address of Client A
Client A
Server B
④ creates a new socket
int sock1 = socket(PF_INET,SOCK_STREAM,0);
① creates a new socket used for listening
int sock2 = socket(PF_INET,SOCK_STREAM,0);
② binds the local address with sock2
bind(sock2, (struct sockaddr*) &localAddr, localAddrLen);
⑤ connects to Server B
connect(sock1,(struct sockaddr*)&DesAddr,DesAddrLen);
③ listens connecting requirements from others
listen(sock2,100);
⑥ accepts new connections and creates new sockets for those connections:
int sock3=accept(sock2, (struct sockaddr*) &clientAddr, &clientAddrLen);
⑦ sends byte streams :
send(sock1,filebuffer,sizeof(filebuffer),0);
⑧ receives byte streams :
recv(sock3, buffer, sizeof(buffer),0);

Figure 6 The Flow chart of a Simple TCP Socket Programming

2.4.3 Multithreaded TCP Socket Programming

However, it is almost impossible to use the structure in Figure 6 practically, especially for large distributed systems (e.g. data centres). The structure is too simple. Under that circumstance, numbers of flows sent by clients and received by servers are strictly limited. To make TCP communication available, multithreaded programming is used. In this section, I will introduce what is a thread [Definition 12] firstly, then discuss how to use thread in TCP socket programming briefly.

Definition 12 A thread is the minimum unit of a process. It is also called "Lightweight Process".

A process usually has many threads for different tasks. A process will allocate resources to threads in it. Threads are atomic. In a process, they compete for taking resources, not run in order. Resources in a process are in different "resource rooms". When a thread wants to use resources of the process, it enters a related "waiting room". In this "waiting room", there are different threads waiting for using the "resource room". Once the thread in the "resource room" finishes its tasks, it leaves. Then, other threads in the "waiting room" compete to enter the "resource room" in the process. In terms of threads, one of main problems is deadlocks. Figure 7 is an example of a deadlock. "thread1" locks "function A" and waits to use "function B". "thread 2" locks "function B" and wants to use "function A". Consequently, they keep waiting. To solve it, two ways are used.

The first way is to make threads run in order. Programmers “mark” threads in order and tell them run one by one. A set of functions are designed for this. The second way is to set a “waiter” for these threads. Threads which gain permissions of the “waiter” can use related resources.

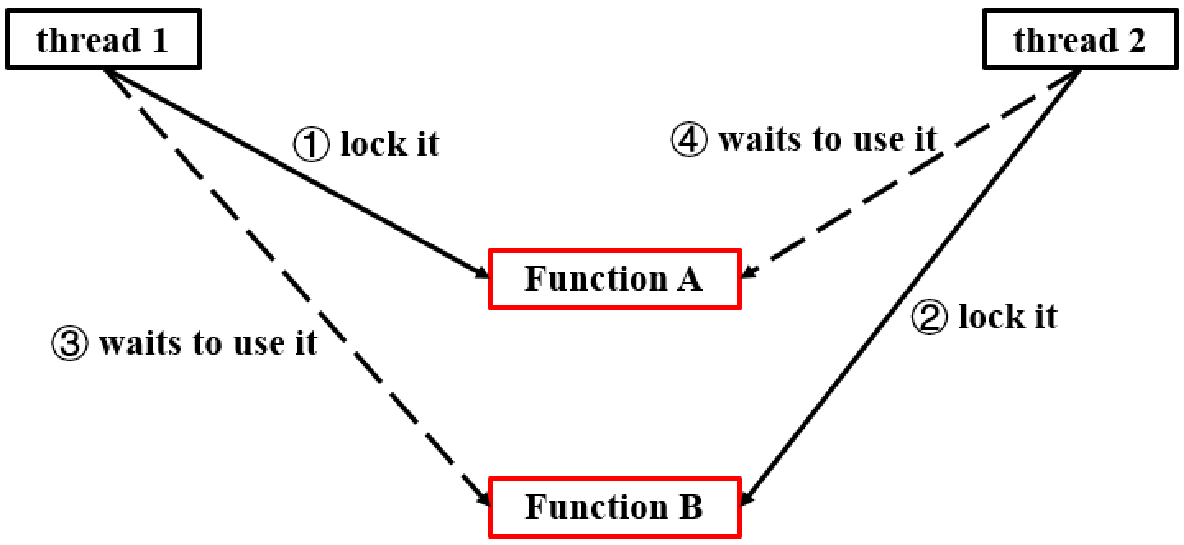


Figure 7 An example of a Deadlock

In TCP socket programming, using multi-threads on the side of clients is very easy. Clients can create a large number of sockets to connect by threads. Programmers only need to pay attention to closing connections correctly and timely. Or, there will be no resource in a few minutes. On the side of servers, threads should be used in proper places. Or, they will be meaningless (e.g. in Figure 6, if I use multi-threads with **recv()**, it means “sock3” has many threads to receive data flows. However, according to features of TCP, “sock3” only can have one connection with a client every time. One **recv()** is enough. If I use threads with **accept()**, every time **accept()** creates a new socket, tasks of the socket will be done by a new thread. Bugs are avoided). Above all, using threads in different blocks of code, different effects occur.

2.5 Some Data Formats

2.5.1 TCP Segment Format

Figure 8 is a TCP segment format. The format contains much information used for a TCP communication.[14] It is well-designed and widely-used in TCP transmission. However, it is very inflexible. All contents are fixed in this format.

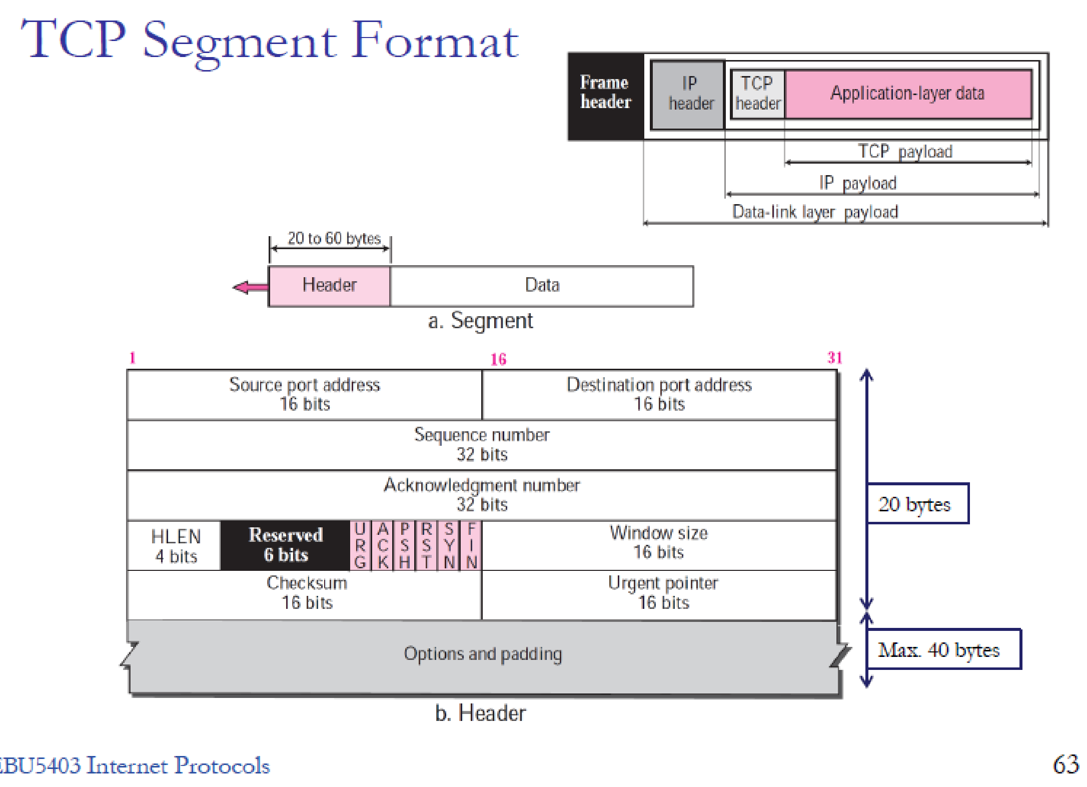
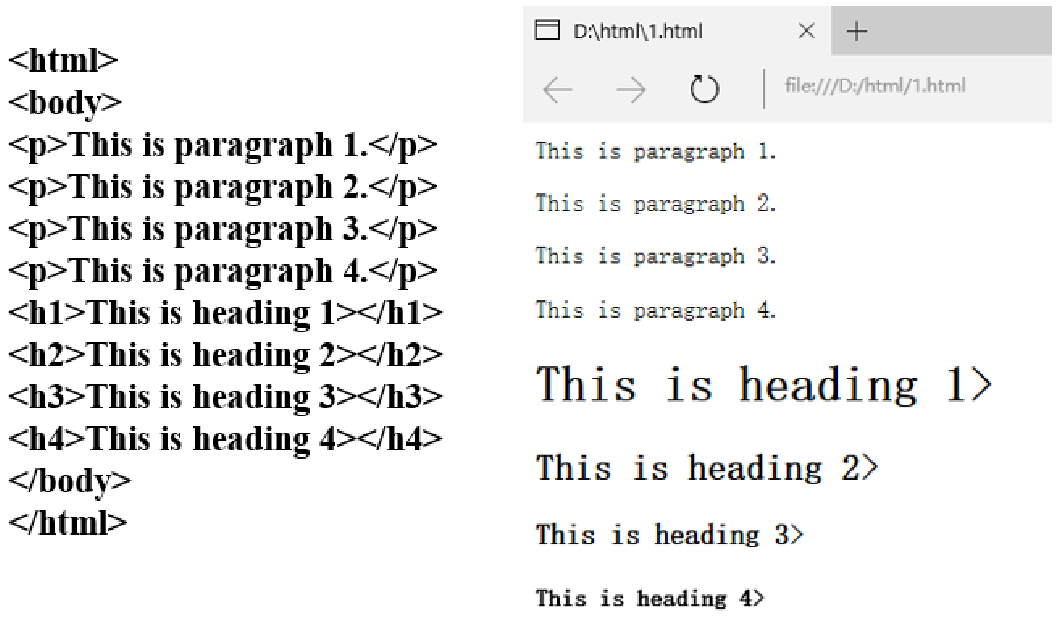


Figure 8 TCP Segment Format[14]

2.5.2 HTML Format

Hypertext Markup Language (HTML) is a mark-up language. It is created for displaying web pages and data, not for data-exchange. Although the format of it is very easy to understand and write, it cannot be extended because all “variables”.[15] of THML are fixed. Figure 9 is a simple example of HTML.

[14] Contents in the format is not the concentration in this project, I will not discuss them in detail.
[15] In HTML, “variables” are tags (e.g. <p> and <h1> in Figure 9).



```
<html>
<body>
<p>This is paragraph 1.</p>
<p>This is paragraph 2.</p>
<p>This is paragraph 3.</p>
<p>This is paragraph 4.</p>
<h1>This is heading 1></h1>
<h2>This is heading 2></h2>
<h3>This is heading 3></h3>
<h4>This is heading 4></h4>
</body>
</html>
```

```
This is paragraph 1.

This is paragraph 2.

This is paragraph 3.

This is paragraph 4.

This is heading 1>

This is heading 2>

This is heading 3>

This is heading 4>
```

Figure 9 An Example of HTML[16]

2.5.3 XML Format

The format of Extensive Markup Language (XML) is very similar with that of HTML. However, it is a totally different language. XML is a data-exchange neutral language. Users can create "variables"[17] they want to use conveniently. Although XML is very easy to extend, because of its format, resources it uses are not few. Next lines are a simple example of XML code.

<bookName1>Harry</bookName1>

<bookName1>Potter</bookName2>

<authorName1>Joanne</authorName1>

<authorName2>Kathleen</authorName2>

<authorName2>Rowling</authorName2>[18]

[16] The code is written by the author of the report.
[17] In XML, "variables" are user-defined tags.
[18] In this case, "<bookName1>", "<bookName2>", "<authorName1>", "<authorName2>" , "<authorName3>" are all tags created by me.

2.5.4 JSON Format

JavaScript Object Notation (JSON) is a lightweight, neutral and cheap data-exchange format. It uses pairs of “key: value” to carry information. It is widely-used so that programmers have developed various JSON parsers with different computer programming languages, including C. Table 1 is pieces of information will be collected in JSON. I will give a case of using JSON:

Table 1 Information of students

First Name	Last Name	Grade	Age	Gender	Mark
Tom	Aaron	1	13	M	67
Jenny	Albert	2	14	F	78
…	…	…	…	…	…

A corresponding JSON string:

{

“student”: [

{“FirstName”: “Tom”, “LastName”: “Aaron”, “Grade”: 1, “Age”: 13, “Gender”: “M”, “Mark”: 67}

{“FirstName”: “Jenny”, “LastName”: “Albert”, “Grade”: 2, “Age”: 14, “Gender”: “F”, “Mark”: 78}

{… …}]

}

In this string, “” represents keys or values. Numerical values don’t use “”. {} is used for objects and [] contains arrays. According to the case in **Table 1** and analysis in last paragraph, I can conclude that JSON is very extensive and lightweight, and this can save a lot of resources. It is easy to parse by its unique characters.

Chapter 3: Design and Implementation

3.1 Preparations

3.1.1 Literature Review

Literature review is based on four fields: data centres, scheduling, TCP, TCP socket programming with C under Linux (TCP socket programming).

Firstly, to get to learn about knowledge above, I review lecture notes of *BBU6404: Internet Applications*, *EBU5403: Internet Protocols*, *EBU6610: Information Systems Management and EBU6042: Advanced Networking Programming*, which are very basic and appropriate for beginners. Secondly, I read several papers, two books, many online materials, and took notes of them. During the literature review of TCP socket programming, I also practice write little simple programs.

3.1.2 IDEs and Compilers

(i) Linux (Ubuntu)

Version: Ubuntu 12.04

Operating System: Ubuntu (32-bit)

IDE: text editor

Compiler: GCC

Tasks: Run C files, test and debug; finish the project

(ii) Windows

Version: Windows 10

Operating System: Windows (64-bit)

IDE: C-Free 5.0

Compiler: Visual Studio 2015

Tasks: Write simple C programs efficiently for the project; Modify C files effectively

(iii) The Shared Directory

To make both OSs work co-ordinately, I link the two systems by a directory named “shared”. This method is available over Oracle VM VirtualBox. Ubuntu is built up as a virtual computer over Oracle VM VirtualBox. Through this directory, users can copy any files from one OS to another. This solves the problem that Ubuntu 12.04 is not readable. Besides, I can build up some snaps of programs under Cfree efficiently and effectively.

3.2 Design

3.2.1 Design of the Overall System

The main task of this project is to build up a simple system. The progress should lead me to get to know and do research on scheduling of data flows in a data centre and TCP socket programming with C. The design of the overall systems has three stages. At stage one, my goal is to establish an extremely simple working system. At stage two, I use multithreads in my project and do some improvements. At stage three, I should improve the performance of the manager substantially. However, because of the limit of time, this stage is not implemented. It will be discussed in 5.2. I will describe the overall system design of the first two stages here.

Firstly, at stage one, I have designed an overall fundamental system. In this system, it should have a client, a C library, a server, and a manager. They are all traditional simple TCP socket programs based on 2.4.2. The manager should be invisible for the client and the server. To establish the system, firstly, I need to have a client. Figure 10 is the flowchart of the client. The tasks of the client are to collect information for data flows into a C structure [Definition 8] and send real flows to the server. Secondly, to make the client send information to the manager before it sends flows to the server, I need a function overridden from **connect()** **[Appendix C]**, called **connect_init()** [Prototype 1]. To make the function can be used widely, it should be in a C library. Users only need to include its C header by “**#include<project.h>**”, and then they call the function. Figure 11 is the flowchart of how the library code works. The tasks of the library are to parse the C structure to a string, send the string to the manager then connect to the server. Thirdly, I need an available server. The tasks of the server are to receive flows from the client and store them into a text file. Figure 12 is the flowchart of the server. Fourthly, a manager should be added between the client and the server. The structure of the manager is almost the same with that of the server. The manager is responsible to receive pieces of data information and store them into a text file.

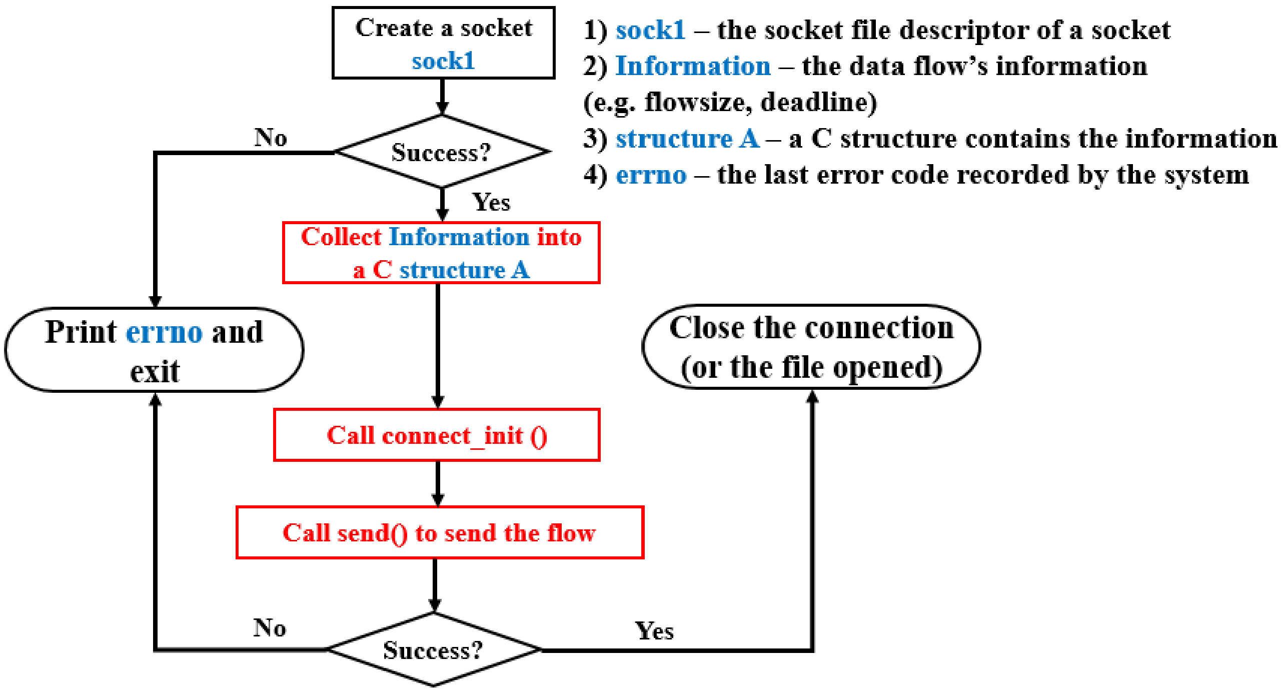

Create a socket sock1
Success?
No
Yes
Collect Information into a C structure A
Print errno and exit
Close the connection (or the file opened)
Call connect_init ()
Call send() to send the flow
No
Success?
Yes
1) sock1 – the socket file descriptor of a socket
2) Information – the data flow's information (e.g. flowsize, deadline)
3) structure A – a C structure contains the information
4) errno – the last error code recorded by the system

Figure 10 The Flowchart of the Client at Stage One

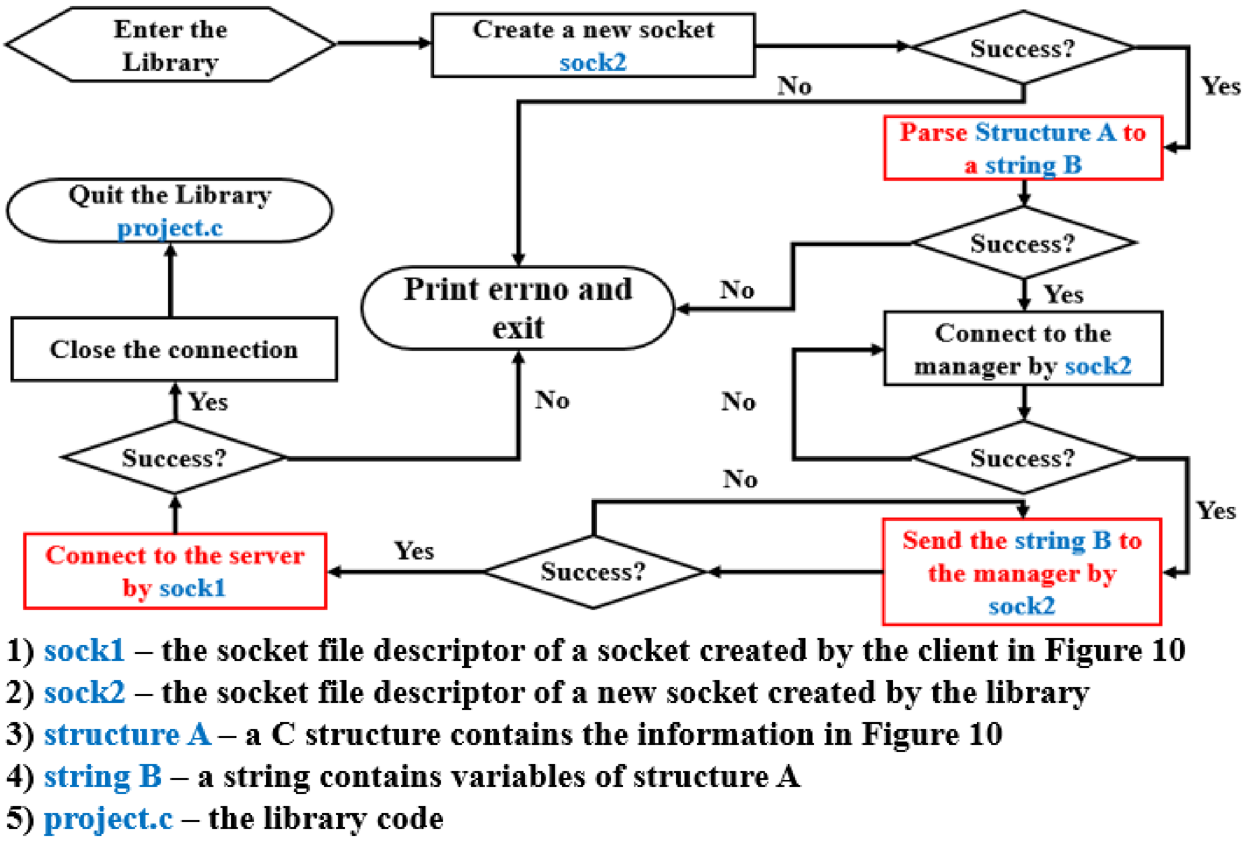
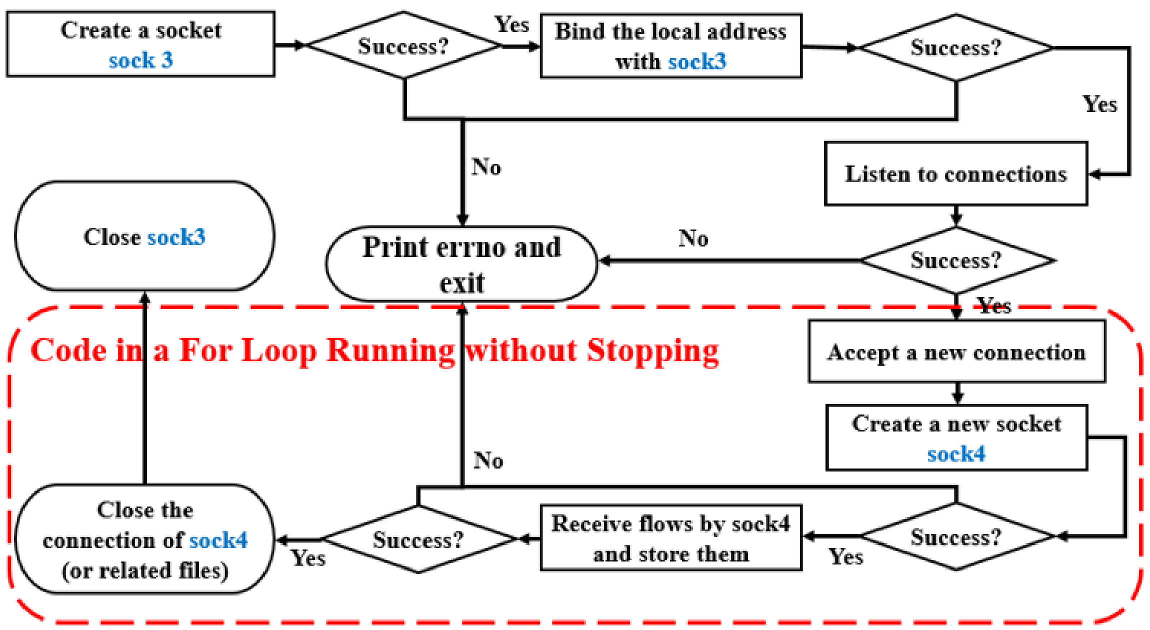

Enter the Library
Create a new socket sock2
Success?
No
Yes
Parse Structure A to a string B
Quit the Library project.c
Success?
Print errno and exit
No
Yes
Connect to the manager by sock2
Close the connection
Yes
No
No
Success?
Success?
No
Yes
Yes
Send the string B to the manager by sock2
Connect to the server by sock1
Success?
1) sock1 – the socket file descriptor of a socket created by the client in Figure 10
2) sock2 – the socket file descriptor of a new socket created by the library
3) structure A – a C structure contains the information in Figure 10
4) string B – a string contains variables of structure A
5) project.c – the library code

Figure 11 The Flowchart of connect_init() in the Library *project.c*

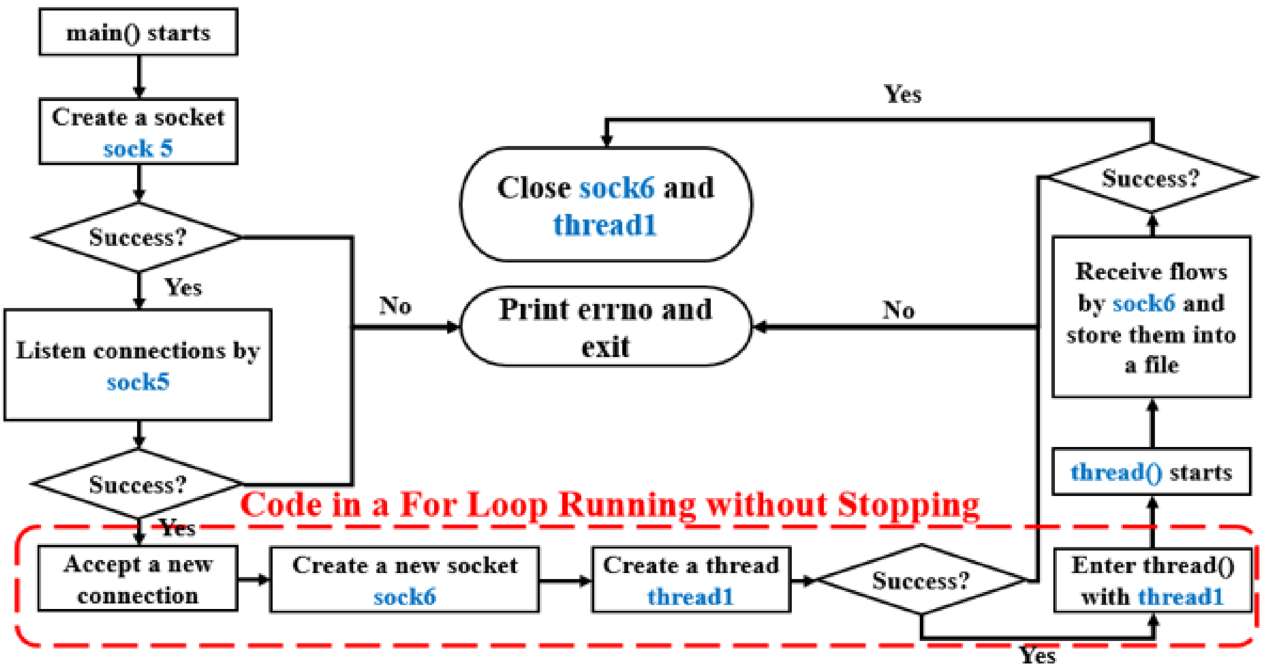

Create a socket sock 3
Success?
Yes
Bind the local address with sock3
Success?
Yes
Listen to connections
No
Close sock3
Print errno and exit
No
Success?
Yes
Code in a For Loop Running without Stopping
Accept a new connection
Create a new socket sock4
No
Close the connection of sock4 (or related files)
Yes
Success?
Receive flows by sock4 and store them
Yes
Success?

1) sock3 – the socket file descriptor of a socket used for listening to coming connections
2) sock4 – the socket file descriptor of a new socket created based on sock3. It is used for receiving flows. In this flowchart, sock4 will be created, used and closed constantly in a for loop

Figure 12 The Flowchart of the Server at Stage One

At stage two, all of terminals in stage one should be multithreaded based on 2.4.3. Threads of the client can make the client send several flows to different servers[19] in a short time. Threads of the server can make the server receive flows from different clients at the same time efficiently. Based on fundamental functions in Figure 12, Figure 13 is the main part of the flowchart of the server at stage two. Besides, I add another interface for the server.[20] This interface is called **accept_init()** **[**Prototype 2**]**, which is overridden from **accept()** [**Appendix** D]. A server can call it by a "trigger" with an integer type. When the "trigger" is set as integer 2, the server will send some network requirements to the manager (e.g. A server sends some variables to the manager, showing that "Now, I only can receive flows which are smaller than 100 Mb. Besides, I cannot deal with urgent flows".). Next, based on the rough structure of the server, the manager has one more thread to calculate and analyse received information simply. Details will be discussed in 3.2.3.

[19] In this system, because of limits of IP address and terminals of Ubuntu, the client only sends flows to a server. However, it is capable of sending to different servers.
[20] This goal is out of the scope of the project. It is an extra achievement.



1) sock5 – the socket file descriptor of a socket used for listening to coming connections
2) sock6 – the socket file descriptor of a new socket created based on sock5. It is used for receiving flows. In this flowchart, sock6 will be created, used and closed constantly in thread()
3) thread1 – the thread id of a thread. In this flowchart, thread1 will be created, used and close constantly in the for loop and thread()
4) thread() – a function used for receiving flows and storing them

Figure 13 The Main part of the Flowchart of the Server at Stage Two

3.2.2 Design of the APIs and the Protocol

Design of this section is based on 2.4.2 and 2.5. The aim to override TCP socket programming functions is to make terminals interact with the manager secretly. Thus, new functions should be similar with overridden functions. The only parameters I need to add into the functions are information for data flows. Referring to universal structures [Definition 8] (e.g. **struct sockaddr** [A]), I decide to store information for data flows into C structures with a fixed form. To make the C structure accessible for programmers, I define it in the library of my project. Also, the two new functions should return values that returned by original functions. Thus, based on prototype of **connect()** [Appendix C], I have:

Prototype 1 connect_init()

connect_init

Defined in header <project.h>

int connect_init (int sock, struct sockaddr *servaddr, int addrlen, struct extra_information this_flow)

Deals with a piece of collected information of a data-flow; Connects to the server.

Parameters:

sock - the socket descriptor

servaddr - the pointer of the struct sockaddr of the destination

addrlen - the length of the struct sockaddr of the destination

this_flow – the structure which contains information of a data-flow

Return Value

Return 0 if success

Return SOCKET_ERROR if fails

Similarly, based on **accept()** [Appendix D], I have **accept_init()**:

Prototype 2 accept_init()

accept_init

Defined in header <project.h>

int accept_init (int sock, struct sockaddr *localaddr, int addrlen, struct extra_information this_flow)

Deals with a piece of network requirement of a server; Accepts a new coming connection

Parameters:

sock - the socket descriptor

localaddr - the pointer of the struct sockaddr of the local address

addrlen - the length of the struct sockaddr of the local address

this_flow – the structure which contains network requirements of a server

Return Value

Return a new socket file descriptor if success

Return SOCKET_ERROR if fails

Comparing the two prototypes with original functions, a new structure **struct extra_information** [Prototype 3] is added into the parameters:

Prototype 3 struct extra_information

```
typedef struct extra_information{
        double flowSize;//the size of the flow in Mb
        int UrgentMode ; //the urgent level of a flow[21]
        double speed;//this is the default value of the speed, the unit is Mbps;
        char deadline[20];//the deadline of this flow
        time_t sendtime;//the transmission time of the flow
        int TransmissionMode;// the transmission mode of a communication[22]
        char *DesIP;//the ip address of the destination of the client
        int role;//the role the terminal is in the communication. 1 is a client and 2 is a server
}this_flow;
```

Variables in this structure will be parsed to a string with fixed form. It should be emphasised that terminals can choose not to fill all variables in the structure. Therefore, default values are needed for most of variables. Next, I consider the protocol used to send the information. I have considered forms of TCP, HTML, XML, and JSON. According to the analysis in 2.5, JSON is the most proper format in this project now. Above all, based on **struct extra_information**, I design a protocol for data-exchange in JSON with following default values[23]:

Prototype 4 The JSON Protocol in this Project

```
{
        "flowSize": 0.000000
        "UrgentMode": 10
        "flowSpeed": 0.000000
        "Deadline": ["NULL","NULL","NULL","NULL","NULL"]
        "TransmissionTime": ["NULL","NULL","NULL","NULL","NULL"][24]
        "TransmissionMode": 2
        "DestinationAddress": "10.0.2.15"[25]
        “Role”: 1 (or 2)[26]
}
```

[21] The urgent level should increase by degree from 0 to 9. 0 is not urgent at all and 9 is extremely urgent.
[22] The transmission mode can be 0 or 1. 1 means unidirectional. 0 means bidirectional.
[23] The default values should be initialized in the library *project.c*.
[24] Contents in “Deadline” and “TransmissionTime” should be: day of the week, month, day of the month, time of the day and year in order (e.g. ["Tue"," Jun"," 14 ","12:07:56"," 2017]).
[25] “DestinationAddress” is collected by the library directly so it does not have a default value.
[26] This variable is set by the C library automatically. Thus, there is no default value.

3.2.3 Design of the Manager

Figure 12 and 3.2.1 have illustrated the design of the simple manager at stage one roughly. This section will focus on the design of the manager at stage two. The manager has two tasks: receives JSON strings and displays information which is useful to scheduling. Firstly, based on the multithreaded server at stage one, the manager can receive flows, efficient and relatively stable. Secondly, the task is separated into two parts: find a way to parse JSON strings into C structures and design a C structure used for storing information carried by JSON strings. The way I use to parse JSON strings is JSMN [12][16]. It is an opensource JSON parser in C. It parses JSON by characters (e.g. "", {}, []). Firstly, it counts how many keys[27] in a JSON strings. Then, it splits keys and assigns values. After confirming this method is available, I design the structure **struct recvInformation [**Prototype 5**]**:

Prototype 5 struct recvInformation

```
struct recvInformation {
        double flowsize;//the size of the flow
        int urgentmode;//the urgent level of the flow
        double flowspeed;//the speed of the flow
        char ddl[20];//the deadline of the flow
        char transmissiontime[20];//the time of the client sending the flow
        int sendmode;//the transmission mode of the communication
        char serverip[25];//the ip address of the server
        char clientip[25];//the ip address of the client
        int role;// the role the terminal is in the communication
        char others[300];//handle unexpeceted contents
        }new_flow;
```

The structure is almost the same with **struct extra_information [**Prototype 3**]**. Two variables ("clientip" and "others") are added. It also has default values the same with these of **struct extra_information.** The manager initialises defaults values then uses JSMN to parse strings to this structure. Then, another thread will be used for analysing and storing analysed outputs. This thread starts to work once the program begins to run. It prints analytic outputs per minute once. [Appendix E] is a table which shows outputs of the manager.

[27] In JSMN, keys of JSON are called "tokens".

Although flows of JSON strings are very small, the manager needs to deal with volume flows in a short time. Thus, it is better to reduce the burden of the manager as much as possible. Meaning, the point is making outputs as much meaningful as possible. For some variables, the manager should record them both in total and in per minute (e.g. total sizes of flows). For others, it is not necessary to record them in both recording periods (e.g. TransmissionMode[28]).

3.3 Implementation

3.3.1 Implementation of the APIs

To implement the APIs, there are two tasks: write the library code; write a C header files to link functions and variables. In the library, there are two functions: **connect_init() and parse_json()** **[**Prototype 6**]**[29]. Figure 14 is the flowchart of **connect_init()** [Prototype 1] in the library[30]. At the stage of declaration, the code defines an integer "ConnectReturn" whose default value is -2. This integer is the return value of **connect_init()**. Every time **connect_init()** uses **connect()** [Appendix C], "ConnectReturn" is assigned the return value of **connect()**. **parse_json()** [Prototype 6] uses **sprintf()**[31] to write the C structure **this_flow** [Prototype 3] into a buffer one by one in designed JSON format [Prototype 4]. To meet requirements of *project.c*, the C header file *project.h* defines a global string "JSONBuffer" with the size of 1024. in *project.h*. This string is used for storing parsed JSON strings and it is the buffer to be sent to the manager.

Prototype 6 parse_json()

parse_json

Defined in header <project.h>

void parse_json(struct extra_information newflow);

Parses struct extra_information newflow to a JSON string.

Parameters:

newflow – a structure storing information of data flows

No Return Values

[28] According to my research so far, transmission mode is not an important parameter which will affect scheduling and the performance of networks. Besides, in computer systems, transmission modes of most computers are fixed. The reason to record it in total is that it is helpful to grasp general communications as much as possible. Also, these data are possible to be used in future study or usage.

[29] **parse_json()** can also be used by **accept_init()**.

[30] Because the overriding way of **connect_init()** and **accept_init()** are very similar, and **accept_init()** is out of the scope of the project. In this section, I only discuss the implementation of **connect_init()** in detail.

[31] A function in *stdio.c*. It can write formative data into a string.

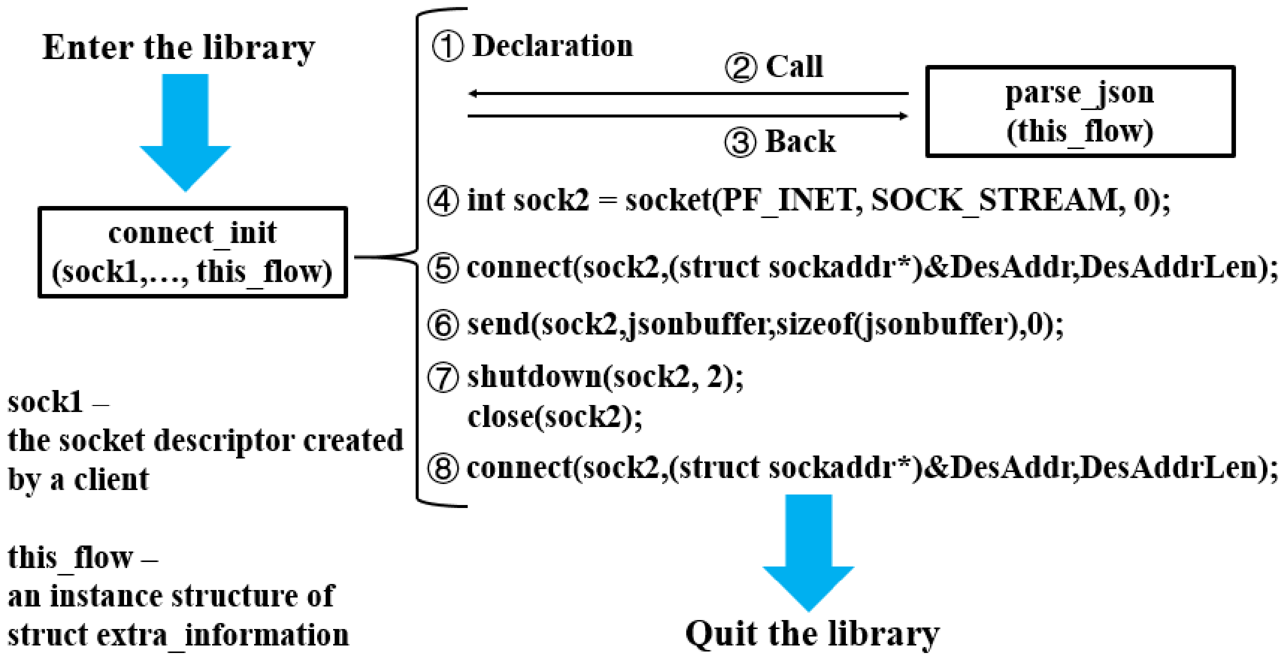

Enter the library
connect_init (sock1,…, this_flow)
① Declaration
② Call
③ Back
parse_json (this_flow)
④ int sock2 = socket(PF_INET, SOCK_STREAM, 0);
⑤ connect(sock2,(struct sockaddr*)&DesAddr,DesAddrLen);
⑥ send(sock2,jsonbuffer,sizeof(jsonbuffer),0);
⑦ shutdown(sock2, 2); close(sock2);
⑧ connect(sock2,(struct sockaddr*)&DesAddr,DesAddrLen);
sock1 – the socket descriptor created by a client
this_flow – an instance structure of struct extra_information
Quit the library

Figure 14 The flowchart of connect_init() in the Library

3.3.2 Implementation of the Manager

There are four functions in *manager.c*. Figure 15 shows the structure of the manager. Firstly, **main()** initialises pairs of values by default values. Those values are used for collecting or generating variables in [Appendix E]. Then, **main()** creates “thread2” and makes “thread2” start to work. Since that time, **thread2()** begins to calculate and store variables shown in [Appendix E] constantly. **thread2()** also displays these variables on the display screen of the manager per minute. At the meantime, after **thread2()** is working, **main()** creates a socket for listening. Once it listens to a new connection, it creates another socket and a related thread, then enter **thead1()**. **thread1()** is responsible to receive flows and record variables of flows into **struct recvInformation [**Prototype 5**]**. Every time **thread1()** has received a buffer, it calls **parse_struct()**. **parse_struct()** is a function contains code in JSMN. Every time **parse_struct()** parses a pair of “key: value” in a JSON sting, it stores the value into the certain variable in **struct recvInformation**. Above all, **struct recvInformation** is used by not only one function. Therefore, it should either be in *project.h* or be a global structure in *manager.c*. Because of the uncertainty of collected information of flows, I determined to make this structure only a global structure. If I defined it in the C header file, it would be not so comfortable for me to modify it.

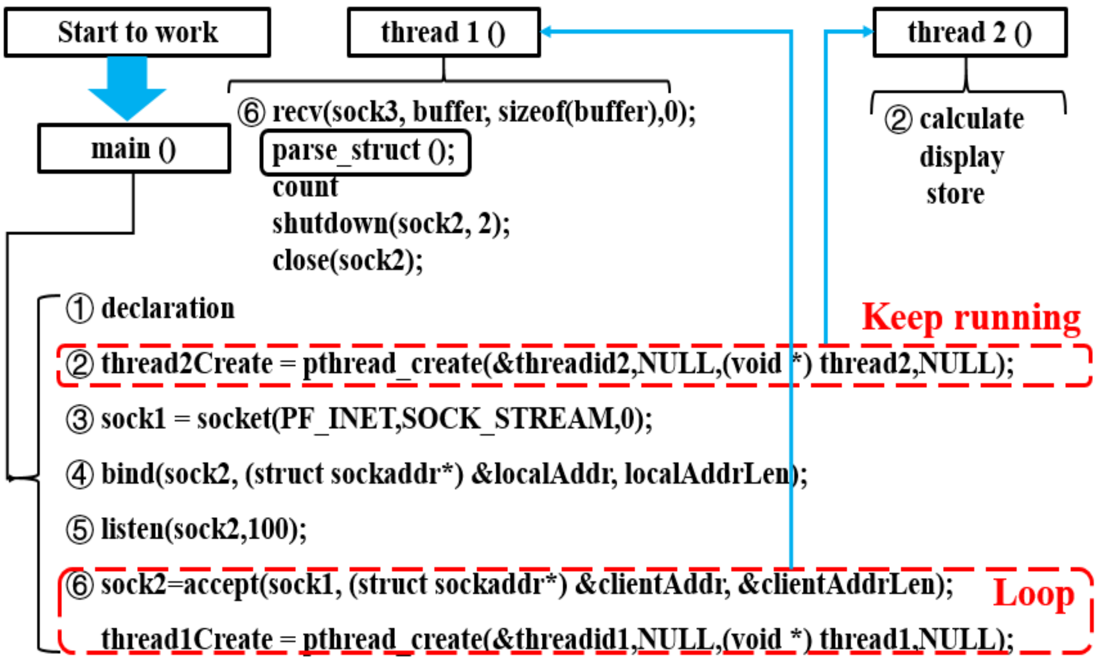

Start to work
main ()
thread 1 ()
⑥ recv(sock3, buffer, sizeof(buffer),0);
parse_struct ();
count
shutdown(sock2, 2);
close(sock2);
thread 2 ()
② calculate
display
store
① declaration
Keep running
② thread2Create = pthread_create(&threadid2,NULL,(void *) thread2,NULL);
③ sock1 = socket(PF_INET,SOCK_STREAM,0);
④ bind(sock2, (struct sockaddr*) &localAddr, localAddrLen);
⑤ listen(sock2,100);
⑥ sock2=accept(sock1, (struct sockaddr*) &clientAddr, &clientAddrLen);
thread1Create = pthread_create(&threadid1,NULL,(void *) thread1,NULL);
Loop

Figure 15 The Structure of the Manager[32]

3.4 Testing

3.4.1 Testing of the APIs and the Protocol

In this section, I will discuss according to stages listed in 3.2.1. At stage one, testing the API [Prototype 1] and the protocol [Prototype 4] is very simple. I need to test whether they are available and reasonable for programmers. After studying JSMN, I modified my protocol several times to make it meet the format of JSMN. During stage two, I have noticed, obviously, information collected into **struct extra_information** was not enough. Thus, I started to review and read relevant papers, adding more variables into the structure.[33] This kind of testing makes collected information for flows more valuable. Besides, I added another API [Prototype 2] and had tested it.

3.4.2 Testing of the Manager and the Server

To test the manager and the server, I used many different clients. Intentionally, I had kept every kind of clients I wrote through my project. Some clients are single-threaded, some are multi-threaded, some can collect information for flows manually, and some can generate a large number of “fake information” in a short time. Under Ubuntu12.04, I can only open six terminals. Thus, I used one terminal for the manager, one for the server, others all for various clients. Usually I ran the manager

[32] To make the code more clear and well-organised, I take functions (e.g. calculating sizes and average speeds of flows) apart from **thread2()** and put them into little C functions. The working mechanism is the same with Figure 15.

[33] **struct extra_information** only has three variables at the beginning. Now it has eight variables.

and the server for 4-5 hours every day. In terms of the longest running time, the manager and the server had run for 14 hours stably[34]. Both the manager and the server work well now.

3.4.3 Testing of the Whole System

Firstly, the system should be error-free and warning-free. After some modifications, I have achieved this. Secondly, the system can work stably. After a long-term running, clients, the server, and the manager all can work well. Thirdly, repeated code should decrease. I have checked to ensure most code appears as less as possible. Besides, I added more annotations to make the code more readable.

3.4.4 Testing of the Outputs of the Manager

The manager is the most important thing in this project. Outputs of it [Appendix E] should be vital. To test the outputs, on one hand, the manager should guarantee they are correct and well-stored. On another hand, the manage should display useful information as much as possible.

I stored received JSON strings in different places and matched them and analytic results. This is the simplest way to test. According to Figure 15, in **parse_struct()**, I stored parsed C structures [Prototype 5] into a text file with numbers of ingress of each flow. The text file is called *StructureInfo.txt*. Besides, I also stored those structures into an excel file to match outputs quickly. The file is called *StructureInfo.xls*. After that, in **thread1()**, I stored received JSON strings as a kind of raw materials into different text files, according to their urgent levels. These text files are *Series RawMaterials.txt*.[35] Also, I stored the display data for the manager in *RecvInfo.txt*. Combined with the text file which stores display information for the manager, I checked whether numbers of flows in total, average speeds of flows and parameters of random flows are matching.

Next I will give an example of testing the outputs of the manager.

Firstly, in “RecvInfo.txt”, I select information of flows in a minute randomly:[36]

------------Thu May 11 22:04:47 2017

38 Flows in Total

[34] The longest testing is from 22:04:47 11 May 2017 to 12:09:46 12 May 2017.

[35] The urgent level of a flow is classified into eleven levels. Level 9-8 is “extremely urgent”, Level 7-5 is “urgent”, Level 4-1 is “not urgent”, Level 0 is “the client does not set the urgent level” and Level 10 is “the server cannot deal with urgent flows”. According to urgent levels, “extremely urgent” flows are stored in *RawMaterials1.txt*, “urgent” flows are stored in *RawMaterials2.txt*, “not urgent” flows are in *RawMaterials3.txt* and other flows are in *RawMaterials4.txt*.

[36] Analysing all data in the report is time-consuming and meaningless. Thus, I select some variables in the data to show the testing outcomes. The pasted content is only part of the entire display data in the minutes.

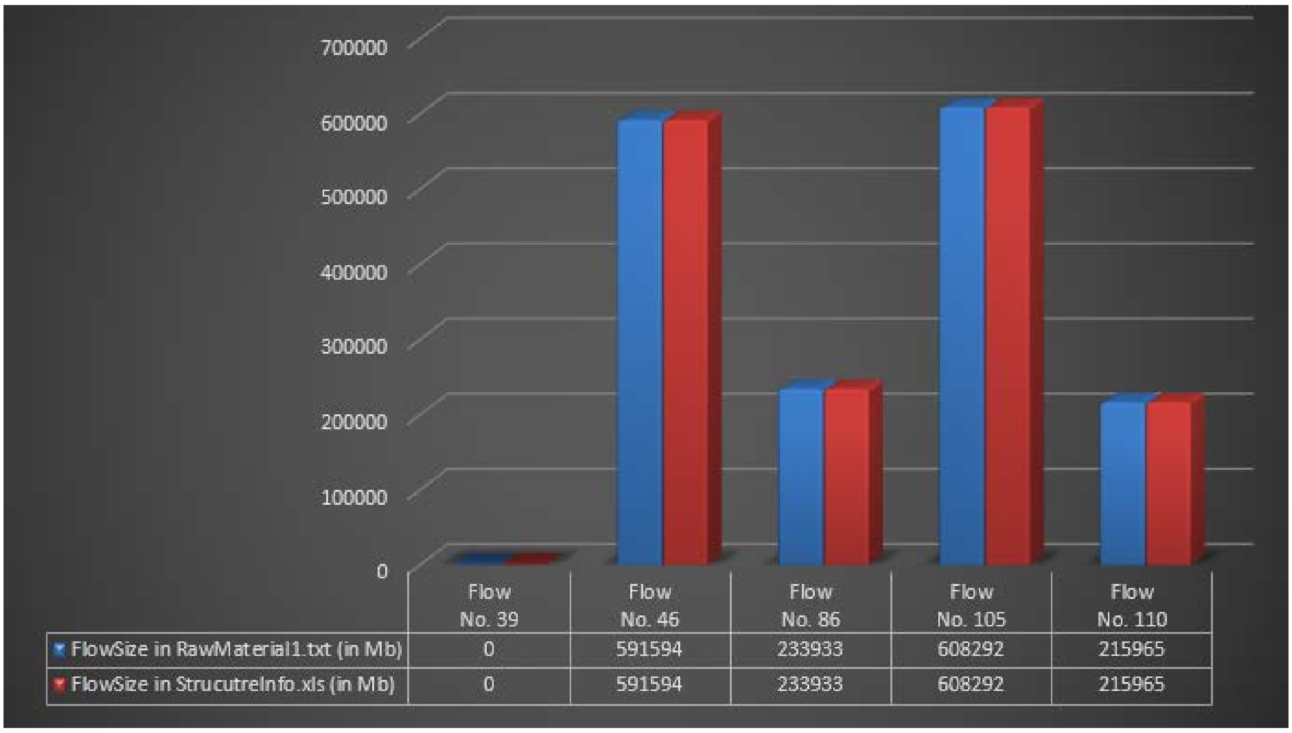
```
…
------------Thu May 11 22:05:47 2017
121 Flows in Total
…

83 Flows in the minute
70 Flows are Communication Flows in the minute
…
The Average Speed of Flows in the minute: 68943.737705 Mbps
…
The Number of Flows whose speeds are higher than 5Gps: 29
The Total Size of Flows in the minute: 29.763788 Tb
…
The Number of Servers which cannot receive urgent flows:6
The Number of Flows which are Extremely Urgent in the minute:10
…
```

I find the set of data should belong to flows received by the manager from “Thus May 11 22:04:47 2017” to “Thus May 11 22:05:47 2017”. Compared with “XXX Flows in Total” in the two minutes, the set flows are from “Flow No. 39” to “Flow No. 121”.

Then, to test whether recorded data are accurate, I select the same 5 flows in this minute from *Strucutre.xls* and *RawMaterial1.txt*[35] separately. Sizes of flows in *RawMaterial1.txt* are from original JSON strings from terminals. Sizes of flows in *Strucutre.xls* are from C structures [Prototype 5] parsed from JSON strings by the manager. Figure 16 is the comparative result. Figure 16 shows that sizes of flows received and parsed by the manager are exactly the same. Besides, owing to that I have various files which stores data of flows depending on different standards, I only use 30 seconds to match these 5 flows in *RawMaterial1.txt* and *Strucutre.xls.*

Thirdly, to test whether analytic outputs of the manager are accurate, I select the related 83 flows in *StructureInfo.xls* (in above paragraph, I have proved data in *StructureInfor.xls* are available and correct). Then, I select and calculate some variables by excel. Figure 17 is the comparative result of outputs on the display screen of the manager and outcomes of StructureInfo.xls. Data of “The

Average Speed in the minute" and "The total size of Flows in the minute" are approximate numbers. In the real testing, the two floats are the same to the first six decimal places.



	Flow No. 39	Flow No. 46	Flow No. 86	Flow No. 105	Flow No. 110
🟦 FlowSize in RawMaterial1.txt (in Mb)	0	591594	233933	608292	215965
🟥 FlowSize in StrucutreInfo.xls (in Mb)	0	591594	233933	608292	215965

Figure 16 Comparisons of the Variable "FlowSize" of some Flows

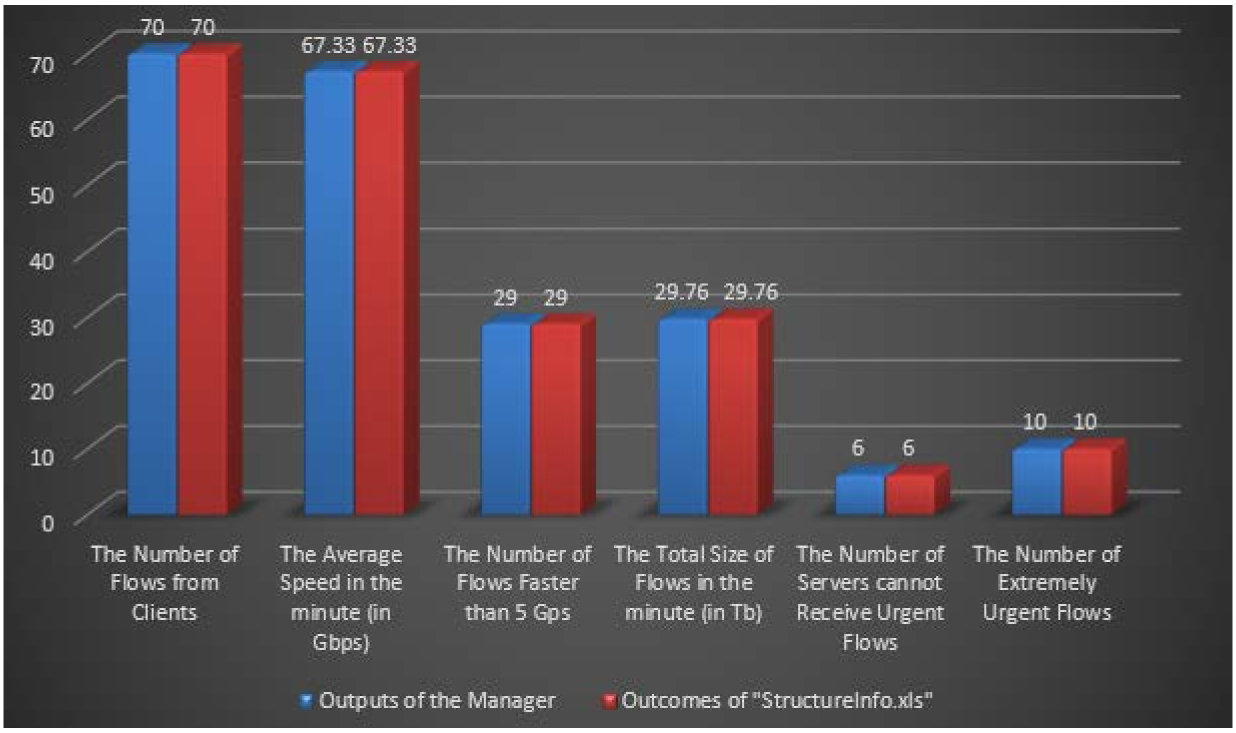


Figure 17 Comparisons of outputs of the manager and *StrucutreInfo.xls*

Chapter 4 Results and Discussion

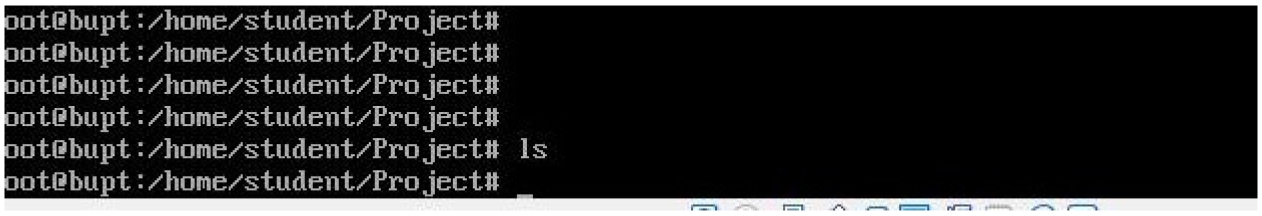
4.1 Literature Review

I have written a literature review before the mid-term check. Now, I have enhanced the knowledge of data centres, TCP, TCP socket programming, and C programming under both Windows and Linux. I have improved my ability to use online opensource tools (e.g. GitHub) efficiently. I am getting to learn about flow scheduling and some methods designed by researchers to solve scheduling and debugging problems (e.g. Varys) of data centres. This helps me to solve a variety of problems in my project and inspires me gradually.

4.2 Programming

This project is to create a simple system in which the manager can collect pieces of information for data flows from terminals. It mainly has four outputs in the system. I will show the outputs by a programming example of this project.

Output 1 This system has a manager.[37] This manager is one of cores of the project. It is for collecting information for data flows, displaying and recording some useful information into a directory "/home/student/Project" ("Project"). Before starting the program, I use a Linux command "ls" to display files in "Project", Figure 18 shows there is no files in the directory.



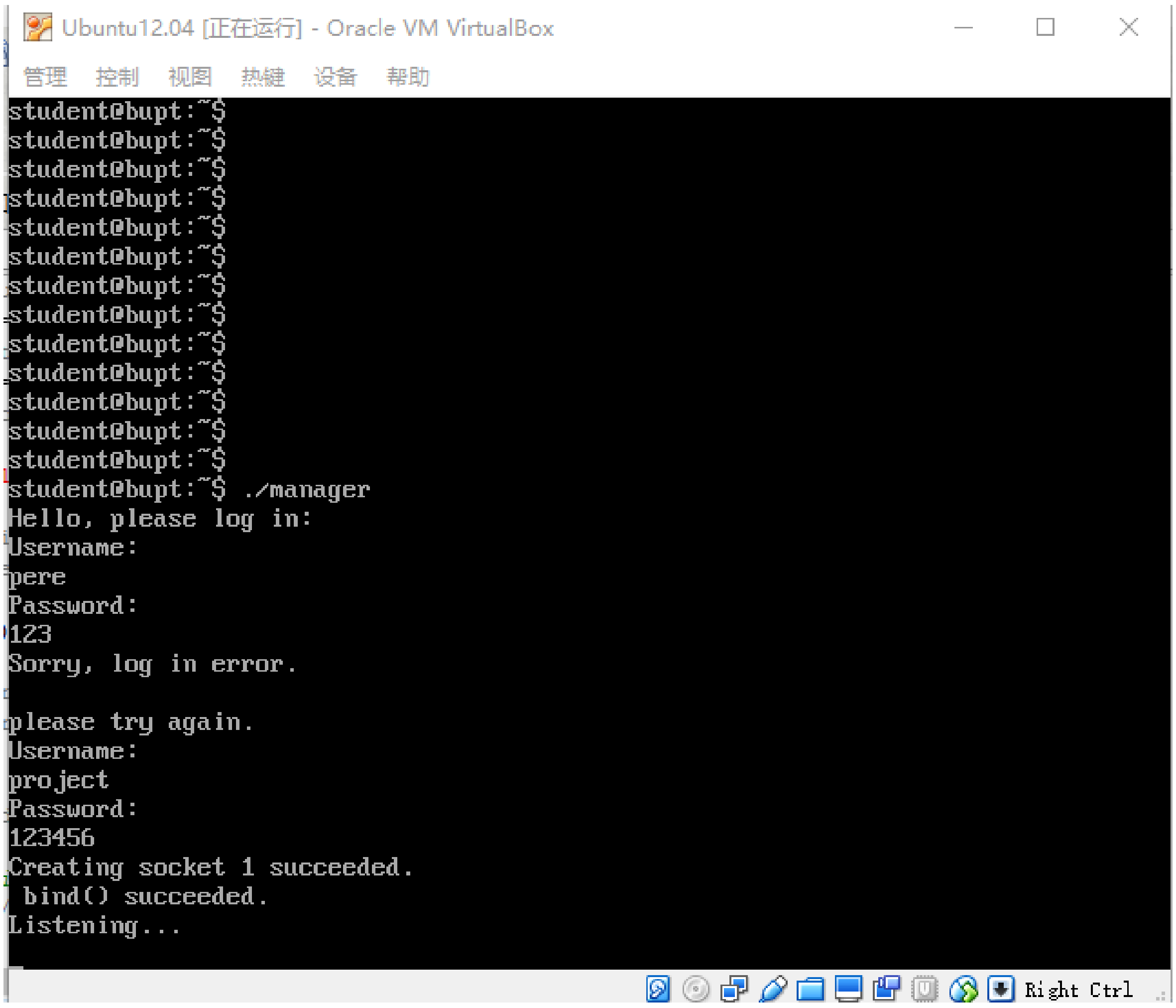
```
oot@bupt:/home/student/Project#
oot@bupt:/home/student/Project#
oot@bupt:/home/student/Project#
oot@bupt:/home/student/Project#
oot@bupt:/home/student/Project# ls
oot@bupt:/home/student/Project# _
```

Figure 18 Files under "home/student/Project" before starting the manager

Then, Figure 19 is starting the manager. To start with, there is a very simple login function for this manager. A user must input a right username and a corresponding password, and then he can use the manager. In this example, the username is "project" and the password is "123456". The reason to have this function is that the central manager should be equipped with some confidential measures. In a data centre, a manager can collect pieces of information for various flows, including those

[37] The manager contains about 950 lines of code.

should be kept secret. However, this is not the point of the project. Thus, I only write an extremely simple login function to show this idea.

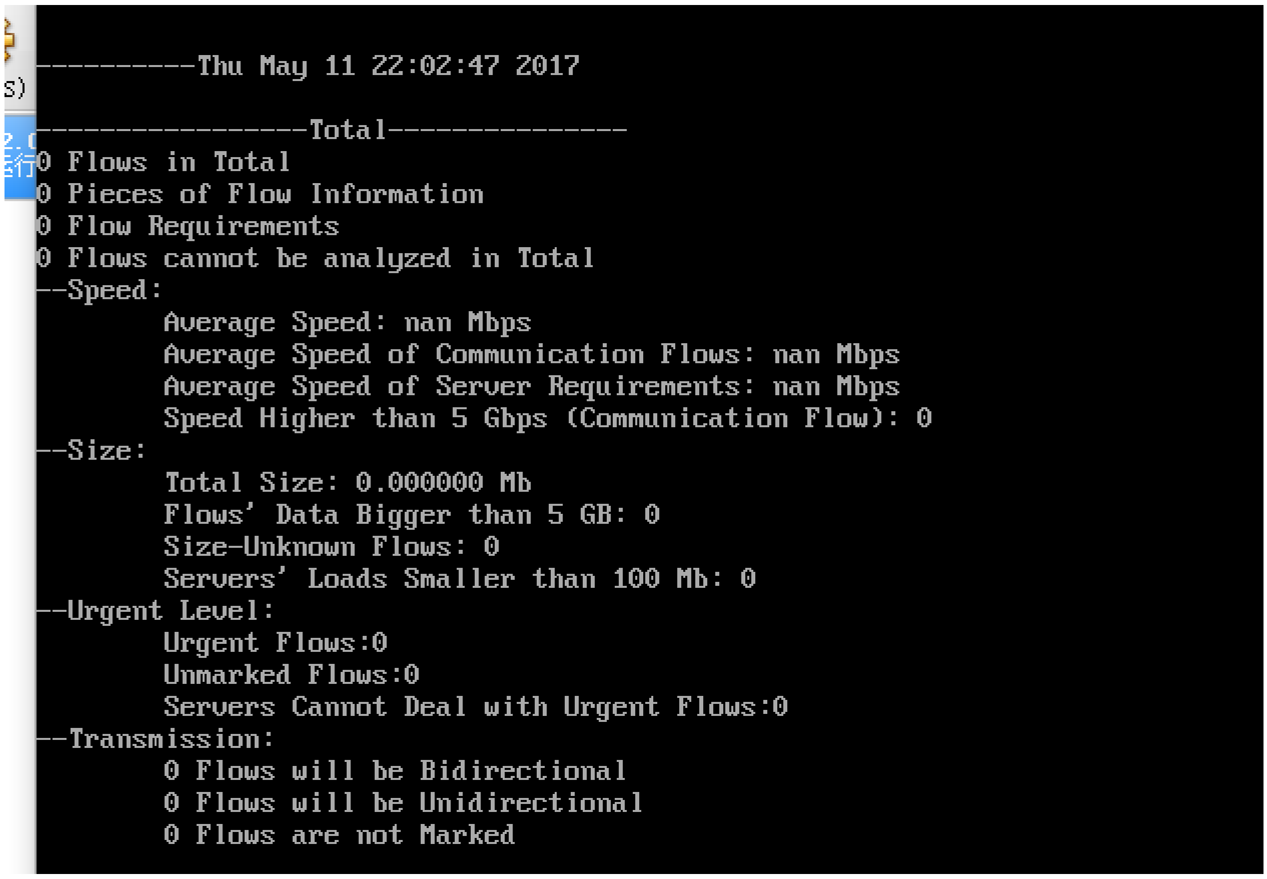

Ubuntu12.04 [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
Right Ctrl

```
student@bupt:~$
student@bupt:~$
student@bupt:~$
student@bupt:~$
student@bupt:~$
student@bupt:~$
student@bupt:~$
student@bupt:~$
student@bupt:~$
student@bupt:~$
student@bupt:~$
student@bupt:~$
student@bupt:~$
student@bupt:~$ ./manager
Hello, please log in:
Username:
pere
Password:
123
Sorry, log in error.

please try again.
Username:
project
Password:
123456
Creating socket 1 succeeded.
 bind() succeeded.
Listening...
```

Figure 19 Starting the manager

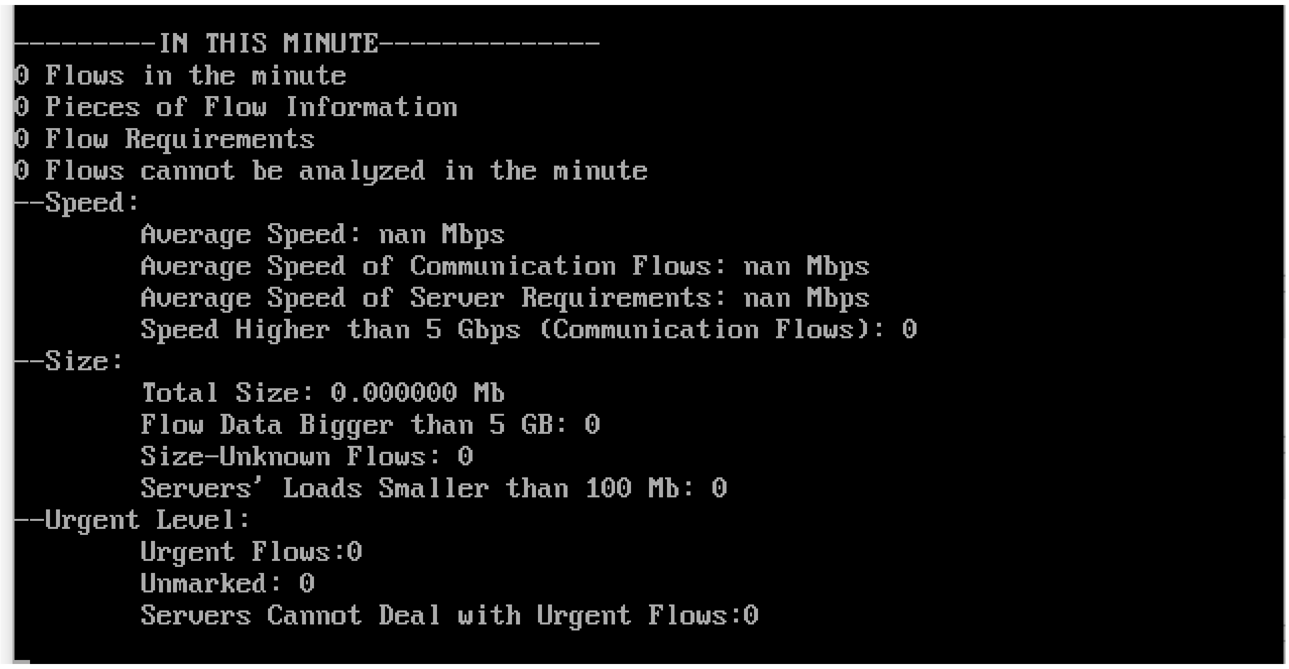
After starting the manager for a minute, the manager display pieces of information [Appendix E] which are about collected information for data flows (at 22:01:47 11 May 2017, BJT)[38]. Figure 20 is the screenshot of the display screen of the manager. “Total” means from the starting of the manager till “now”. Figure 21 is the screenshot display data in the minute from “22:01:47 11 May 2017, BJT” to “22:02:47 11 May 2017, BJT”. Because there is no coming flow, most variables are “0” or “nan”[39].

[38] Beijing Time (BJT) is only used in the display screen as a local time to make users easy to control time. Other times in the system are all Universal Time Coordinated (UTC).
[39] “nan” means “Not a Number”. The reason why “nan” occurs is that in these calculations, the denominators are 0 temporarily.



```
----------Thu May 11 22:02:47 2017

-----------------Total---------------
0 Flows in Total
0 Pieces of Flow Information
0 Flow Requirements
0 Flows cannot be analyzed in Total
--Speed:
        Average Speed: nan Mbps
        Average Speed of Communication Flows: nan Mbps
        Average Speed of Server Requirements: nan Mbps
        Speed Higher than 5 Gbps (Communication Flow): 0
--Size:
        Total Size: 0.000000 Mb
        Flows' Data Bigger than 5 GB: 0
        Size-Unknown Flows: 0
        Servers' Loads Smaller than 100 Mb: 0
--Urgent Level:
        Urgent Flows:0
        Unmarked Flows:0
        Servers Cannot Deal with Urgent Flows:0
--Transmission:
        0 Flows will be Bidirectional
        0 Flows will be Unidirectional
        0 Flows are not Marked
```

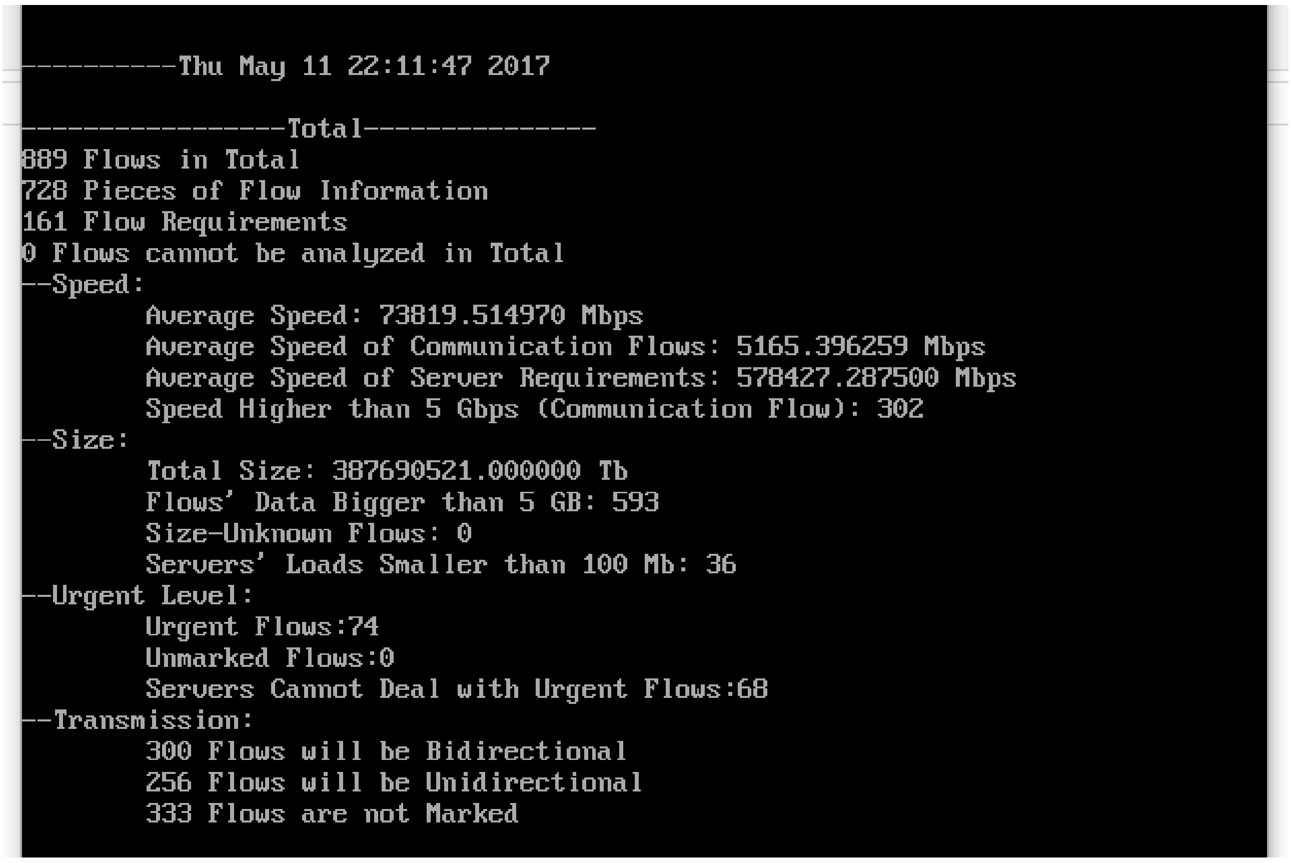
Figure 20 Display Information of the manager in the first minute (1)



```
---------IN THIS MINUTE--------------
0 Flows in the minute
0 Pieces of Flow Information
0 Flow Requirements
0 Flows cannot be analyzed in the minute
--Speed:
        Average Speed: nan Mbps
        Average Speed of Communication Flows: nan Mbps
        Average Speed of Server Requirements: nan Mbps
        Speed Higher than 5 Gbps (Communication Flows): 0
--Size:
        Total Size: 0.000000 Mb
        Flow Data Bigger than 5 GB: 0
        Size-Unknown Flows: 0
        Servers' Loads Smaller than 100 Mb: 0
--Urgent Level:
        Urgent Flows:0
        Unmarked: 0
        Servers Cannot Deal with Urgent Flows:0
```

Figure 21 Display Information of the manager in the first minute (2)

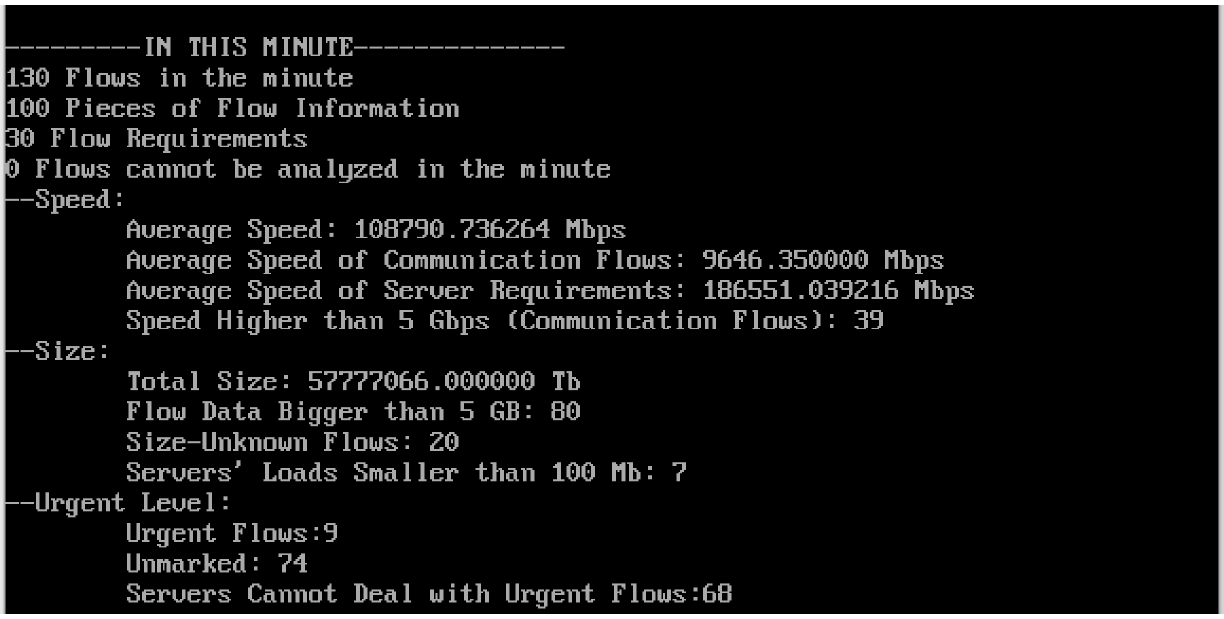
Then I start the server then clients. After nine minutes (22:11:47 11 May 2017, BJT), I have two screenshots of a running manager. Figure 22 is about display data in total. Compared with the beginning, there are 889 flows sent to the manager. 728 flows are from clients and 161 are from servers.[40] In Figure 23, from “22:10:47” to “22:11:47”, there are 130 flows sent to the manager.



```
----------Thu May 11 22:11:47 2017

-----------------Total---------------
889 Flows in Total
728 Pieces of Flow Information
161 Flow Requirements
0 Flows cannot be analyzed in Total
--Speed:
        Average Speed: 73819.514970 Mbps
        Average Speed of Communication Flows: 5165.396259 Mbps
        Average Speed of Server Requirements: 578427.287500 Mbps
        Speed Higher than 5 Gbps (Communication Flow): 302
--Size:
        Total Size: 387690521.000000 Tb
        Flows' Data Bigger than 5 GB: 593
        Size-Unknown Flows: 0
        Servers' Loads Smaller than 100 Mb: 36
--Urgent Level:
        Urgent Flows:74
        Unmarked Flows:0
        Servers Cannot Deal with Urgent Flows:68
--Transmission:
        300 Flows will be Bidirectional
        256 Flows will be Unidirectional
        333 Flows are not Marked
```

Figure 22 The Screen of a Running Manager (1)

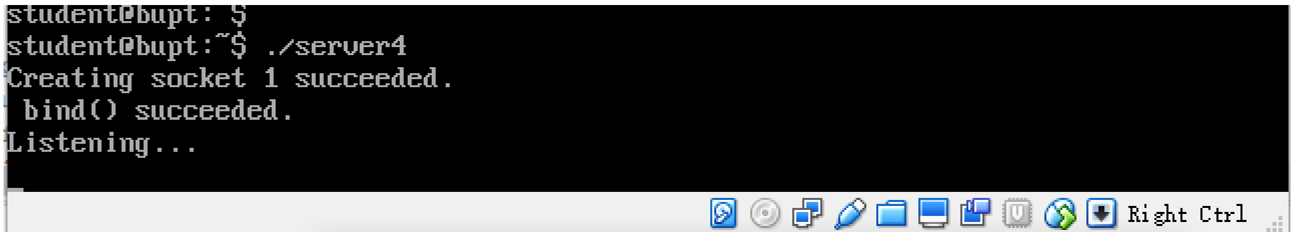
[40] There is only a server in this example but this can also work with many servers. I need to get a large number of flows in a short time efficiently, however, because of Ubuntu terminals limit, I use a server send requirements with high sending frequency.



```
---------IN THIS MINUTE--------------
130 Flows in the minute
100 Pieces of Flow Information
30 Flow Requirements
0 Flows cannot be analyzed in the minute
--Speed:
        Average Speed: 108790.736264 Mbps
        Average Speed of Communication Flows: 9646.350000 Mbps
        Average Speed of Server Requirements: 186551.039216 Mbps
        Speed Higher than 5 Gbps (Communication Flows): 39
--Size:
        Total Size: 57777066.000000 Tb
        Flow Data Bigger than 5 GB: 80
        Size-Unknown Flows: 20
        Servers' Loads Smaller than 100 Mb: 7
--Urgent Level:
        Urgent Flows:9
        Unmarked: 74
        Servers Cannot Deal with Urgent Flows:68
```

Figure 23 The Screen of a Running Manager (2)

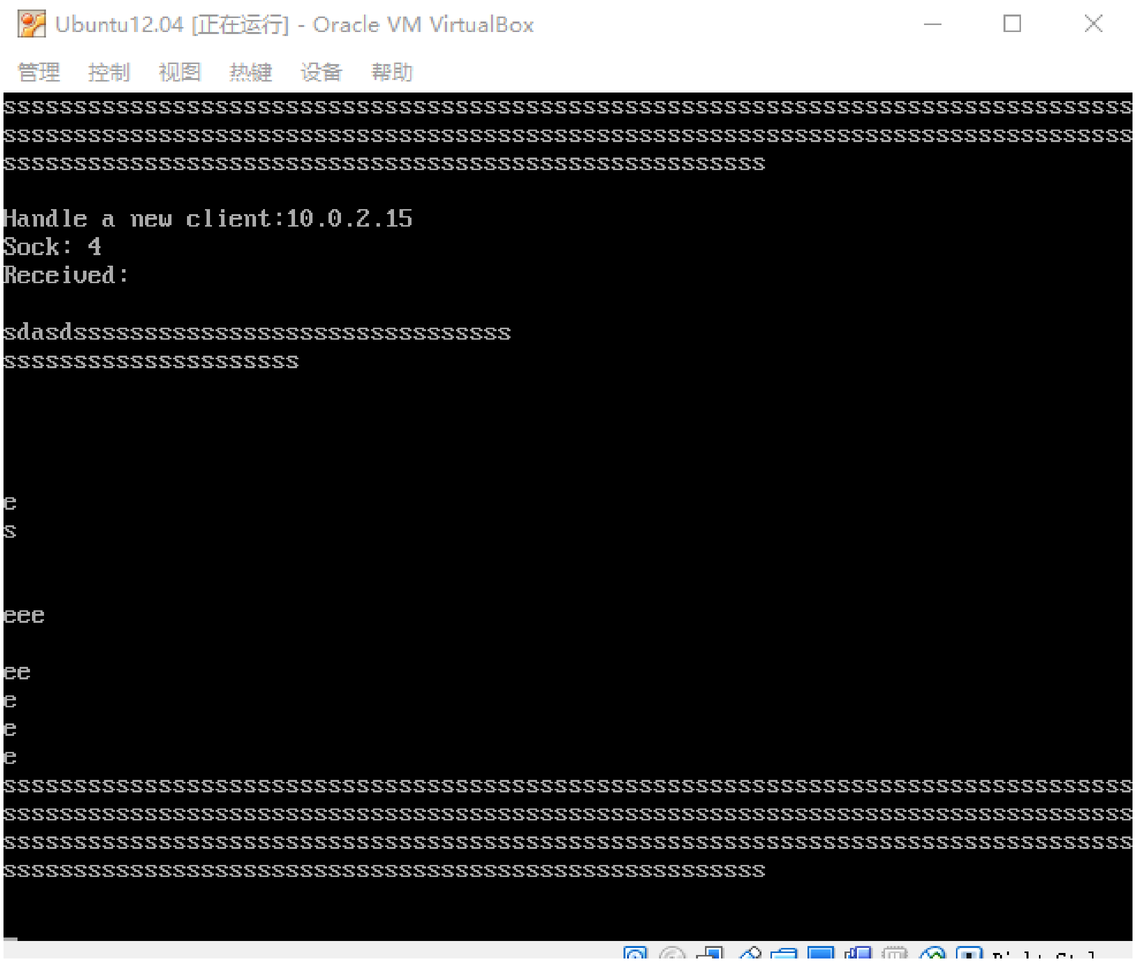
Output 2 This project has a multithreaded server.[41] The server can receive data flows from many clients at the same time, store data into “ServerReceive.txt”. Besides, it can send network requirements to the manager. Figure 24 is starting the server. Figure 25 is a screenshot of the screen of the running server. To make the receiving process visible, the server prints received flows on the screen directly with “Handle a new client: 10.0.2.15”, its socket file descriptor “Sock: 4”, and “Received:” firstly. Also, as shown in Figure 26, when the server sends its network requirements to the manager, it prints them on the screen in JSON forms [Prototype 4] directly. However, only keys “flowSize”, “UrgentMode”, “flowSpeed”, and “Role” are available for servers. If a server wants to limit sizes of coming flows, it can set a value to “flowSize” and send it to the manager with the whole JSON string. Similarly, if a server cannot receive urgent flows anymore, it can set “UrgentMode” as 10. In this case, the server only set “flowSize” as 293,747.00. This means the server can only receive flows which are smaller than 293,747.00 Mb.



```
student@bupt: $
student@bupt:~$ ./server4
Creating socket 1 succeeded.
 bind() succeeded.
Listening...
```

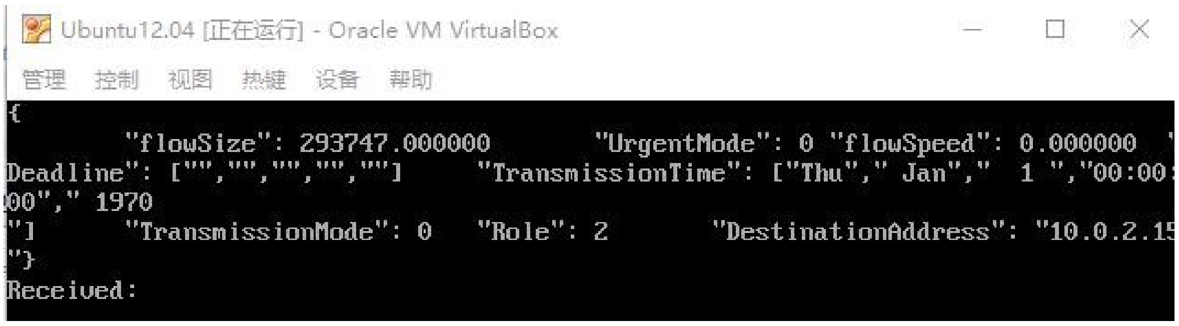
Figure 24 Starting the Server

[41] The server contains about 150 lines of code.



```
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssss

Handle a new client:10.0.2.15
Sock: 4
Received:

sdasdsssssssssssssssssssssssssssssss
sssssssssssssssssssss





e
s


eee

ee
e
e
e
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssss


_
```

Figure 25 The Screen of a Running Server (1)

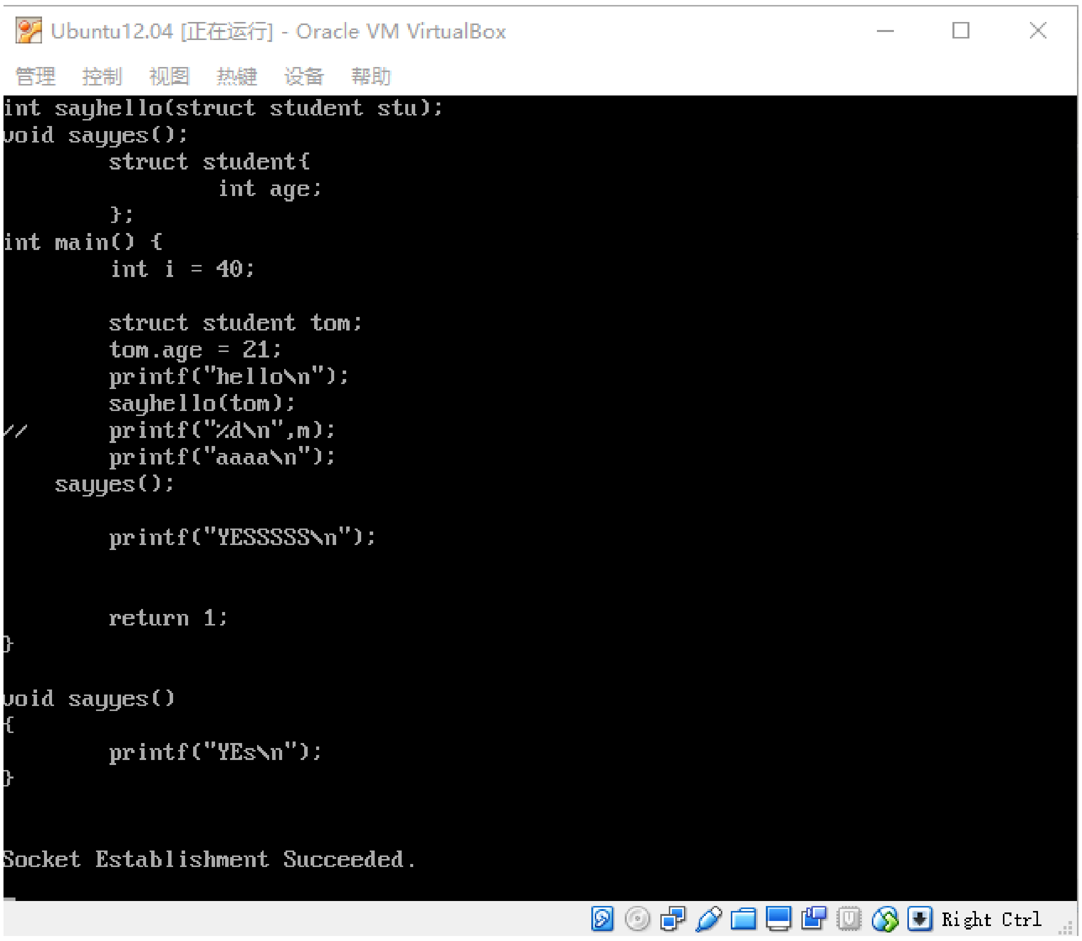


```
{
        "flowSize": 293747.000000       "UrgentMode": 0 "flowSpeed": 0.000000  "
Deadline": ["","","","",""]     "TransmissionTime": ["Thu"," Jan","  1 ","00:00:
00"," 1970
"]      "TransmissionMode": 0   "Role": 2       "DestinationAddress": "10.0.2.15
"}
Received:
```

Figure 26 The Screen of a Running Server (2)

Output 3 This project has a multithreaded client.[42] This client can send flow s to a server. In this example, I start three clients in turn. Figure 27 is one of them. It sends the content in a C file to a server every second. Similarly, to show the progress clearly, clients print sent flows with "Socket Established Succeeded" printed firstly.

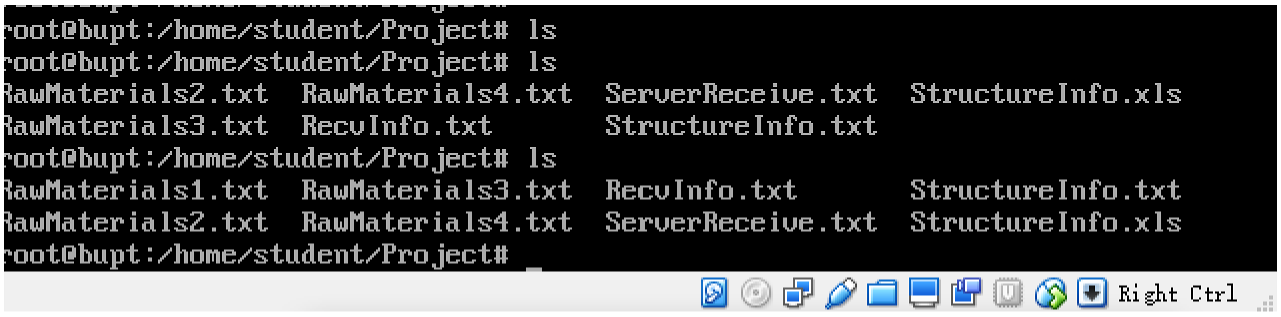
[42] Each client contains about 210 lines of code.


Ubuntu12.04 [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
Right Ctrl

```
int sayhello(struct student stu);
void sayyes();
        struct student{
                int age;
        };
int main() {
        int i = 40;

        struct student tom;
        tom.age = 21;
        printf("hello\n");
        sayhello(tom);
//      printf("%d\n",m);
        printf("aaaa\n");
    sayyes();

        printf("YESSSSS\n");


        return 1;
}

void sayyes()
{
        printf("YEs\n");
}


Socket Establishment Succeeded.
```

Figure 27 The Screen of a Running Client

Now, as shown in Figure 28, using the command “ls”, there are some new files. There are some files created by the manager. Series *RawMaterials* are created by the manager to store received JSON strings, according to different urgent levels[35]. *RecvInfo.txt* is for storing display information of the manager every minute. Series *StrucutreInfo* are for storing C structures [Prototype 5] parsed from JSON strings by the manager. There is a file created by the server. *ServerReceive.txt* is for storing received flows by the server.



```
root@bupt:/home/student/Project# ls
root@bupt:/home/student/Project# ls
RawMaterials2.txt  RawMaterials4.txt  ServerReceive.txt  StructureInfo.xls
RawMaterials3.txt  RecvInfo.txt       StructureInfo.txt
root@bupt:/home/student/Project# ls
RawMaterials1.txt  RawMaterials3.txt  RecvInfo.txt       StructureInfo.txt
RawMaterials2.txt  RawMaterials4.txt  ServerReceive.txt  StructureInfo.xls
root@bupt:/home/student/Project# _
```

Figure 28 Files in "/home/student/Project"

Output 4 Besides the visible working system, the project has a C library which stores some TCP socket interfaces for a data centre.[43] This library is another core of the project. Mainly, it has a C structure [Prototype 3], two functions called **connect_init()** **[**Prototype 1**]** and **accept_init()** **[**Prototype 2**]**. **connect_init()** is responsible to make clients send information for data flows to the manager. **accept_init()** is for making servers send network requirements to the manager. This library can continue to expand for future usage.

[43] The library contains about 300 lines of code (a C header file included).

Chapter 5 Conclusion and Further Work

5.1 Conclusion

This project aims at doing researches on scheduling of flows in data centres and creating a simple system which can reflect information for data flows roughly. Substantially, I have:

1. Knowledge of data centres, scheduling, existing solutions to scheduling problems of a data centre (e.g. Varys, SDN, ADDA in 2.2), and TCP socket programming.
2. A simple working system including different kinds of clients, a multithreaded server, and a multithreaded manager.
3. APIs with their related library and a C header file.
4. A protocol for data-exchange in JSON.
5. Ways of parsing a C structure to a JSON string and parse it back.

There are some points of doing this project. The most vital points are to understand the mechanism of this system and how to design it. Besides, techniques of coding with C under Linux and Windows play a vital role in implementing this project. Designs must be checked and improved by implementations.

5.2 Further Work

I need to consider some detailed design of the project. To start with, there is a problem of repeating sending. Assume that, a client fails to connect to a server. However, it has finished the process of sending a data flow information to the manager. When it recalls **connect_init()** [Prototype 1], the information for the data flow will be sent to the manager again. Data recorded by the manager will be inaccurate. This problem can be solved easily by clients and the library: *project.c* can make the client continue to connect to a server until it successes. However, this solution is not ideal for a practical system. The reason is, in a real system, the manager cannot re-configure networks after it receives the information for the data flow. Before it tries to take some measures, the flow has already been sent to the server. If I want to solve the problem by the manager, there should be a more complicated design of the manager. Secondly, if the central manager is capable of controlling networks in a large distributed data centre, it cannot meet the feature of decentration in a distributed system. The possible solution is to create a topology of a set of distributed managers. Each manager is central in its working range. Thirdly, in some cases, data of the manager could be significant and confidential. Ways of storing such kind of data should be well-designed.

I need to improve the working system. This system is still extremely simple. To make the solution of this project 1.2 persuasive, firstly, this system should be more similar with practical working systems in a data centre. This requires stability of the manager and complexity of clients and servers. Secondly, the way of storing and dealing with the received JSON strings is very primary. More advanced tools of data analysis should be involved (e.g. MySQL). Thirdly, this system should be tested by thousands of communications to ensure it is stable. It should have a variety of handing unexpected situations (e.g. Client A sends a large number of parameters to the manager. However, these parameters are not in the range of the collection by the manager).

References

[1] Chowdhury, M. Zhong and Y. Stoica, I. (2014) 'Efficient Coflow Scheduling with Varys', Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14), ACM, New York, NY, USA, 443-454. doi: 10.1145/2619239.2626315

[2] Zamfir, C. Altekar, G and Stoica, I. (2013) 'Automating the Debugging of Datacenter Applications with ADDA', Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference, Budapest, Hungary, 1-12 June. doi: 10.1109/DSN. 2013.6575303.

[3] Stoica, I. Abdel-Wahab, H. Jeffay, K. Baruah, S.K. Gehrke, J.E. and Plaxton, C.G. (1996) 'A Proportional Share Resource Allocation Algorithm for Real-Time, Time-shared Systems', Real-Time Systems Symposium, 1996, 17th IEEE, Washington, DC, USA, USA, 288-299 December. doi: 10.1109/REAL 1996.563725

[4] Habibi, P. Mokhtari, M and Sabaei, M. (2016) 'QRVE: QoS-Aware Routing and Energy-Efficient VM Placement for Software-Defined DataCenter Networks', Telecommunications (IST), 2016 8th International Symposium, Tehran, Iran, 533-539 September. doi: 10.1109/ISTEL.2016.7881879.

[5] Niels L. M. van Adrichem, Christian Doerr, Fernando A. Kuipers 'OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks', Network Operations and Management Symposium (NOMS), May 2014.

[6] http://www.wendangwang.com/doc/b0fd80358fd07ee48d4eb93e/5 (Accessed: 28 April 2017)

[7] Metz, C. (2012) 'Google's Top Fie Data Center Secretes (That are Still Secrete)', Business, Wired, (6:30 AM, 18 October 2012)

[8] Miller, R. (2013) 'Microsoft's $1 Billion Data Center', Data Center Knowledge, (31 January 2013)

[9] Liang, G. and Matta, I. (2001) 'The War Between Mice and Elephants', Network Protocols, 2001. Ninth International Conference, Riverside, CA, USA, USA, 180-188 August. doi: 10.1109/ICNP.2001.992898.

[10] Wei, W. Yi, S. Salamatian, K. and Zhongcheng L. (2016) 'Adaptive Path Isolation for Elephant and Mice Flows by Exploiting Path Diversity in Datacenters', IEEE Transactions on Network and Service Management (Volume: 13, Issue:1, March 2016), IEEE, 12 January 2016, 5-18.

[11] Yan, M. Li, C. Yan, S and Xiaohong, H. (2016) '4 Network Programming 2-20160321', BBU6404: Network Applications. Beijing University of Posts and Telecommunications. Available at: http://www.mayan.cn/IA/#lectures (Accessed: 29 February 2016).

[12] https://www.opennetworking.org/sdn-resources/openflow

[13] https://msdn.microsoft.com/en-us/library/windows/desktop/ms740496(v=vs.85).aspx (Accessed 25 March 2017)

[14] Yue, C. Michael, C. (2015-2016) 'Part 3 Transport Layer', EBU5403: Internet Protocols. Queen Mary University of London. Unpublished.

[15] http://zserge.com/jsmn.html

[16] https://github.com/zserge/jsmn/blob/master/example/simple.c

[RN_3384 Extending the TCP Socket Interface for the Data Centre]

[17] Stoica, I. Pothen, A. (2002) ‘A Robust and Flexible Microeconomic Scheduler for Parallel Computers’, High Performance Computing, 1996. Proceedings. 3rd International Conference, Trivandrum, India, India, 406-412, doi: 10.1109/HIPC. 1996.565855.

[18] TSE, Ng. Stoica, I. Zhang H. (2002) ‘Packet Fair Queuing Algorithms for Wireless Networks with Location-Dependent Errors’, INFOCOM’ 98. Seventeenth Annual Joint Conference of the IEEE Computer and Communication Societies, San Francisco, CA, USA, USA, 1103-1111 vol.3, doi: 10.1109/INFCOM.1998.662920.

[19] Nannan, W. Jason, P. J. Ruijie, Z. (2017) ‘Survivable Bulk Data-Flow Transfer Strategies in Elastic Optical Inter-Datacenter Networks’, Global Communications Conference (GLOBECOM), 2016 IEEE, Washington, DC, USA, doi: 10.1109/GLOCOM.2016.7842309

Acknowledgement

Firstly, I appreciate my supervisor for his precise academic attitude and careful instruction. He gives us chances to study our projects as much as we can and guide us patiently, both academically and linguistically. I am inspired constantly by him.

Secondly, I want to thank friends who have the same supervisor with me. Although we may study different topics, we give supports and advices to each other timely as much as possible.

Thanks to all of you.

Appendix

A. struct sockaddr

```
struct sockaddr {
        unsigned short sa_family; // address family, AF_xxx
        char sa_data[14]; // 14 bytes of protocol address
        };
```

B. struct sockaddr_in

```
struct sockaddr_in {
        short     sin_family; //address family
        u_short sin_port;//port number
        struct   in_addr sin_addr;//IP address in network byte order
        char      sin_zero[8];//no practical meaning
}; [13]
```

C. Function **connect()** Prototype

connect

Defined in header <sys/socket.h>

int connect (int sock, const struct sockaddr * DesAddr, int DesAddrLen)

Builds up a connection to a specific socket.

Parameters:

sock – the socket descriptor

DesAddr – the pointer of the struct sockaddr of the destination

DesAddrLen – the length of the struct sockaddr of the destination

Return Value

Return 0 if success

Return SOCKET_ERROR if fails

D. Function **accept()** Prototype

accept

Defined in header <sys/socket.h>

int accept (int sock, const struct sockaddr * LocalAddr, int LocalAddrLen)

Accepts a coming connection; Creates a new socket used for the communication based on this connection.

Parameters:

sock – the socket descriptor

LocalAddr – the pointer of the struct sockaddr of the local address

LocalAddrLen – the length of the struct sockaddr of the local address

Return Value

Return a new socket file descriptor if success

Return SOCKET_ERROR if fails

E. Outputs of the Manager

Table 2 Outputs of the Manager

Parameters		Recording Period	
		Recorded or Not (in Total)	Recorded or Not (per minute)
Recording Time		-	Recorded
How Many:	Flows	Recorded	Recorded
	Flows from Clients	Recorded	Recorded
	Flows from Servers	Recorded	Recorded
	Unanalysed Flows	Recorded	Recorded
	Flows Bigger than 5 GB	Recorded	Recorded
	Loads of Servers Smaller than 100 Mb	Recorded	Recorded
	Flows of Clients Faster than 5 Gbps	Recorded	Recorded
	Urgent Flows	Recorded	Recorded
	Servers Cannot Deal with Urgent Flows	Recorded	Recorded

	Flows without Urgent Levels	Recorded	Recorded
	Bidirectional Communications	Recorded	Not
	Unidirectional Communications	Recorded	Not
	Unmarked Communications	Recorded	Not
	Flows with Unknown Sizes	Recorded	Recorded
What:	Average Speeds in Mbps	Recorded	Recorded
	Average Speeds of Clients in Mbps	Recorded	Recorded
	Requirements of Average Speeds of Servers in Mbps	Recorded	Recorded
	Total Size in MB	Recorded	Recorded

F. Specification

北 京 邮 电 大 学
本科毕业设计（论文）任务书
Project Specification Form

学院 School	International Schoc	专业 Programme	E-Commerce (H6NF)	班级Class	2013215115
学生姓名 Name	LI Yuqi	学号 BUPT student no	2013213384	学号 QM student no	130806550
设计（论文）编号 Project No.	RN_3384				
设计（论文）题目 Project Title	Extending the TCP socket interface for the data centre				
论文题目（中文）	拓展用于数据中心的TCP Socket接口				
题目分类 Scope	Research	Networks	Software		

主要任务及目标Main tasks and target:	By
Task 1: Review of data centre scheduling literature	31 January 2017
Task 2: Creation of API where clients can communicate with servers to send flow information	15 March 2017
Task 3: Creation of server that can receive information	15 April 2017
Task 4: Creation and testing of working system where TCP sockets are annotated with extra information	01 May 2017

Measurable outcomes
1) Client software that can communicate information to the server
2) Server software that can receive the communicated information
3) Working system that can open sockets after communicating information about them to the server

主要内容Project description:

In big data systems, for example, Hadoop, flows can be synchronised (several flows to the same server from different clients for example). Flows might have a particular deadline (needs to be completed by a particular time). Flows might know how much information they have to send.
Data centres can make use of this information. [1] In this project you will look at a way to extend the TCP socket API so that a client requesting a socket can add information. So, for example, your socket call instead of just saying "Open a socket to address:port" can also say "It will send 15GB and must complete by 15:03". This involves writing a wrapper for the TCP client socket open and a server that can communicate to receive information about the new socket.
[1] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with Varys. In Proceedings of the 2014 ACM conference on SIGCOMM (SIGCOMM '14). ACM, New York, NY, USA, 443-454.

Project outline

This project aims to create extending TCP socket interface used by data centre under Linux. Client sockets send messages with some detailed information such as sizes of packages, speeds of sent data and whether packages are urgent, etc. Server sockets receive packages,store those detailed information and return values if necessary. During this process, usual socket functions also will be available.

Users operate this software under Linux by running C files and typing several commands. Client software and Server software are different and they will realize different functions. Client software will generate extra helpful data. Server software will be called to receive and store those data. To coordinate these two software, a socket library which contains extending socket functions will be established and used.

To achieve this goal, there are some necessary knowledge and tools. C language will be suitable(Under C Free 5.0). Basic functions and those based on TCP transmission will be used frequently. Network Programming, especially Socket Programming is vital(In this project, Oracle VM VirtualBox Ubuntu 12.04 will be used).

To test this system, client code will be used. The code will call socket interface and check whether the server operates as designed. Stored information will be checked, too. Robustness will be required as much as possible.

Relevant literature will be of great significance. To get background materials, both paper books and the Internet will be used.

1. Mossharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with Varys. In Proceedings of the 2014 ACM conference on SIGCOMM(SIGCOMM'14). ACM, New York, NY, USA, 443-454.
2. Gary R. Wright, W.Richard Stevens, Addison-Wesley Professional; 1 edition (February 10, 1995), TCP/IP Illustrated Volume 2: The Implementation.
3. Behrouz A. Forouzan, McGraw-Hill Education; 4, TCP/IP Protocol Suite.

What I expect to have working at the mid-term oral

All literature review about data centre, TCP and socket programming has been completed.

Fundamental clients API used for users has been established. The Protocol used for data communication between client and server has been designed and realized, although bugs may exist.

The server used for receiving data is under preparation.

Overall working system construction is under preparation and simple testing for the system is established.

Fill in the sub-tasks and select the cells to show the extent of each task

	Nov		Dec		Jan		Feb		Mar		Apr		May	
Task 1: Review of data centre scheduling literature														
Data Centre Review		▨	▨	▨										
TCP Review		▨	▨	▨										
Socket Programming Review			▨	▨	▨									
Complete Literature Review			▨	▨	▨	▨								
Task 2: Creation of API where clients can communicate with servers to send flow information														
Pseudo Code Design				▨	▨	▨								
Fundamental API Establishment					▨	▨	▨	▨						
Writing Code that Sends Clients' Data					▨	▨	▨	▨	▨					
Completion of the API that Communicates about the Data							▨	▨	▨	▨	▨			
Task 3: Creation of server that can receive information														
Protocol(used for the API) Design							▨	▨	▨	▨				
Pseudo Code Design							▨	▨	▨	▨	▨			
Writing Code that Receives Clients' Data								▨	▨	▨	▨			
Completion of Working Server that Understands Protocol from Client									▨	▨	▨	▨		
Task 4: Creation and testing of working system where TCP sockets are annotated with extra information														
Testing of Protocol and API between Client and Server									▨	▨	▨			
Testing System with Many Clients in the Same System all Communicating with the Server										▨	▨	▨		
Testing and Debugging of Complete Working System										▨	▨	▨		
Create Output from Server which Summarises the Data Sent to it by Clients										▨	▨	▨	▨	

[RN_3384 Extending the TCP Socket Interface for the Data Centre]

G. Early-term Progress Report

北 京 邮 电 大 学

BBC6521 Project 毕业设计 2016/17

Early-term Progress Report

初期进度报告

学院 School	International School	专业 Programme	E-Commerce	班级 Class	2013215115
学生姓名 Student Name	LI Yuqi	BUPT 学号 BUPT Student No.	2013213384	QM 学号 QM Student No.	130806550
设计（论文）编号 Project No.	RN_3384	电子邮件 Email	2013213384@bupt.edu.cn		
设计（论文）题目 Project Title	Extending the TCP socket interface for the data centre				

已完成工作：

Finished Work：

Firstly, I have read some materials. In *Efficient coflow scheduling with Varys*, the authors introduce *Varys*, a new algorithms used in data-parallel communication to make coflow scheduling meet communication requirements. One of the authors of that, Ion Stoica, is the proposer of coflow. He also publishes several papers to describe coflow, which is used in data centre and scheduling to meet communication requirements. Besides, according to parts related to TCP in *TCP/IP Suite* by Behrouz A. Forouzan, the author describes TCP mechanism, TCP characters and its software package in detail. He also compares TCP with several protocols such as UDP to emphasize features of TCP. In *TCP/IP Illustrated Volume 2: The Implementation*, Gary R. Weight and W. Richard Stevens describe socket programming in TCP and use several codes with C to show practical examples. To enhance knowledge, I also finished reading several published papers about data-parallel, data centre, TCP socket programming and protocols.

Secondly, I have written simple codes about TCP socket programming. Initially, C Free under Windows was used to write code and Ubuntu was only responsible to run those. To improve efficiency and avoid some unnecessary problems, codes are written and run under Ubuntu directly now. To help myself write code and debug easily, I also learned errno and netcat to test my C files. C Free is only used for some simple debugs. Simple C files, such as creating a client and a server to echo .txt files succeeded. Simple threads, which could print words and numbers with C, could run under Ubuntu.

Thirdly, I have designed the pseudo code of client API. To explain the communication process clearly, I have drawn some simple diagrams. To create client API, several socket network programming functions with C need to be re-written to meet requirements. New parameters sent by clients, such as the size of data, will be added by C structure. Moreover, there is a manger controlling the communication between clients and servers of data centre. Clients should contact to the manager first then communicate with servers.

是否符合进度？On schedule as per GANTT chart?	[YES/NO]YES

下一步：

Next steps:

Firstly, I will finish the fundamental Client API establishment. Codes that send data of clients should be under construction then. Secondly, I will design the protocol used for the API. To meet requirements of data centre, this protocol should balance efficiency and flexibility of communication. Thirdly, the manager, managing the communication process between clients and servers, should be designed. Fourthly, I will design pseudo code of server and implement that step by step.

H. Mid-term Check Report

北 京 邮 电 大 学

BBC6521 Project 毕业设计 2016/17

Mid-term Progress Report

中期进展情况报告

学院 School	International School	专业 Programme	E-Commerce	班级 Class	2013215115
学生姓名 Student Name	LI Yuqi	BUPT 学号 BUPT Student No.	2013213384	QM 学号 QM Student No.	130806550
设计（论文）编号 Project No.	RN_3384	电子邮件 Email	2013213384@bupt.edu.cn		
设计（论文）题目 Project Title	Extending the TCP socket interface for the data centre				

毕业设计（论文）进展情况，字数一般不少于 1000 字
The progress on the project. Total number of words is no less than 1000

目标任务: Targets set at project initiation:

(must be the same as “What I expect to have working at the mid-term oral” in the Spec)

All literature review about data centre, TCP and socket programming has been completed.

Fundamental clients’ API used for users has been established. The Protocol used for data communication between client and server has been designed and realized, although bugs may exist.

The server used for receiving data is under preparation.

Overall working system construction is under preparation and simple testing for the system is established.

是否完成目标 Targets met?	[YES/NO] YES

目前已完成任务 Finished Work:

Firstly, I have a primary design of the whole working system. There are three roles in the system: Clients, a manager, servers. A client does not know the existence of the manager. Thus, collection of the file information and sending it to the manager will be done by the library code. The client only needs to call one or more functions in the library I wrote and sends the file to a server. The manager will receive the information and carry out corresponding steps (maybe this function will not be implemented because time limit). The server will receive the data from the client as normal.

Secondly, I have finished the fundamental Client API. To finish this task, I must know the theory about how the C library works. Thus, I learned how to write and use simple library code under Linux. Now I can write simple C files with corresponding C header files and use them in other C files. After that, I discussed with my supervisor about the design of the API many times. Now the fundamental Client API is finished and it has six functions, although details may need to be modified constantly. The API describes that the library can collect information about the size, the transmission speed, the arrival deadline of a file; it also supplies a function for users to set an urgent level of their files; it will collect all above data and store them in a C structure, which will be parsed to a JSON string to send to the manager.

Thirdly, I have designed a protocol used for communication between clients and the manager. At first, JSON, TCP, HTML and XML were considered. After discussing with my supervisor, we thought HTML and XML were not suitable for this project. According to Mosharaf Chowdhury, Yuan Zhong, Ion Stoica. 2014. *Efficient Coflow Scheduling with Varys.* In Proceedings of the 2014 ACM conference on SIGCOMM, (SIGCOMM ’ 14). ACM. New York. NY. USA. 443-454., normally the data transmission is stable. However, when a large flow occurs, this flow will slow down many small flows and even cause missed deadlines. Thus, the protocol should be as much as flexible to carry useful information about the flow. Only then can it have a good predictability to avoid the “larger flows” problem. Comparing to traditional TCP headers, JSON meets the requirement more. JSON is one of the widely-used forms in the world wide now. It is easy to be generated and easy to understand. Other clients, not like clients in my project, could send data information to the manager with JSON, too. Thus, I used JSON to design the protocol.

Fourthly, I have a very simple way to test the API and the protocol. Although the fundamental library code has been established, bugs still exist. Thus, to check the outputs, I separated the whole API functions into several parts in different C files and run those files many times with correct or wrong inputs. All files could be compiled and run correctly. I am also adding some fault-tolerant statements into the library. The whole system testing is not ready yet. In terms of the testing of the protocol, I have a C file called simple.c from https://github.com/zserge/jsmn. This website is an extension of the introduction of JSON: http://www.json.org/index.html. It introduced a JSON parser in C, called JSMN. There are corresponding useful library files called jsmn.c and jsmn.h. Users can use JSMN to make JSON strings available in C. By one example code on the Internet, the simple.c, I can test my protocol by making my JSON string be printed in C.

Fifthly, I have designed the fundamental functions of the manager. The manager plays a significant role in my project. It is the hardest part to design and implement. I have a very simple draft of the manager now. A client can only send data to a server. The manager receives a JSON string from the client, parses it into proper form and stores it.

Sixthly, I have designed a general draft of the server. Roughly, it is the same as normal and simple servers under TCP.

尚需完成的任务 Work to do:

Firstly, I should have a client which can send information of a file to the manager and send the file to the server.

Then, I will complete the API which communicates about the data.

Thirdly, I should design the pseudo code of my server which will later be gradually improved. In the meantime the implementation of the server will get underway.

Fourthly, I will test the working system in different ways. I will complete the testing about the API and the protocol mostly. To make the project more practical, I should also test the manager and the server with several clients running. The whole system should also be tested. Besides, I should ensure the outputs of the manager are correct and useful.

Finally, I should prepare my draft report.

能否按期完成设计（论文） Can finish the project on time or not:	[YES/NO] YES

存在问题 Problems:

1. According to my *Literature Review*, I am not equipped with enough knowledge about this project.

2. The information collected by the manager sometimes may be very important and confidential. Thus, some simple authentication should be designed.

3. The knowledge about JSMN is not specialty enough.

4. Usually some detailed designs are not practical.

拟采取的办法 Solutions:

1. Continuing to read papers about my project.
2. Trying to find some easy authentication ways. At least, a simple log-in function should be designed.
3. Continuing to read materials and do practices about JSMN.
4. Having a good time management so I can implement my designs and find problems early. I can have time re-design or modify them.

最终论文结构 Structure of the final report:

日期 Date:	8 March 2017

Risk Assessment

This project is about a pure research and C programming work. Thus, there are no physical risks in it. The main risks are about time management.

Table 3 Risk Assessment

Description of Risk	Description of Impact	Likelihood rating	Impact rating (Consequence level)	Preventative actions
Work Behind Schedule	Cannot finish in time, influence next step	Moderate	Very Serious	Make preparations for next stage as early as possible
Bugs cannot be fixed soon	Schedule delay	Likely	Serious	Use online materials to seek solutions and prepare another way to achieve the goal
System or software crash down	Miss part of work or lose all work	Rare	Catastrophic	Backup timely

Environmental Impact Assessment

This project is totally based on researching and computer programming. Thus, there is no environmental impact assessment.