Network Capacity Planning with Network Application Simulator

H. SUFEHMI

UNIVERSITY OF CENTRAL ENGLAND

MSc

January 2000

Network Capacity Planning with Network Application Simulator

This is a complete study of a Microsoft Windows-based network software project, from design, implementation, and quality assurance phase.

by

HARRY SUFEHMI

A thesis submitted in partial fulfillment of the requirements
of the University of England for the degree of
Master of Science

January 2000

University of Central England in Birmingham
in collaboration with
Central IT department of Birmingham City Council

I would like to dedicate this work to my God,

who have been very kind to this insignificant person.

Indeed He put me through the tests, that will bring down most people I know,

but He always provides me with the motivation,

and in the end reveals a (previously) hidden way out,

that brings me through it

Also to my parents,

whose love I just realised in the recent years,

masked behind their harsh discipline to me

Acknowledgements

During the length of this project, so many people helped me, it will be almost impossible to list them all. But several simply stand out from the rest.

First and foremost I would like to thank Jim Martin, for believing in us and fully supporting us all the way until the end. He treats us as an equal, and is a very fair person.

Then I'd like to thank Dr. Michael Boyd, for assisting me through the whole course. I faced a new education system and not familiar with it, but he helped me happily everytime I needed it, even though he's a very busy man. He's also a very fair person, which I really appreciate.

I'd like to extend my thanks to the people in Student Services department, especially to Sacha and Liam. I'm a stranger in a new country with very different system, and faced a lot of problem in the beginning of my stay here. They helped me sincerely every single time I came to them, so those problems are my past now thanks to them.

This list would not be complete without thanking some other people who enabled me to realise this project with their technical advice and/or assistance. Francois Piette with his ICS (Internet Communication Suite) enabled us to build solid network code easily and quickly - and he provided the software for free including the source code. I learned a lot just by reading his code. Stephen 'Sly' Williams created MiniChat application, which we used to built the skeleton of our basecode. He also put the software for free on Internet including the source code. Therefore we can built reliable client/server code very quickly. Many thanks also to the people on TWSocket mailing list, for helping us to solve our network and Delphi programming issues - especially to Wilfried. A sincere thank you goes as well to the members of Delphi programming forum (delphi-programming@egroups.com), especially to Paul, for their assistance.
Without them, this project will still in the state of limbo, but thanks to them we got it in its current state now.

Last but not least, word of thanks are reserved to my family and my colleagues in Birmingham City Council. Thanks to my family for supporting me through the hard times, and for understanding when I can't spend my time with them. Thanks also goes to Peter Everitt (especially for letting me to use the Testlab facilities), Simon Bentley, Jaswant Malhi, Maurice Smith, Timothy Charge for being my friend and supportive colleagues - especially to Steve Hewkin for approving my project in the first place and support me all the way through it.

A special thanks goes to Ian Paterson - thanks for believing in me.

Network Capacity Planning

with

Network Application Simulator

"*...When I was a graduate student in the mid-1950s, I could read **all** the journals and conference proceedings; I could stay current in **all** the discipline. Today my intellectual life has seen me regretfully kissing subdiscipline interests goodbye one by one, as my portfolio has continuously overflowed beyond mastery. Too many interests, too many exciting opportunities for learning, research, and thought. What a marvelous predicament! Not only is the end not in sight, the pace is not slackening. We have many future joys.*"

"*The tar pit of software engineering will continue to be sticky for a long time to come. One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate of man's handiworks. This complex craft will demand our continual development of the discipline, our learning to compose in larger units, our best use of new tools, our best adaptation of proven engineering management methods, liberal application of common sense, and a God-given humility to recognize our fallibility and limitations.*"

Frederick P. Brooks, Jr,

The Mythical Man-Month, 1975 -1995

Table of Contents

EXECUTIVE SUMMARY

This thesis discuss a software project, which aims to simulate an application's behaviour over network. This, in turn, will enable the Network Administrator to gain the data needed to do a proper network capacity planning.

The dissertation discusses the complete project life cycle. Starting from system design & analysis, implementation, and testing cycle. Technical issues, especially on network programming, noted and solved wherever possible. Also discussing the management aspect of the project; some interesting lessons learned from this project, mostly noted in the conclusions chapter.

1.0 INTRODUCTION

Network Capacity Planning is an aspect of a networked computer system that many IT manager/Network manager forgot to do. Especially for IT managers in a growing company, there are many cases where they got the shocking surprise when their current network infrastructure (seems) suddenly just overwhelmed by the load imposed by its users.

In a WAN (Wide Area Network) environment especially, if a network application will be deployed, it is advised that the company first assess its possible impact on the WAN resources. Because most WAN links has only small bandwidth, even an application that runs fine in a 10 Mbps is still possible to bog down the WAN links when deployed without tested first, making it unusable.

NUS (Network Utility Suite) is a software suite which contains 3 modules, based on client/server architecture. It contains Network Benchmarking, Network Monitoring, and Network Application Simulator modules. This dissertation concerns itself only with the last module, Network Application Simulator, in relation to the interest to network capacity planning.

Each of these modules are done by different people, and presented as a separate project. In the end however, it is intended to merge all of those modules to form a single software suite, called NUS.

NUS's Network Application Simulator is designed to be able to perform robust simulation on the real network itself, if desired, outside office hours for example. It will also be able to perform it in an isolated environment, in a Testlab for example. The following lists its features:

- Ability to perform simulation on the real network, by simulating the specific application tasks over network between the server and client.
- Ability to test the whole subnet of the WAN (more realistic result, but must be done off-hours to avoid impact to working users), or do the test just in an isolated network environment (less realistic, but can be done anytime) between as little as 2 computers.
- Produce statistical report to the user
- Easy-to-use user interface

The ability to simulate an application on the network could be realised with the aid of

a spesialised scripting language (NetSim) that will simulate the behaviour of that particular application. So basically we will have a mini interpreter in this software to interpret the script, translate it into a form understandable by the computer, and run it. Before we can create the script, we need to capture the application's traffic first by using any packet capture software; such as LANalyzer, Microsoft's SMS, or some other tools available in UNIX. Then based on the captured information, we can create the script.
The script is very flexible and can be tailored to simulate practically any network traffic scenario. The options menu will provide only basic configurability, for maximum configurability the user will need to tune the script. But since the structure and the syntax of the scripting language is very simple, it should not be a problem.

The application also has a reporting feature. It will report the bandwidth requirement for the duration of the simulation, and will also provide a graphic chart of the simulation process. Therefore user can further analyse the network traffic in much detail, since the graphic chart is quite versatile - we can move/zoom in/zoom out/print the chart easily.

The simulation software produced from this project will be rather basic. One can say that it's a proof-of-concept software; demonstrating the validity of the student's hypothesis on network simulation, although still providing basic functionality. But, inside it already has the foundation for further enhancements, so it will be easy to extend the software or to add more features to it.

The student shall continue to maintain this project. In the event of his inability to continue maintaining the project, the student intend to release the full source code to a trusted third-party or put it on the Internet, whichever in his judgement will provide the best benefit to the community; so to enable further development of the software. MRTG (Multi Router Traffic Grapher software) is one prove of many that by providing your source code to the community, you'll accelerate its development (see http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/mrtg.html#HIST).

All software released from this project will be available for free for use by educational and government institutions. And in addition to it, full source code will be made available to educational institutions on request, to encourage further development and study of this topic. Distribution of the software will be done by utilising the Internet.

2.0 DEFINITION OF THE PROBLEM

Simulate.

With nowadays computer processing power, we can simulate nearly everything. Some people create games (that run on computer) that simulate the behaviour of animal pet, and have it selling like hot cakes. Some simulate the war zone, with tanks, soldiers, helicopters, and weapons that acts like the real thing. Some people simulate the star constellations. And some others build massive clusters of super servers, to run atomic bomb simulation. Etceteras.

Now, we have got a problem in our hand.

We need to deploy a new application software on our WAN. But first, we need to know, how will it impact our WAN? Will our current infrastructure handle the additional load gracefully? Or, will it be torn apart under the stress of the imposed load?

Another problem, we need to find this out **before** we deploy the software. Otherwise, it will be too late for us. How can we do this, since we need to deploy the software to find out its network traffic requirements?

This is a classic chicken-and-egg problem, in which simulation software fits the bill perfectly.

Imagine the following scenario, say a company, XYZ Inc, will deploy a new SAP-based GL (General Ledger) application. SAP being designed as a good Client/Server application uses bandwidth very efficiently. So the Head of IT decided to deploy it right away, because he doesn't have the resources (or so he thinks) to do an assessment study regarding its deployment impact to the XYZ's WAN - therefore he just assumed that SAP will use very little WAN bandwidth.

It turned out that their SAP-based application has been modified slightly by the development team, and now it requires more bandwidth than ever. Coupled by the fact that there are 4500 users that use the application, soon the WAN links of XYZ's Inc are experiencing very bad network traffic jam, especially the offices with only 64K connection.

In the morning, when all of its 4500 users are trying to logon to the server simultaneously, it will also cause major performance slowdowns. And also after lunchtime, when everyone comes back from his or her break and start logging in again to the system.

The server handles it fine as planned, but not the WAN links in between.

But their nightmare still hasn't ended yet, because they still have another HRIS (Human Resources Information System) that nears completion, that of course will requires another portion of network bandwidth. Now they have to re-engineer the GL system to use less network bandwidth, upgrade the WAN links, and do a study on HRIS application to assess its impact to the WAN - all in the same time.

If only they had done the GL's assessment study since its early development, they'll have plenty of time to address those issues far in advance.

Therefore, a mechanism to easily calculate/simulate future network capacity requirements is needed. Which is what the simulation software will do.

We can simulate the application software multiplied by the number of users using it quite easily by using 2 or more computers, linked in a network that should be isolated from the WAN.

Simulation software also provides us with the following benefits, compared to deploying the software first and measuring it later:

- It is less costly.
- Results can be obtained much quicker.
- The disruption of day-to-day operations on the user, which often accompanies a field research, is completely avoided.
- Different schemes can require significant changes to the workstation configuration (software upgrade, etceteras), which are not acceptable for research purposes.

Now we know what our problem is, and what kind of software that can help us to solve it, we'll look into the possible solutions available for it.

3.0 REVIEW OF POSSIBLE SOLUTIONS

Currently, there is some network simulation software available on the market. But basically, after looking and analysing their specifications, all of them can be categorised within the following three types of network simulation:

1. Simulate the whole network on computer

This software package will simulate the whole network - routers, links, bridges, and etceteras - on a computer. Nothing goes in/out to/from it, all simulation operations are done within that single computer. This software type is most suited to do the design of a network, because it can only do global simulation of the network, not an application-based one. A sample of this software is ComNet III, product of CACI Products Company.

2. Simulate the traffic on network

This type of software will simulate network traffic over the actual network. It will recreate an application software's network traffic and then measure it. The simulation can be run over an isolated LAN, so it will not disturb the users. A sample of this type of software is Chariot, product of Ganymede Software Inc.

3. Simulate the actual software behaviour

This is the one that will give the most realistic feedback of three. This software will not only simulate the network traffic, but will also put the server under the stress like it will experience with the actual application software simulated. This is because it will mimic the user's interaction with the software and it will connect to the actual server itself.

Therefore, this kind of software not only can be used to do network capacity planning, but also can be used to estimate server requirements as well. However, this means very closely mimicking each software's behaviour, and it's not an easy thing to do - in fact, it's close to impossible. We can do it, but for different software we will need to develop different simulator. Not very feasible to become the basis of a software project indeed.

But now thanks to the proliferation of thin-client technology, which utilises browser as the client software, finally we got a common platform to simulate. It's easier to simulate software’s behaviour on a browser, although it's still not very easy to do because the browser itself can have many ways to interface to the server.

A sample of this type of software is LoadRunner 6, product of Mercury Interactive Corporation.

Of those three types of network simulation tools, what we need to solve our problem is the

type number 2, simulate the traffic over network.

So, there are some products of this kind actually available by the time of the writing of this dissertation. So why redevelop again? Why should we reinvent the wheel?

First, those software come with no source code. It means that the users are fully dependent on the vendor - which feature will be included, which software bug will be fixed, which support queries will be answered, etceteras. For corporate users (big companies, etceteras), most of them will feel fine with this arrangement, since they won't have time and resources anyway to tinker with it. But for other users, such as educational institutions, they can make use of the source code if it is available. This is so when they need the software to do something a just little bit different, instead of requesting the feature to the vendor and (if the vendor decides to implement it) waiting for it to be implemented (with no finished time guarantee from the vendor), they can do it themselves straight away.
Also, availability of source code is good because then it can be used as a case study / study tool for the students. For advanced students, simply nothing beats the actual source code for learning computer programming on certain topics.

Second, most of those software packages are very expensively priced. For example, one of them is priced at 10000 GBP for only 10 workstations simulation. If you want to simulate more workstations, then you'll need to buy additional licensees. Unfortunately, most educational and government institutions are already in a very tight budget, those prices are simply out of their reach. But because WAN implementation are so widespread nowadays with infrastructure costs continuing to drop, they have to have it, otherwise they'll have problems every time they deploy something new on their WAN.
By developing this software and providing it for free to them, they won't have to waste the taxpayers' money on those expensive software packages.

Now we have a very valid ground for developing the software, we continue to start researching the topic.

4.0 RESEARCH

Network application simulation is a topic that's rather unfamiliar to academic environment. This is because just several years ago, only a handful of big companies could afford WAN - and this topic mainly of interest when applied to WAN. But then suddenly, with Internet as the catalyst, WAN is almost everywhere. Companies like Cisco, that create devices for WAN, reap tremendous benefits from it and nearly in an instant become something from almost nothing. Even commercially there are not many products of this type available. No wonder there are people that's desperate to do it, but can't find the appropriate software for it - there are too few of them and the concept is (not yet) well known & widespread.

Therefore, although we were surprised at first, it's only understandable that we found difficulties in obtaining references in this specific topic. We can find technical information on basic network and windows programming with little problem, but for the topic itself we can't find it anywhere.

Fortunately, Birmingham City Council's Central IT department have one copy of Ganymede's Chariot, a software that specialise in Network application simulation as well. The department bought this software based on recommendation from its consultant. So, we can utilise it as a comparison product for our project.

Previous work experience as System Administrator for two years and self-taught programmer for 5 years proved to be invaluable in this project. We can quickly get the feel of this topic, and devise a basic system design, to be tested first. When this first design prove to be correct, then we continue to progress on the actual software.

We also gather a lot of sample data by utilising add on software, such as Microsoft SMS's (System Management Software) packet capture utility, etceteras. Then the data obtained is further analysed.

The whole analysis result then becomes the basis for the design of NetSim scripting language. Refer to chapter 9.1 for complete information and documentation on NetSim.

We also found no problem on deciding which programming language that will be used. Delphi simply wins hands-down against other alternatives such as Microsoft Visual C++ / Visual Basic, based on the following criteria:

1. Easy to learn

Delphi is basically Object Pascal. Pascal is a language that's very easy to learn, yet maintains

strict good programming practices, that it is usually used to teach programming to students new to computer.

2. RAD (Rapid Application Development)
This is one of the time we really feel like developing in the speed of light, since Delphi lets us to concentrate on the logic and it handles other nitty-gritty details for us. Its IDE (Integrated Development Environment) is also very intuitive & easy to use and it's not getting in our way.
After the establishment of the basecode, we can develop even faster. Simply a breathtaking experience.

3. Optimised result
With Delphi, the default compilation result is a single, small executable file. This makes distribution of our software very easy to do. Maintenance is easy too, since we only have a single file to concern, dramatically reducing version conflict problem possibility.
Visual Basic on the other hand will produce very big executable file by default, with lots of links to other DLLs (Dynamic Load Library). This can make distribution and maintenance to become a major headache. And this has been proved on many occasions.

4. Good performance
Borland has always been associated with performance since its Turbo Pascal v1.0 product, and Delphi Professional v4.0 is no exception. The resulting code is very small and efficient, compared to standard Windows executables. And yet we can still develop in it very quickly. No wonder experts all over the world use Delphi to develop serious applications.

5. Stability
Again, Borland has always been known to produce rock-solid products, and we are very satisfied with its Delphi 4 product. Later we found out that on Delphi community, Delphi 4 is regarded as the most bug-ridden version of Delphi. But even then, throughout the duration of the project we only stumbled on one bug, if we can call that particular problem a "bug". We never experience unexpected crashes, even though by its very nature our software put the system under huge load, even so that we were actually kind of expecting it to crash anytime soon. But it just keep on going on, and on. Stable software is a quality that's unfortunately hard to find in Microsoft products due to the way they develop their software - they care more to the number of features included rather than the software's stability.

It is worth noting that the Internet is the center of our research. We still went to libraries from time to time to obtain research and reference materials, but most of the time we spent our research time on Internet. As stated before, not many papers on network application simulation topic are available, but we found some other invaluable resources. We

can find the further details on network programming there. We can also chat to our friends all over the world, asking for assistance on certain things. However, the method that proves to be the most useful is mailing list.

Mailing list is a bit like newsgroups on Internet, USENET, but with the following advantages:

1. The discussions are delivered to our mailbox, very convenient.
With USENET, we need to have separate newsreader software first, and then it's not always very easy to use. On the other hand, we're all already very comfortable using our mail software.

2. Better controlled, because in a mailing list there is at least one person appointed as the list's moderator. The moderator's role can be of a lot of thing, but basically he ensure that only appropriate messages get posted on the list. With USENET, on contrary, pretty much everybody can post anything, resulting in a lot of "junk" messages.

So tremendous the benefits we realise from utilising Internet's mailing lists, we don't think this project will be in its current state if we didn't utilise it. And generally mailing list has started to become recognised as an important piece of Internet, especially since the emergence of mailing list directory and service provider such as eGroups (http://www.egroups.com) and OneList (http://www.onelist.com). However, some of the best mailing list are the unlisted ones, means you have to look around really hard to find it.

We list below the mailing lists that have assisted us on the development of this project:

1. TWSocket mailing list
(twsocket@rtfm.be, more information from http://www.rtfm.be/fpiette/supportuk.htm)
This mailing list is provided by Francois Piette, the maker of TWSocket component, in order to provide support for that product. But actually it's more of Winsock and Delphi programming mailing list, not very strict on just TWSocket. A lot of very knowledgeable people subscribed in this mailing list as well. They're very courteous and supportive to each other, giving the best example of the Internet sharing spirit. Especially Francois himself, Wilfried and Stephen Williams are among the best of them, although the others are very helpful as well.

2. Delphi programming mailing list
(delphi-programming@egroups.com, more information from http://www.egroups.com)
This is strictly a mailing list for discussions relating to Borland Delphi programming. We've got some help here, especially when we found problem with TOpenDialogBox. At first we thought

we have found a new bug in our copy of Delphi 4. Consultation with members of this mailing list reveals that the problem is actually the expected behaviour because we did things slightly different than what it should be - because of lack of detailed documentation. After correcting our code just a bit, it works smoothly.

eGroups, with features like scheduling, real-time chat, file archive, database, etceteras; is actually more of a teamwork assistance service than just a plain mailing list service provider. We even setup our own mailing list there, named nus_project@egroups.com to assist us to communicate on this project. Also its file archive features means we can store our work on Internet as well and therefore can access it from anywhere, anytime - giving tremendous flexibility and also serve as a backup strategy for our work.

3. PAU-Mikro
(pau-mikro@nusantara.net, subscribe by sending empty mail to pau-mikro-subscribe@nusantara.net)
This mailing list is an Indonesian computer society's mailing list. A lot of helpful and very expert people currently subscribed to this mailing list; with profession ranging from system administrator on some multi national company to high-school student. We got our eyes opened to new perspectives, and have gained a lot of invaluable knowledge just by reading the discussions that's going on there.

Although not as extensively used as the mailing list, newsgroup is still a goldmine of information; if we know the most efficient way to access it. And the way is through the newsgroup search engines.

The oldest newsgroup search engine is DejaNews (http://www.dejanews.com), while Altavista Search follows much later (http://altavista.remarq.com). In the Testlab we can only use Altavista, since access to Dejanews is (unfortunately) blocked by the Birmingham City Council's firewall. Both service archive almost every existing newsgroup's contents, and index them. So by searching in there, we can avoid getting the junk and receive just the ones that's of interest to us. A huge saving on time indeed. Also for technical issues, we found out that it's better to rely on newsgroup search engine rather than web search engine, the later tend to point us to totally unrelated websites.

And last, web search engine is still useful for finding white papers on specific topics. Of all available, we always found Inference Find (http://www.infind.com) to be the best. The way it works is by submitting the search query to 6 major search engines, collect the results, and weed out the doubles. Not only that, it also group it by location. So you'll got much less number of hits (usually about 50 websites) but most of them are the ones related to that you're looking for.

5.0 WORK UNDERTAKEN

5.1 SYSTEM DESIGN

In the beginning, it is our intention to have proper System Design Document.
But then, I remembered a rule that I learned from previous software development experiences:

"*It is impossible to create a fixed System Design document for medium to large software projects*"

If the system designer is experienced enough, he can write a system design document that will be closer to the end product, but still not exactly the same. Change of specification will *always* happen on the duration of the project. In our case, we have no previous network programming experience, and very little Windows programming experience. And with simply no previous system design experience needed for this kind of project, we feel that creating a complete system design document will be a waste of time that *otherwise* could be better utilised for research.

In another word, we fully agree with Microsoft on this case (Nusenoff, 1996, and Cusamano, 1995) :

"*Build functional spec instead of complete product spec*"

Microsoft, being a high profile software company with big software projects, in a very volatile market with development time limited to mostly just 1 year, can't abide to conventional software engineering practices. This is because of the following reasons:

- They need to adapt to new threats, which may appear at anytime. Quick adaptation to new threats could mean major design changes near the deadline. Conventional software engineering practices don't cope with this well, they assume that the system design concepts are fixed (Brooks, 1975). A sample memorable situation of this is when Microsoft almost released Windows95 - suddenly Internet was booming from almost nothing. Microsoft was able to react fast enough and apply last minute changes to the operating system to embrace the Internet. This proved to be very essential - yet it was not in the original Windows95 design specification.
- Time limitations are very constraining. Especially after the Internet revolution, once the software development cycle can run for 24-36 months, now software

developers got only 12-18 months.

- With so many interests influencing the software project, again design changes could happen anytime.

Therefore, they adopted the concept above. By building functional spec, their project manager just creates a product vision statement. They simply see design documents as too constraining. However, it is up to the project manager to ensure the end product's quality and conformance to first product design statement, because there is no strict design documentation to guide the whole developers through the project other than the source code itself. And to further save time, usually the code itself is very light in comments. From the whole source code of Microsoft Excel, for example, the comments are only about 1% of it.

Of course it all comes at a cost. The typical Microsoft software product reaches store shelves with only 75 percent of the features contained in its functional spec's original draft. And for quality assurance, Microsoft adopts a technique called *tester-developer pairs*. What it means is basically for each developer there will be a tester assigned to him/her (Nusenoff, 1996). It is a very expensive overhead indeed, since there are about 1850 developers working for Microsoft. And still afterwards we know that Microsoft's products are typically bug-ridden.

Amazingly enough however, it works. Microsoft is now the number 1 software maker on Earth. They're a big company running huge software projects, yet able to move at incredible speed, thanks to this technique.

In our case, we realised that we have some similarities to Microsoft's case:

- We're severely limited in time. Remember that none of our team member has comprehensive experience about the topic of this project.
- We have radical design changes halfway. This is again because we're inexperienced on this topic, indeed it is very possible that we designed it wrong the first time.
- We're severely limited in human resources. We need to adopt a way that enables us to fully utilise it in a most flexible manner.

Therefore, we adopt an approach that basically very similar to the one adopted by Microsoft.

In the beginning of the project, we held some brainstorming sessions to share our ideas, and agree on a set of important priorities. The details are communicated along the

way. The communication also serves to ensure everyone have the same idea of what they're doing and avoid any misunderstanding for the duration of the project.

Later on, in the middle of the development, I was convinced that we finally got a good system design idea. At that time I'm halfway developing the basecode, and have a much better idea of the future of the software. Therefore, only then I composed the System Design Document, available for reference in chapter 9.2.

We never found this kind of software engineering technique in any textbooks before. However, our resource limitations forced us to be more on the practical side; and the technique proved to be very essential for the completion of our project on time.

5.2 IMPLEMENTATION & PROJECT MANAGEMENT

5.2.1. Incremental Software Development

When we started to learn to do computer programming, we were usually writing software. That's it, we just write it, sometimes with no clear idea of the end result and no strategy at all. Usually it will be finished, if it finished at all, after stopped and restarted for so many times.

Then came the *build* metaphor. Instead of *writing* software, we're now *building* software. So we understand how like other building processes the construction of software is, and we freely use other elements of metaphor, such as *specifications*, *assembly of components*, and *scaffolding* (Brooks, 1975).

However, for a complex software, this metaphor could become a real danger. First, developers could get bored real fast with complex system. Second, it's easy to go through the project for months and still have nothing running, further making everyone feel stressed. Third, with complex systems, it's very easy for one to get lost in all of its details.

A new metaphor then arise.

Quoted from Brooks, 1975:

"*Some years ago, Harlan Mills proposed that any software system should be grown by incremental development. That is, the system should first be made to run, even though it*

does nothing useful except call the proper set of dummy subprograms.
Then, bit by bit is fleshed out, with the subprograms in turn being developed into actions, or calls to empty stubs in the level below."

"*I have seen the most dramatic results since I began urging this technique... The approach necessitates top-down design. It allows easy backtracking. It lends itself to early prototypes. Each added function and new provision for more complex data or circumstances grows organically out of what is already there.*"

"*The morale effects are startling. Enthusiasm jumps when there is a running system, even a simple one. Efforts redouble when the first picture from a new graphics software system appears on the screen, even if it is only a rectangle. One always has, at every stage in the process, a working system. I find that teams can grow much more complex entities in four months than they can build.*"

How simple and common sense it is, yet nobody realised it before.

We experienced exactly the same phenomenon when we applied the technique. Everyone cheered when the first build of the software ran. It still got horrible bugs in it, but nobody cared. It drove up everyone's morale, and got everyone up to speed.

For the length of the rest of the project, we continue to apply it.

5.2.2. Plan to throw one away

Building simple software is easy, sometimes you don't need any preparation at all and just start writing it. But for complicated systems, even the most experienced system designer can have too many design errors of it. And then, the developers can have errors in implementing it. The fact is, this is because software projects are among the most complex projects that we, humans, ever undertake.

Therefore, Brooks (1975) proposed that in the beginning, the system designer should plan to throw a full system away, and then start rebuilding the software from scratch again. It's such an extreme suggestion, that it's only normal should one instantly questioned it. But, there is some basis for the suggestion:

- The first system built will (almost) always be barely usable, you really don't want to use it. It will be either too big, too slow, too bug-ridden, or all three of them. And changing it may require far more work instead of rebuilding it again.

- Since no software project is alike, the first system build really is a learning experience for the programmer, so you can't expect high quality end product from it.
- "*The only constant is change itself*". By the time the first system is built, the whole team would have learned a lot of things that in many cases it requires immediate change in the design.
- All big software projects has experienced this (Brooks, 1975).

So we really should have planned to throw one software build away, we will anyhow. If we did plan it, then we can take it into account, and will be able to create a more accurate project schedule.

Unfortunately, we didn't.

We got an unpleasant surprise halfway when we realised that our current work won't be very extensible, and won't scale very well in the future. This event is explained in more detail in chapter 7.1.1. In the end, we have to scrap our current work, and rebuild it again from scratch - resulting in the development of the NUS basecode. Everyone's morale was decreased. If we have planned it in advance and include it in our project schedule, then we could squeeze extra efforts to meet the deadline well in advance. The extra time gained then can be utilised to give extra time for writing the dissertation and to establish a rigorous quality assurance process; and we'll also avoid the unpleasant surprise.

5.2.3. Project's Timetable

Below is the original project timetable that we wrote in the beginning of this project.

General Project Plan

July:
- Setup mini-lab
- Create design document
- Research about Winsock2 / interfacing to TCP/IP from Windows environment
- Delphi 4 training → coding environment, language syntax, debugging technique, advanced programming, etc
- Create a sample TCP/IP-based client/server application, to become the base of our development

August:
- Coding → User interface, sample of reports
- Attempt to further fluent ourselves in Delphi 4 and Winsock API programming
- Utilising the mini lab to test early builds of the software

September:
- Coding → Main network code
- Crosschecks – evaluation to work so far
- Start quality checks
- Start utilising Birmingham City Council's Testlab facility to test the software. All testing are scheduled to be done in Friday, after office hours to avoid overloading the network should something goes wrong.

October:
- Coding → Finishing up code, attempt to include client's auto-upgrade capability
- Crosschecks – evaluation to work so far
- Quality checks
- Testlab use scheduled every Friday, after office hours
- Optimising network code to get better performance

November:
- Transferring debugging log from source-code/logbook to the thesis document
- Documenting software
- Final quality checks
- Testlab use scheduled every Friday, after office hours

December:
- Consolidation
- Attempt to unify all coding work, to become a single software suite
- Integration testing
- Porting client software to other platforms (Solaris & Linux)

15 December 1999 – Finished all work

5.2.4. Technical notes

Throughout the projects, we have learned so many things. We lists some of them below:

5.2.4.1 Winsock (Windows Sockets)

Quoted directly from Quinn and Shute (1996):

Winsock Overview

In the beginning of 90's it was becoming apparent that Microsoft Windows was going to play a definitive role in the future of desktop computing. At the same time, TCP/IP was going to have major impact on the world. At the intersection of Microsoft Windows and TCP/IP, however, lay a problem. All the vendors of TCP/IP products for PCs had produced different programming interfaces for their own products. There must be a better way.

That "better way" got started at a BOF ("Birds of a Feather") session at Interop in October 1991. In a room containing 30 or so of the brightest minds in the industry, the idea of creating a single, standard transport interface took shape. The key criteria were to keep it as close to the existing Berkeley Sockets API as possible and not to require a shoe horn to use the interface in a Windows message-based application. It's often tempting for technical people to want to do a perfect job and include everything in a technical design. One key decision made by the group was to limit what included in the interface to what the majority of programmers needed. Combined with the energy and willingness of competitive vendors to work together, this laid the foundation for the creation and success of the Windows Sockets API, or "WinSock" as it has become known.

Over the next 15 months, many people on the WinSock mailing list contributed to the development of the preliminary specification. Several interoperability sessions (WinSockathons) took place, and the result of this co-operative effort was the Windows Sockets version 1.1 API specification.

WinSock lies right at the heart of the explosion of interest in developing and using communications applications on Microsoft Windows PCs. The global community of developers who co-operated to define this programming standard is being joined by other programmers from all over the world who want to add communications capabilities to their software applications by using WinSock.

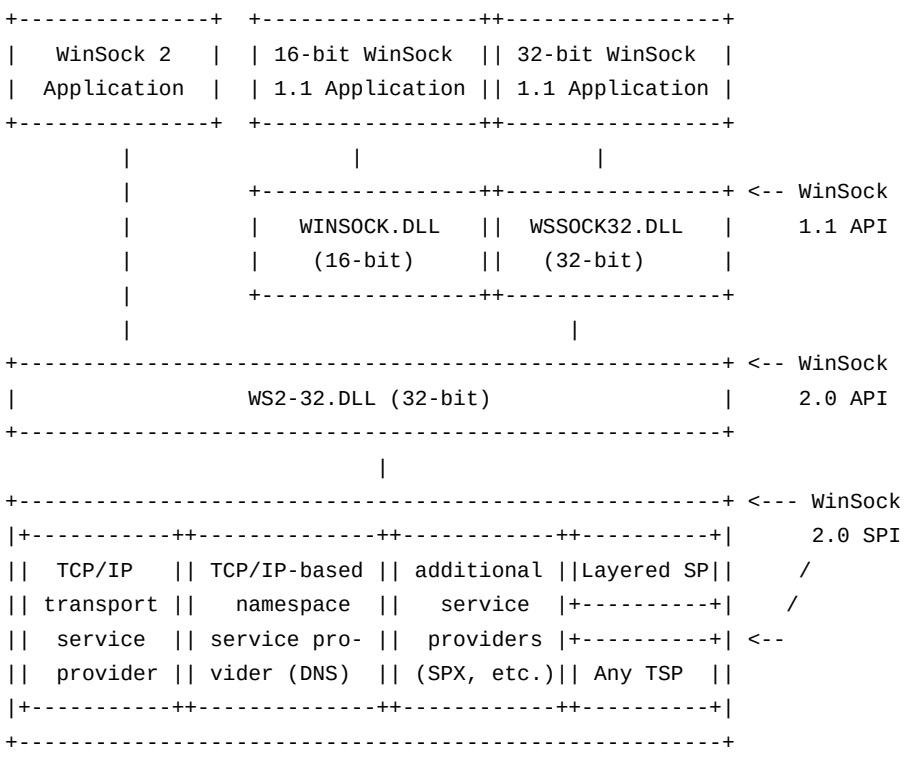
WinSock Architecture

The authors of the original WinSock (version 1.1) deliberately limited its scope in the name of expediency. One result of this is the simple architecture of WinSock 1.1. A single WINSOCK.DLL (or WSOCK32.DLL) provides the WinSock API, and this DLL "talks" to the underlying protocol stack via a proprietary programming interface. This works fairly well since v1.1 WinSock only supports one protocol suite--TCP/IP--and most computers running Windows have only a single network interface.

However, this old WinSock architecture limits a system to only one WinSock DLL active in the system path at a time. As a result, it is not easy to have more than one WinSock implementation on a machine at one time. There are legitimate reasons to want multiple WinSock implementations. For example, one might want a protocol stack from one vendor over the Ethernet connection and a different vendor's stack over the Serial Line.

The new WinSock 2 has an all-new architecture that provides much more flexibility. This architecture allows for simultaneous support of multiple protocol stacks, interfaces, and service providers. There is still one DLL on top, but there is another layer below, and a standard service provider interface, both of which add flexibility.

WinSock now adopts the Windows Open Systems Architecture (WOSA) model, which separates the API from the protocol service provider. In this model the WinSock DLL provides the standard API, and each vendor installs its own service provider layer underneath. The API layer "talks" to a service provider via a standardised Service Provider Interface (SPI), and it is capable of multiplexing between multiple service providers simultaneously. The following sketch illustrates the WinSock architecture.



The WinSock specification has two distinct parts: the API for application developers, and the SPI for protocol stack and namespace service providers. Of course I will be concerned only to the API part, since the API can provide transparency over the lower parts.

Notice also that the intermediate DLL layers are independent of both the application developers and service providers. These DLLs are provided and maintained by Microsoft and Intel. And lastly, notice that the Layered Service Providers would appear in this illustration one or more boxes on top of a transport service provider.

5.2.4.2 Event-driven versus Multithreading programming technique

Quoted directly from Piette (1997):

Event-driven

Event-driven is the natural way of doing things with Windows operating system. When you drop a component on a form, you can see in the object inspector that each component has a lot of events. Most well known is OnClick event for many visual components such as TButton.

Everybody knows that OnClick event is triggered when user clicks on the visual component and everybody knows that he has to supply code in corresponding event handler. This code will be executed when user clicks on the component.

An event is simply something that occurs in the system, such as the user clicking on a button. Speaking Delphi, it translates to code that is executed when that thing occurs: user clicks on a button and code you placed in OnClick handler is executed.

Multithreading

We have seen that event driven with asynchronous component will multitask nicely. It is called co-operative multitasking. But what about if your processing need a lengthy blocking operation such as executing a SQL request? Simple: use pre-emptive multitasking. Under Windows operating system, this is called multithreading.

A Windows program can have several threads. At start-up, each main thread is executing. It can launch other threads. Each thread executes for a short period of time (a few milliseconds), then next thread executes and so on. At human scale, all threads seems running simultaneously. If a thread execute a lengthy operation, it is simply pre-empted after his time slice has been exhausted. His operation is stopped while other threads are running each at each turn. After a short while, stopped thread will run again for his time slice.

Multithreading is interesting because even with blocking operation, execution is done in parallel. Of course, if you use computer resource in thread processing, there is no miracle: everything is slowed down considerably. And switching from one thread to the other impose an overhead to the operating system.

All in all multithreading is ALWAYS globally slower than single threading.

The advantage is that system take care of thread scheduling and resource sharing. Another advantage is on multiprocessor system: each processor is probably able to execute one thread (not always because threads are using shared resources such as memory, disk, I/O,...).

Multithreading is complex to program because of parallel execution. Two threads can't access same variable at same time without problems! You have to use synchronisation objects such as mutex, semaphores or critical sections to make sure only one thread access a given variable or resource. Most Delphi components are not thread safe. They can't be used simultaneously by several threads. For example, two threads can't update user interface without doing special programming. Delphi provide Synchronize method which in fact defeat

multithreading by blocking a thread, wait until main thread is ready to execute a given procedure, start procedure execution by main thread, wait termination and restart blocked thread.

Conclusion

Observing the overhead & level of complexity involved, it is not recommended to develop a multithreading software.
Utilising Message Pump technique is already confusing, and multithreading is way more complex then that.

5.2.4.3 UDP versus TCP

Quoted directly from Piette (1997):

ADDRESSING

TCP and UDP use the same addressing scheme. An IP address (32 bits number, always written as four 8-bit number expressed as unsigned 3-digit decimal numbers separated by dots such as 193.174.25.26) and a port number (a 16-bit number expressed as a unsigned decimal number).

The IP address is used by the low-level protocol (IP) to route the datagram to the correct host on the specified network. Then the port number is used to route the datagram to the correct host process (a program on the host).

For a given protocol (TCP or UDP), a single host process can exist at a time to receive data sent to the given port. Usually one port is dedicated to one process.

UDP

UDP stands for User Datagram Protocol. It is described in STD-6/RFC-768 and provides a connectionless host-to-host communication path. UDP has minimal overhead; each packet on the network is composed of a small header and user data. It is called a UDP datagram.

UDP preserves datagram boundaries between the sender and the receiver. It means that the

receiver socket will receive an OnDataAvailable event for each datagram sent and the Receive method will return a complete datagram for each call. If the buffer is too small, the datagram will be truncated. If the buffer is too large, only one datagram is returned, the remaining buffer space is not touched.

→ UDP is connectionless. It means that a datagram can be sent at any moment without prior advertising, negotiation or preparation. Just send the datagram and hope the receiver is able to handle it.

→ UDP is an unreliable protocol. There is absolutely no guarantee that the datagram will be delivered to the destination host (but to be honest, the failure rate is very low on the Internet and nearly null on a LAN unless the bandwidth is full).

→ Not only the datagram can be undelivered, but also it can be delivered in an incorrect order. It means you can receive a packet before another one, even if the second has been sent before the first you just received. You can also receive the same packet twice.

→ Your application must be prepared to handle all those situations: missing datagram, duplicate datagram or datagram in the incorrect order. You must program error detection and correction. For example, if you need to transfer some file, you'd better set up a kind of zmodem protocol.

The main advantages for UDP are that datagram boundaries are respected, you can broadcast, and it is fast.

→ The main disadvantage is unreliability and therefore complicated to program at the application level.

TCP

TCP stands for Transmission Control Protocol. It is described in STD-7/RFC-793. TCP is a connection-oriented protocol that is responsible for reliable communication between two end processes. The unit of data transferred is called a stream, which is simply a sequence of bytes.

→ Being connection-oriented means that before actually transmitting data, you must open the connection between the two end points. The data can be transferred in full duplex (send and receive on a single connection). When the transfer is done, you have to close the connection to free system resources. Both ends know when the session is opened (begin) and is closed

(end). The data transfer cannot take place before both ends have agreed upon the connection. The connection can be closed by either side; the other is notified. Provision is made to close gracefully or just abort the connection.
Being stream oriented means that the data is an anonymous sequence of bytes. There is nothing to make data boundaries apparent. The receiver has no means of knowing how the data was actually transmitted. The sender can send many small data chunks and the receiver receive only one big chunk, or the sender can send a big chunk, the receiver receiving it in a number of smaller chunks. The only thing that is guaranteed is that all data sent will be received without any error and in the correct order. Should any error occur, it will automatically be corrected (retransmitted as needed) or the error will be notified if it can't be corrected.

At the program level, the TCP stream look like a flat file. When you write data to a flat file, and read it back later, you are absolutely unable to know if the data has been written in only one chunk or in several chunks. Unless you write something special to identify record boundaries, there is nothing you can do to learn it afterward. You can, for example, use CR or CR LF to delimit your records just like a flat text file.

At the programming level, to send data, you just need to call the Send method (or any variation such as SendStr) to give the data to be transmitted. It will be put in a buffer until it can be actually transmitted. Eventually the data will be sent in the background (the Send method returns immediately without waiting for the data to be transmitted) and the OnDataSent event will be generated once the buffer is emptied.

To receive data, a program must wait until it receives the OnDataAvailable event. This event is triggered each time a data packet comes from the lower level. The application must call the Receive method to actually get the data from the low-level buffers. You have to Receive all the data available or your program will go in an endless loop because TWSocket will trigger the OnDataAvailable again if you didn't Receive all the data.

As the data is a stream of bytes, your application must be prepared to receive data as sent from the sender, fragmented in several chunks or merged in bigger chunks. For example, if the sender sent "Hello " and then "World!", it is possible to get only one OnDataAvailable event and receive "Hello World!" in one chunk, or to get two events, one for "Hello " and the other for "World!". You can even receive more smaller chunks like "Hel", "lo wo" and "rld!". What happens depends on traffic load, router algorithms, random errors and many other parameters you can't control.

On the subject of client/server applications, most applications need to know command boundaries before being able to process data. As data boundaries are not always preserved,

you cannot suppose your server will receive a single complete command in one OnDataAvailable event. You can receive only part of a request or maybe two or more request merged in one chunk. To overcome this difficulty, you must use delimiters.

→ Most TCP/IP protocols, like SMTP, POP3, FTP and others, use CR/LF pair as command delimiter. Each client request is sent as is with a CR/LF pair appended. The server receives the data as it arrives, assembles it in a receive buffer, scans for CR/LF pairs to extract commands from the received stream, and removes them from the receive buffer.
Using CR/LF as delimiter is very handy because you can test your server using the well know application "telnet".

5.2.4.4 General Delphi Programming

This section contains things I found out during this project.

TTimer problem

In a lot of occasions, we need to time the operation. For example, we need a timer to measure a "ping" packet time out. And of course, for timing operations; such as timing the benchmark, timing the simulation process, and timing the reply-back time from clients.

The logical choice for it would be indeed the TTimer component that came from Delphi itself, or so I thought.

Turned out that it's a very bad choice and wasted several days of productive work due to debugging required to solve the problems it caused.

The root of the problem is very simple actually, TTimer is not accurate if there's another resource-intensive process running in the same time.

At first, I didn't consider this as a possibility when I first encountered a timing problem, because I didn't think of it as a possibility at all, especially if I remember the quality of the Delphi package overall.
I only started to consider this as a possibility after repeated executions under low resource usage's gave accurate timing results, while executions under high resource usage's yielded timing inaccuracies - sometimes in the measure of minutes (amazing if we consider that timing error of 1 second is already considered to be very bad!)

No TTimer-based workaround can be found to solve this particular problem.

The solution is to use Windows API "GetTickCount" which will provide us with 1 ms accuracy. But for that advantage, for every case we need to reconstruct the code properly, while if we're using TTimer we just need to put something to handle the OnTimer event. Anyway, this is a trade-off that simply can't be avoided, because otherwise our applications just won't work - it's too critical.

How TListView work

TListView is one of a component that's not documented so thoroughly in the Delphi's manual books.

The first time I discover it, I soon realise that the use of this control is essential for our project's success.
But at the first I was put off by the lack of good documentation & examples for it.

It looked easy at first, and it actually is. First you need to specify the columns & its properties (titles, etc), and then start adding items to it by calling ListItem.Add. But at the beginning I can't figure this out. I was confused on how to add a new row, and how to fill in the columns in that row.

Fast-forward several days, and then I figure it out (finally).
To add a row, call ListItem.Add.
To put a value in the first column of that row, assign some string to ListItem.Caption.
To put a value in the next column, assign some string to ListItem.SubItems.Add - repeat for the number of columns
To delete a column, call ListItem.SubItems.Delete(n). It will delete column number (n - 1), and the value of the next column will be moved to that columns. This was the part that confuses me the most.

Now it works, but I encountered one bug.

Somehow, at the first time we create a new row, we need to enter initial values to one more column than the number of columns we defined in Column Editor. Otherwise, Delphi will give us some exception error. Then we need to delete that extra column as well.

After that, it works flawlessly as it's supposed to.

Script Interpreter

In my project, I'll need to implement some kind of script interpreter, because my module will utilise script so it'll be able to simulate any network applications. I was thinking about implementing full-blown interpreter. But since the beginning I have doubted its feasibility from time viewpoint, because I'm no expert in this field. In fact, this will be the first time ever I touch this subject in detail.

After analysing different traffic patterns generated by different applications, I got a feeling that I should be able to construct a very simple script language to accommodate this software's needs, because over the network what happens basically is just this:

1. Server sends packet, with certain size
2. Sometimes there's a delay from the client before responding
3. Sometimes client reply back with packet of certain size
4. Sometimes there's a delay, before we loop back to point #1

Therefore I was able to design a strict script structure that will be easy to interpret, yet allows the script file to simulate any kind of network traffic pattern.

The script's structure is like this:

```
;-------------------------------------------------------------------------------
[NUS_simulator_script]
;
;loop_timingrecord

        delay_before_start=0
        timing_record=100

;       loop_transaction
                delay_before_transaction=0
                ;--- will send 10 packets in each transaction
                transaction_per_record=50
                ;--- number of packet sent during each transaction
                number_of_send=3

                delay_before_sending1=0
                ;--- each packet will by 300 bytes in size
                send_data1=300
                ;--- in return, request client to send 1000 bytes packet
                instruct_client_to_send1=1000
                ;--- with certain delay
                instruct_client_to_delay1=0
```

```
                delay_before_sending2=0
                ;--- each packet will by 300 bytes in size
                send_data2=300
                ;--- in return, request client to send 1000 bytes packet
                instruct_client_to_send2=1000
                ;--- with certain delay
                instruct_client_to_delay2=0

                delay_before_sending3=0
                ;--- each packet will by 300 bytes in size
                send_data3=300
                ;--- in return, request client to send 1000 bytes packet
                instruct_client_to_send3=1000
                ;--- with certain delay
                instruct_client_to_delay3=0

                delay_after_transaction=0
                disconnect_after_transaction=true
;       end_transactionloop

;end_timingloop
;-------------------------------------------------------------------------------
```

If it suspiciously looks like an INI file, it is. I utilise IniFile unit to read the file, so it has to be structured like a *.ini file. It has to:

1. Have a heading, which must be "[NUS_simulator_script]",
otherwise the script won't run

2. Under that heading, then all commands follows

3. Comments supported, prefixed by a ";"

The complete explanation of the commands listed here will be available from chapter 9.1 of this dissertation.

"Message Pump" technique

In the beginning of the development process, we found a problem in the benchmark module - we could not press the [CANCEL] button to cancel the benchmark currently running. We were confused for days, after all, Delphi is a programming language for Microsoft Windows - a multitasking OS - so why it can't do just a simple multitasking like this?
The answer comes with the knowledge of the inner working of Microsoft Windows.

I suspected that the current benchmark loop simply drained too much CPU power to it, and not much else is available for other threads - including the thread that handle the [CANCEL] button. After asking to a Delphi forum on Internet, apparently I was a bit wrong here. I was right that the current benchmark loop is draining too much CPU power to it, but the [CANCEL] button is not handled by a thread.

I learned that Microsoft Windows is a message-based system. Everything is communicated by messages. In a second, there could be hundreds of messages being transferred all over the system. This message transfer process is being handled by a Windows API, which is encapsulated in Delphi as Application.ProcessMessage method.

If we draw too much CPU power in an event-handler, then Application.ProcessMessage method won't get executed at all.
Therefore, when user press button and the button send a message to the message queue (to be transferred to [CANCEL] button event-handler), it won't be processed and stuck there.

The solution is by executing Application.ProcessMessage within our loop.
This will give a small timeslice for it to process any pending messages, and then it will quickly transfer control back to our application.

This technique is well known as Message Pump technique.

I was worried that doing this will create additional overhead that could slow down the performance of the software. Apparently, it's quite efficient in doing its job, because we're unable to detect any slowdown in the process. But it caused another issue to arise; re-entrancy.

By executing Application.ProcessMessage within our loop, we're making it possible for the user to keep interacting with our user-interface while the loop is running. This can cause problem indeed.
For example, when the loop is running I tried to exit the software by selecting the "Exit" menu. Instead of exiting gracefully, the software crashed. So we need to alter our code to take this into account.

Fortunately after thorough analysis of our situation, the solution is quite easy. Just disable all user-interface/menu items, and we are fine.

Software performance note

Our projects are different, but still have one thing in common - network performance analysis.
To be able to measure network performance, it has to be performing very well itself. Therefore, I've invested a large portion of my time dedicated to solve performance-related problems.
Thankfully, Delphi already generates a very efficient code, so I really don't need to do much with regard to Delphi programming itself. But I found some performance issues with Winsock instead.

I'm utilising TWSocket component to interface to Winsock. At first, my software performs well. But then after I specify a longer simulation time, it became apparent that it's putting the computer in where it's running under a very heavy load.
A Pentium III 500 MHz with 64 MB of RAM can barely do anything else if I run a 5 minutes simulation. And the simulation result varies wildly as well with longer simulation time.

Observing things manually, I suspected that something in my software is using memory intensively. But the question is, which one? I don't use much memory explicitly in my code, so it must be the TWSocket component.

Turned out that I was both right and wrong. Yes TWSocket is using a lot of memory, because first it buffers outgoing packets to memory before actually sending it to the network. But I'm wrong because it's not the only one, Winsock is doing the same as well!
Since our software is sending packets as quick as possible, especially Mobeen's benchmark module, two problems arise.

First we use a lot of memory because those packets are being buffered twice automatically - first by TWSocket, and second by Winsock.

Second, it can cause inaccuracies in results. This is because even though the server had finished doing benchmark, when I debug the client I saw that the client is still receiving packets ! Apparently in the background both TWSocket & Winsock are still clearing their buffers for several more seconds.
Also, if the memory is filled up with buffer which contain our packets, then Windows will start swapping out to/from disk – which will slowdown everything.

I got around the second problem, well, partially.
What I did is whenever the server is finished sending packets, it'll clear TWSocket's buffer immediately.
But I still haven't found a way to disable Winsock's buffering, and it's possible that I'll never will because the way buffering integrated to Winsock mechanism, is too tight.

Anyway, visual observation reveals that the client now stopped receiving packets when the server stopped as well - so I'm happy now.

I'm still looking for the solution of the first problem - how to disable buffering altogether. Unfortunately, both TWSocket & Winsock have buffering integrated rather tightly to their code, so it's very hard to disable it.

So the situation right now is:

1. The results are accurate
2. But the software (server software, to be precise) still uses computer resources heavily.

5.3 QUALITY ASSURANCE

In the project proposal, we have agreed that the final quality assurance process will be done exclusively by another team member, Mr. Hery Rakotonirina. He was appointed as Quality Assurance manager in that proposal. This is to ensure consistency in the end of the project, and a dedicated Quality Assurance manager will allow others to accomplish other tasks.

He will be responsible for:

- Researching proper software testing techniques, and inform the others about it.
- Oversee and manage all testing processes
- Advise colleagues on how to proceed with Alpha testing stage (Alpha-testing stage is where the developer does the testing)
- After the developers have finished the Alpha testing stage, the product will be handed in to him to be Beta tested (Beta testing stage is where the testing is done by external tester)
- After he's satisfied with the testing result, the software will be further tested by making it available on Internet. This way, the software will get maximum exposure and can have the most rigorous testing situation.
- Manage all incoming response from Internet beta tester
- Collate the test results, and hand it in to the developers, to be fixed.

On the development stage, a lot of efforts have been put to ensure maximum

software quality. Those efforts are listed below:

- Utilising RAD (Rapid Application Development) tool, in this case, Delphi. A RAD tool helps us to minimise number of potential human-error by masking out the details and let us concentrate on the main logic instead.
- Reusing established code. We use TWSocket to accomplish Winsock programming. And we utilise MiniChat application's source code as the base of our own basecode. Since both are well-known code and available on Internet; they've been tested extensively and we can use them to accomplish our project *sans* the bugs.
- Adhering to good programming practices. Such as putting comments as clearly as possible, avoiding vague "shortcuts", indenting the code to make it easier to read, making the variable/procedure/function names as clear as possible, avoiding crashes by always assigning default value first to variables whenever an input is expected from the user (so when they doesn't provide anything, the variable will still contain a value that's safe for the program's execution), etceteras.

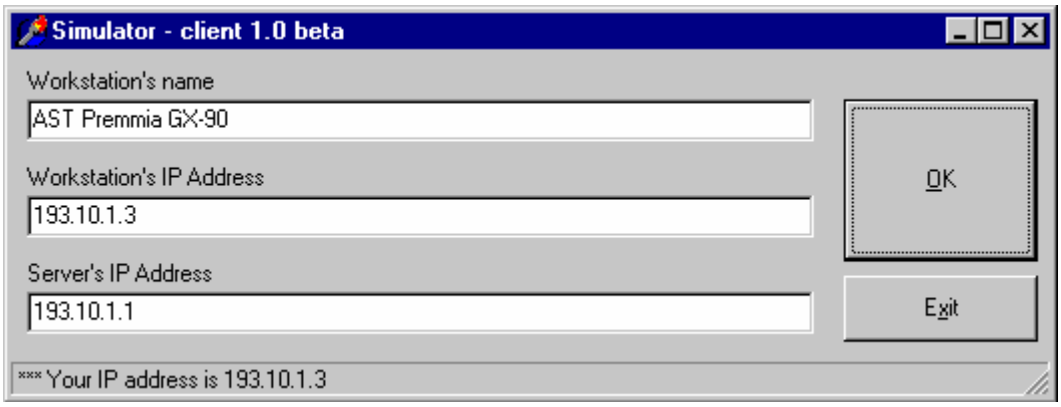
After doing the points above, every developer shall test each new addition to the source code before adding the new ones. This way, if a software bug surfaced, we can easily pinpoint the source of the problem. Each developer is responsible to try making his code as stable as possible before submitting it to the Quality Assurance manager.

As you can see, our quality assurance methods are quite robust. And the end result reflects this as well - it's very stable and rock-solid.

6.0 RESULT OF WORK UNDERTAKEN

6.1 Network Application Simulator - CLIENT

The client software was designed to be as simple as possible. As you can see from the screenshot, the user just need to enter 3 self-describing parameters; workstation's name – by which the workstation will be listed on the server screen, workstation's IP address – to register itself on the server, and server's IP address – to enable it to establish connection to the server.

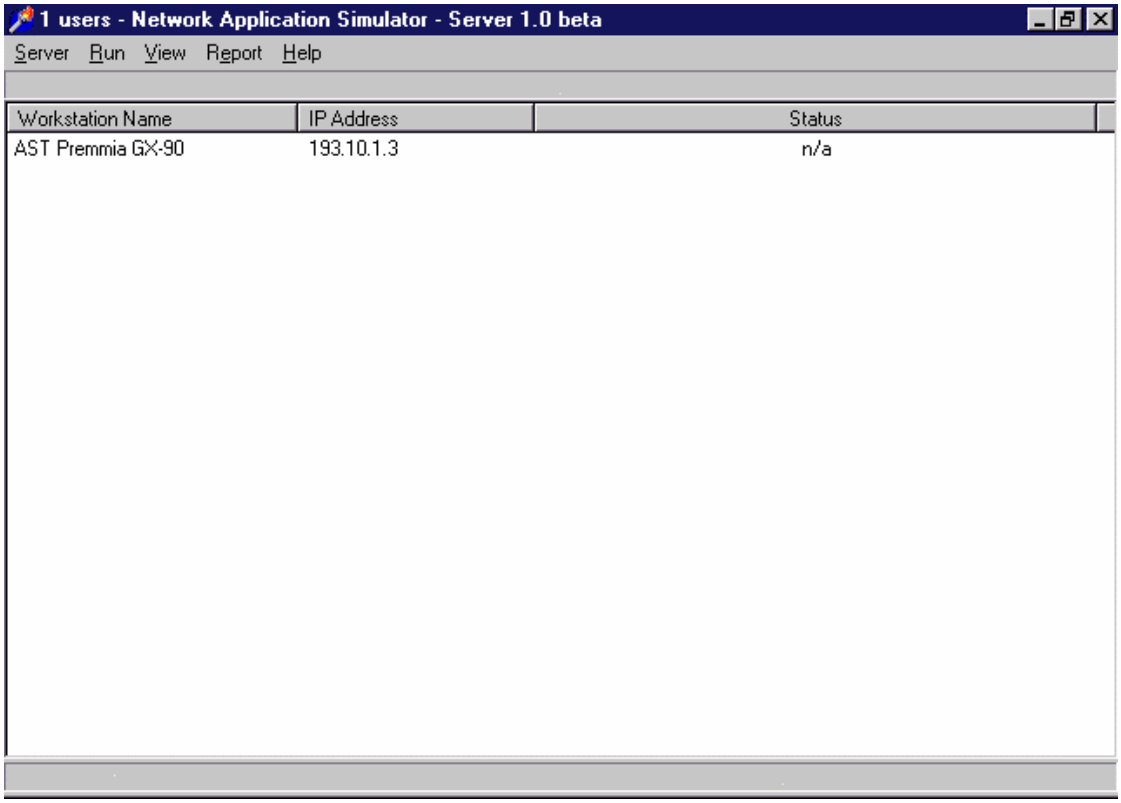


When the user press OK, the software will try to establish connection to the server. If it failed, it will inform it to the user. If it succeed, it will register itself to the server, and start the auto-reconnect feature.

With auto-reconnect feature, if the server suddenly becomes unavailable, the client will retry to establish connection to the server every 60 seconds.It will continue to do so until connection to the server is established, or the client is shutdown by the user by clicking on the "Exit" button. The auto-reconnect feature consume very little bandwidth, and it's done in a very wide interval (60 seconds) so the process won't overload the network even over a dial-up connection.

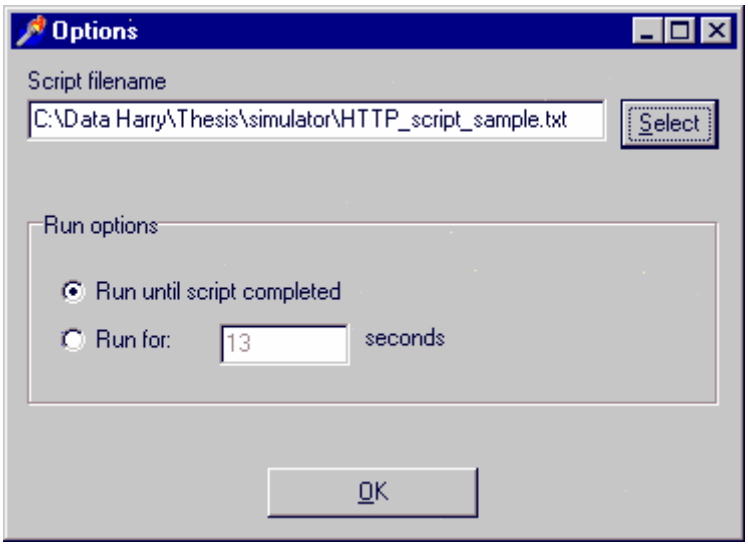
After the connection is established to the server, it will stay idle, ready for any instruction from the server. When any instruction arrives, it will execute it automatically, without bothering the user who is using the workstation. After it finished executing the instruction, it will return to its idle state, ready for the next instruction.

6.2 Network Application Simulator - SERVER



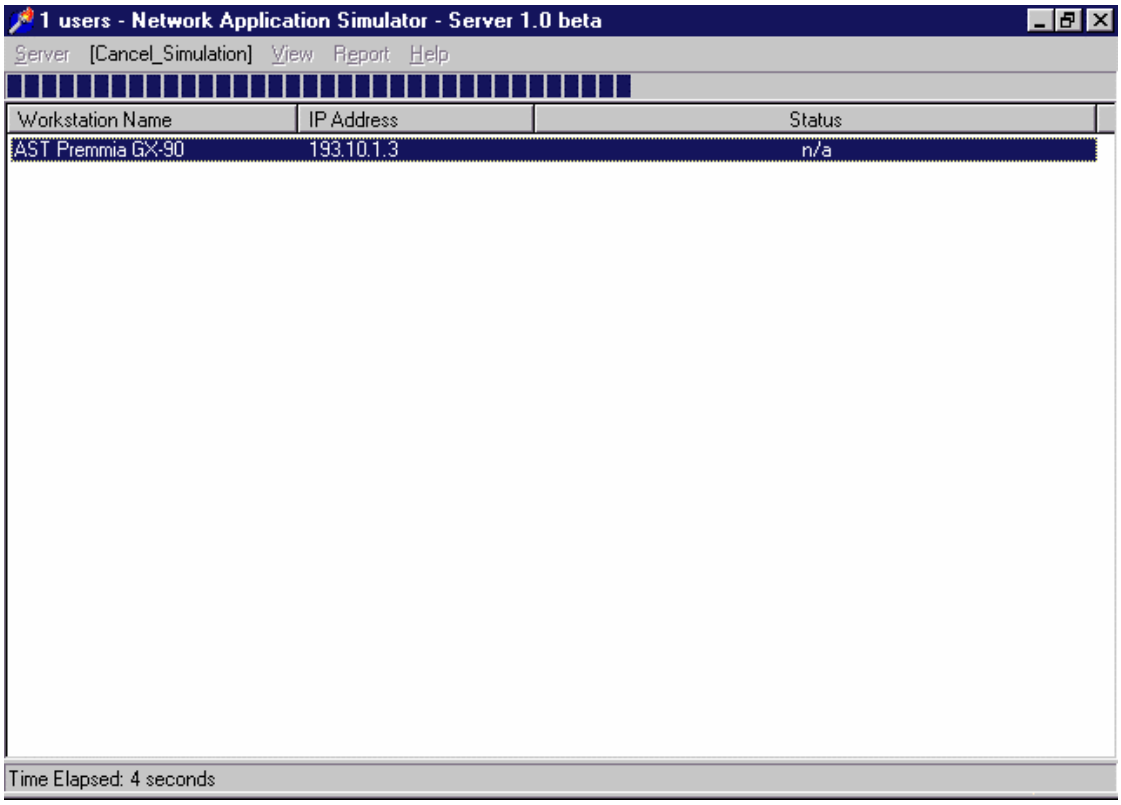
The screenshot above show how it looks like when we run the server software with one client connected. The status column is still empty, since we haven't run any simulation yet.

When we invoke the Run – Options menu, we will see the following dialog box:



There are basically two options there. The first one, “Script filename”, lets us to specify the script we’d like to use for the simulation. Just press on the “Select” button, and select the script file from the convenient Windows dialog box. The script that’s currently selected will simulate HTTP trafic to/from a webserver.

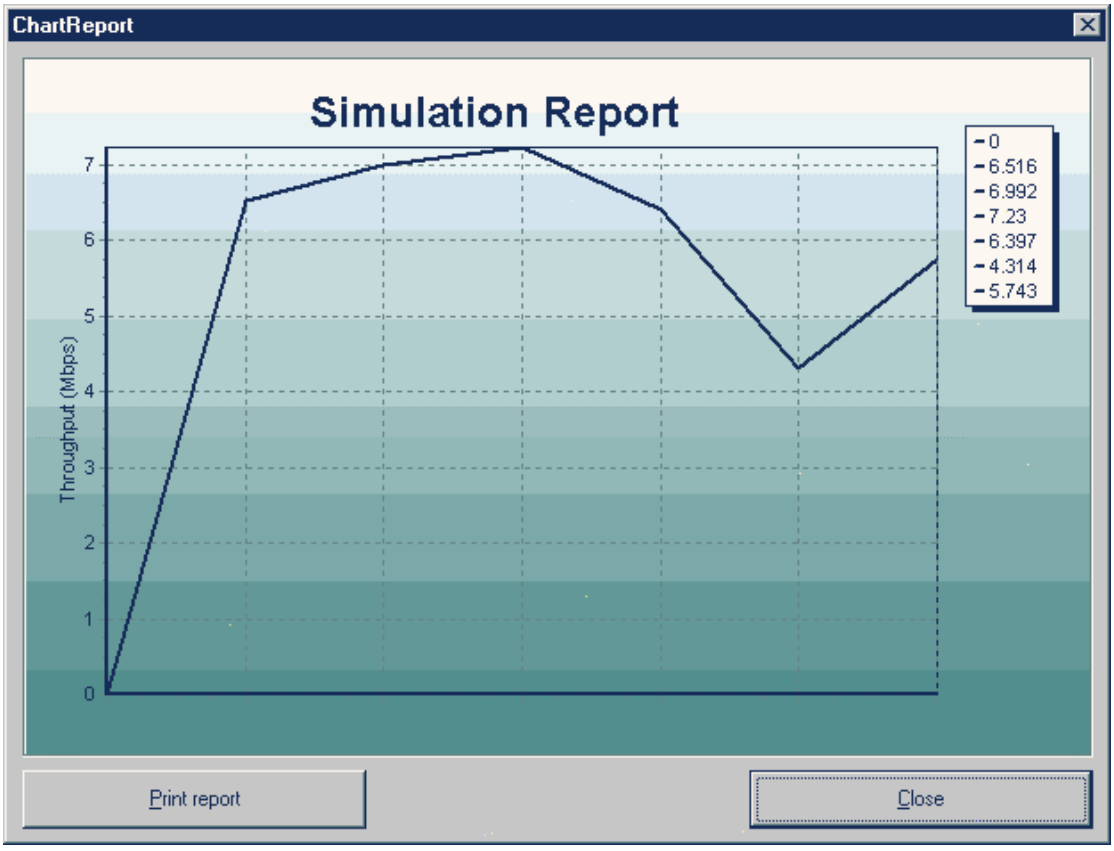
The second option, “Run options”, lets us to specify whether we’d like to run the test until the script finished, or whether we’d like to run it for a fixed pre-determined time. The second option is mostly useful when either our script is too long and will take too much time to execute, yet we don’t have time to wait for it to complete; or either it’s too short that we can’t get a good measurement. In those case, then we can specify for how long exactly we’d like the script to run.



After we set all options properly, we may start the simulation by selecting any client and clicking on the “Run” menu. All menu items will be disabled, and “Run” menu will be changed into “[Cancel_Simulation]”. By selecting that menu item, the user can cancel the simulation at any time. A sample of the software in action is available from the screenshot above.

When it finished executing the simulation, it will show us the following dialog box displaying the summary of the simulation:

Shown below is the graphic chart of the simulation process:

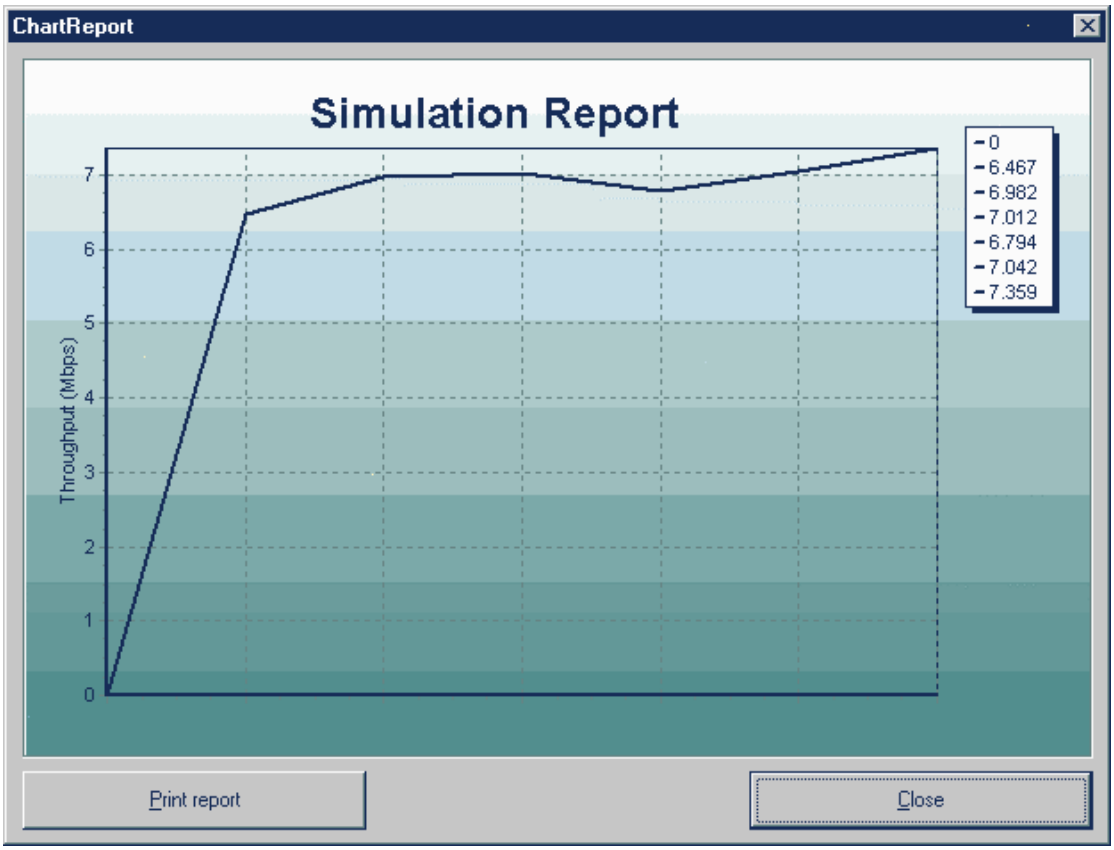


The graphic chart is quite versatile. We can zoom in, zoom out, move the chart around, and also print it to the printer.

You may see near the end of the simulation run there's a big drop in throughput – from around 7 Mbps down to 4.5 Mbps. This is because at that time I invoked the screen

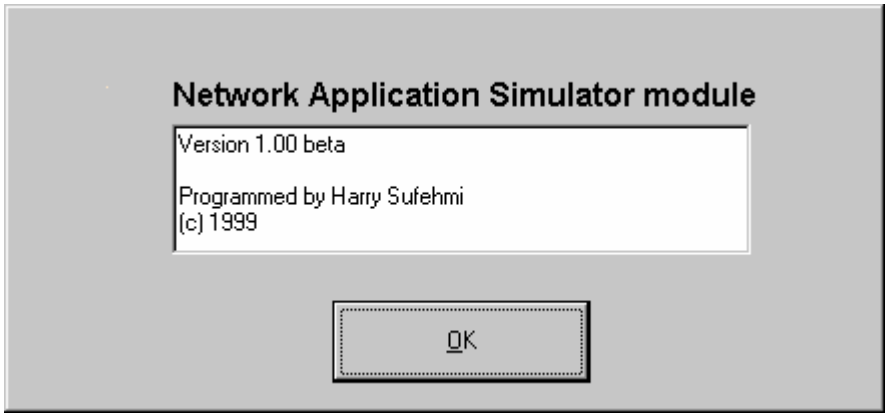
capture utility. The software capture the screen, but in that particular instant also consume almost 100% of CPU time, so the server software didn't get much processing power to run the simulation. Hence the drop in the throughput. But after the screen was captured, all CPU time was available again to the server software, and you can see that in the end the throughput was rising again.

So then I re-run the test again without any interruption, and this is the actual result:



Now it's pretty much settled on 7 Mbps mark.

And finally, the following dialog box will be shown if we select the Help – About menu:


Network Application Simulator module
Version 1.00 beta
Programmed by Harry Sufehmi
(c) 1999
OK

7.0 CONCLUSIONS

7.1 PROBLEMS ENCOUNTERED

Writing software is never easy work for an average person. There are exceptions, such as Free Software Foundation's Richard Stallman and Linux kernel maintainer's Alan Cox who write computer programming codes like we speak English. But the rest of us had a lot of problems constantly writing software.

Listed below are the problems we encountered during the duration of the project:

7.1.1. Wrong approach

The first time we started building our software, we agreed that each of us will built it from scratch. How each of us will code the software shall be irrelevant, and all we need to agree on is just the communication protocol, to ensure that our end product will be able to communicate to each other.

We found out after several months that this is a very wrong approach.

First, the initial overheads, from the research viewpoint, are very, very high. Especially the fact that we were experiencing a lack of documentation problem (see the next point) made the overhead even higher.
Windows programming is non-linear, it's event based because it's able to multitask. Things will not happen in certain sequence, you have to be prepared for anything - especially if you implement the Message Pump technique.
This is very hard to grasp for programmers that are used to dwell on DOS-based programming language.
And then network programming, in this case Winsock interfacing, is not just like putting a stream of data on certain ports and that's it; we got to handle many exceptions based on many rules. And those rules can (seem to) be conflicting sometimes, enough to baffle even the most experienced programmers.
We will need some time to do some research and familiarise ourselves with it.

If we choose to build everything independently, then each of us will have to go through these steps.
But, if we can have a common code on which we can concentrate on the algorithm instead on

the details, we will become much more efficient.

Second, building each of our software from ground-up separately can cause serious inconsistencies in the user interface.
For software projects that intended to be unified in the end to become a single software suite, this would be a very fatal mistake. The user could become frustrated in using our product, will make mistakes easily; and in the end may refuse altogether to use it again in the future.

Third, very different source code may in the end prevent us from merging all three projects easily. If in the end we may be able to convert it, it's possible that it will contain new software bugs, due to different ways of interfacing between modules.

Therefore, by observing those problems, we then decided to develop a basecode. A basecode is a common code, a foundation, on which then we will develop our own project's specific codes. This basecode shall cover three aspects:

- Basic user interface
- Network interface
- Basic client/server communication protocol

This was not an easy decision to take, because we already do several months of work. If we implement basecode, we would have to scrap our previous software build and develop the basecode again from the beginning - back to square one.

But after weighing the pros and cons carefully, finally we decided to go to basecode route.

Indeed we lost several months of work, but the result is worth it. In parallel to the advantages listed above, we also gained several more advantages.

First, now our software can be easily made to have common look-and-feel. Common look-and-feel will greatly benefit the users that they will feel comfortable using any of the modules that we're currently developing. So it will actually encourage user to use the product.

Second, the basecode architecture makes it easy to further enhance the software rapidly. Also some major changes to the design & code means that this base code will be highly extensible. Particularly the fact that we're utilising Delphi's linked-list & ListView component.

Third, the code is much cleaner now. Cleaner code will make it easy for us, and other programmers as well, to work on it. Also it will make it harder to make a mistake.

Fourth, we were amazed on how fast we can develop our project on top of it. Developing the basecode took months, but then, each of our projects can be developed in days. Delphi is already a very powerful OO RAD (Object Oriented Rapid Application Development) tool, and with our basecode we're able to develop the software even faster.

All in all, we're very happy with our decision. Our only regret is that we didn't do it from the beginning, so we wasted quite a lot of time. But then again, this is our first attempt to undertake this kind of project, we have no previous experience about it. So if we think about it, we're pretty much satisfied with ourselves.

7.1.2. Lack of documentation

We experienced this problem in almost every aspect. For example, we were stuck once in Delphi because we haven't got enough documentation on how the TListView object works. After much trial and error and several days, finally we figured it out. Delphi came with a large help file, but even it wasn't much of help for us. Books on Delphi also only touch its surface. When we dived deeper into it, we're basically on our own.

Then, we were stuck on several Windows API (Application Programming Interface), but then again, it's already quite notorious for being hard to understand.

And finally, the Winsock documentation is not one among the best as well.

We can say that it's quite frustrating at times, and definitely slowed down the development pace in many occasions.

7.1.3. Project Management

This is an area that we had already anticipated as a possible source of problems, because none of us have previous software project management experience before. Problems arose in many forms, but after thinking about it we agree that it all boils down to one: communication.

We have setup many ways on which we can communicate. Phone, email, alternate phone numbers, and we even gave our home address to each other. But coupled with the fact that each of us also have different commitments (such as: family, etc) we still experienced some communication problems. As such, sometimes it took several days before a research result

can be informed to a team member.

7.1.4. Steep learning curve

This is a new playground for us. We boldly go where we never go before.

The main background of the writer of this thesis is DOS database programming. Other team members' background is DOS C programming. All of them follow the same paradigm: linear programming. When we create the program, we know exactly what will happen after a certain line of code is executed. Everything is executed in sequence, nothing can happen out of control.

When we started this project, we had to face a new programming paradigm: event-based programming.
Microsoft Windows operating system is heavily based on this.

What is event based programming?
Basically, our software will stay passive on background, and waiting for something to happen. If something happened, which could be anything - user accessing mouse/keyboard, incoming data, new window popped up, etc - Windows will send a message to our application. Our application will examine whether the incoming message has a code to handle it. If not, then it will be ignored. If yes, then the related code will be executed.

So far it's still rather easy to understand. But unfortunately (or fortunately, depending on how you look at it), Windows is a multitasking operating system.
So, in a second there could be hundreds of messages flying around all over the system, invoking many handlers. And worst, the same handler can be re-invoked again while it's still executing. This is called re-entrancy problem. As you can see, now we're in for some serious headaches.

That's already pretty hard to digest and to overcome. Initially, we simply didn't realise it, we knew it but we didn't realise when we encountered problems related to it. Only after our friend reminded and explained it again we realised what happened. It took us several days to re-code our software to handle this properly.

Then, we also face Windows API. Then Network/WAN (Wide Area Network) concepts. Then Winsock interfacing & programming. Etceteras.

It was too much for us, we thought we would never be able to produce something from this project.

Thankfully, the solution for this is rather easy: by taking things one by one, step by step.
So we're not overwhelmed as before, although it's still a steep learning curve for us.

But the throwback is that we consume more time while we take it one by one - and time is one of the resources that we don't have in abundance. So in the end, sometimes we still found ourselves rushing through things to meet the deadlines.

7.2 FURTHER WORK

By no means is this the end of the work. A software is never finished, we just have to decide at a point to release it.

The same applies to our software here. There are still some areas to be improved/worked on. We list some of them below, but please bear in mind that it's not all, there are still much, much more features that we can add to this software. But one thing that we also have to remember is to prioritise. Those features won't be of much use if the software itself is constantly in development.

So here it is, the list of some important features/things that still need to be worked on:

7.2.1. Implement multi client feature

Due to the time constraint, we haven't got the chance to implement this feature. So currently the software could simulate only one client. Anyway, it's already enough to proof that our concept can be implemented and is working.

But the foundation needed and pre-requisite to build this feature is already there, so it should be not be hard to do it.

7.2.2. Implement plug-in system, enabling easy upgrade for client/server

We were aiming for self-upgrade capability for the client. Therefore, anytime we add new features, we just need to update the server, and then the server can notify all clients to self-upgrade themselves. This is a very powerful feature that's not available in most software available today, and can give our software a significant advantage over the others because

then we can easily extend our software to do practically anything.

For example, with this feature implemented, we can even do remote installation. So if the network administrator needs to upgrade Netscape software in all 500 workstations - no problem, just invoke the NUS server and specify the location of the new package and parameters needed to install it. The server in turn will contact all clients, transfer the file, and the clients will then install it on the computer they are running at.
So the next time the users run the workstations, they will already be using the latest version of Netscape, without the network administrator having to visit each and every workstation to install it by him/herself.

And more, the possible usage of this feature is only limited to our imagination. You can see for yourself that it will be massively helpful to the network administrators.

Nearing the end of our project however, we got information of the existence of a Delphi component called UIL Plugin System v5.0 (http://www.uil.net/uilps5.html). Based on information on its website, it should be able to do what we want to do even more easily, and best of all it's free and comes with the source code. It looks very interesting indeed.

7.2.3. Provide a script generator

Right now, even though NetSim scripting language is easy to use, gathering the data needed to create a script is still a rather cumbersome task. The following are the steps that one will need to do to accomplish it:

- Start a network analyser/packet capture software
- Start the application software, and in the same time start capturing network packets
- After finished doing the activity, stop capturing network packets
- Analysing the packets captured, and start creating script from it

All of those activities above, except step number 2, could be done by a single application. Time saving obtained from utilising a script generator software will be substantial, especially on step number 4. And it also will dramatically reduce the possibility of human error; because almost everything is done automatically.
It'll be a major convenience feature for the user.

In order to accomplish this, we'll need to insert our own hook in Winsock. An issue arises though, because the easiest way to do this is by having Winsock version 2.0 installed. WindowsNT machines already have Winsock v2.0 installed by default, unfortunately, most others still have Winsock v1.1 as the default installation. And even after that, it's still not that

easy.

7.2.4. Porting the client to other platforms - especially Solaris and Linux.

Actually this feature is not very important, because the majority of desktop computers (which will run the client software) run Microsoft Windows operating system. Sun's Solaris and Linux are still mostly used to run the servers.

We included this task in the beginning because we wanted to learn more about UNIX network programming. We were intrigued by our experiences when doing the first UNIX programming in our course. It is a very enjoyable experience, everything seems to be very much in order and documentation is easy to be found (unlike Windows programming). In case of problem, it's easier to find people willing to help us on the Internet too. It is an experience that we're looking forward to.

7.2.5. Further increasing the accuracy of the simulator - need to implement hooks into Winsock

We have done what we can in the allocated time to make the software give the most accurate results as possible. What we have done are:

- Significantly reducing the overhead, by making the code as tight as possible
- Disabling background send - previously even after the server finished sending packets, the client may still receive some more packets for several seconds. This is because in the background, some packets are still being sent from the packet buffer.

But still, a little bit more accuracy can be gained, if we can disable buffering altogether in Winsock itself.

Let's step back a little first, perhaps you'll ask: How can buffering affect accuracy?

It's actually quite easy to understand, once you know how Winsock works.
This is because while Winsock buffer outgoing packets from our software, it uses up memory. From our experience, it took only 30 seconds for the packet buffer to completely fill up 64 MB of memory. Afterwards, Windows, detecting that it has run out of memory, starting to swap to/from hard disk. But this really slows everything down, because even the fastest hard disk is still about 1000 times slower than RAM. Our software is affected as well, it can't send packets and record statistical data as fast as it should be. We got a system-wide slowdown now.

Unfortunately, the buffering facility is build rather tight into Winsock itself, so even if we're able to disable it, we're at a major risk of making every other software that depends on Winsock to become unstable. So we need to find out how to disable buffering, and ensuring that every other software, which perhaps currently numbered in thousands, will not be affected by it. Not a small feat indeed.

7.2.6. Further stabilise the code

This is actually a never ending task, but a very important one nonetheless, especially after implementing any of the new features listed above.

7.3 RECOMMENDATIONS

For anybody attempting to do a similar project, we can recommend several things to do based on our experience. They are all listed below:

7.3.1. Utilise the Internet

If we have to do this project in 1993, we don't think we will ever be able to finish it on time. This kind of project requires a lot of information that was still very hard to obtain at that time. And some of them are not available for free, which will be a very difficult problem for students like us.

Thanks to the Internet, now you're overloaded with information. Almost everything is there. Now it only depends on your skill to search through all the junk and find what you need. Or, you can also chat online and/or discuss your problem in mailing lists. Finally you can access a massive source of knowledge easily.

7.3.2. Keep communications going on

With a project this size, we can't emphasize this enough: communication, communication, and communication.

Lack of communication is a sure recipe for failure.

7.3.3. Get hands-on experience

With a project that is very technical in nature like this, you can't just drown yourself in theory books and hope it will get completed somehow. It's called daydreaming. You simply got to get your hands dirty. It's a jungle of techno horror, but you have to get yourself comfortable with it, you must overcome your fears. Projects like this are pushing yourself beyond your limits (or, what you previously thought as your limits).

It involves a lot of trial-and-error process, especially in case of lack of documentation. It involves a lot of stress. But it's the way if you want to get the project finished on time and as planned. And in the end, you'll be rewarded with precious experience you won't get just from reading the books.

7.3.4. Go through the project step-by-step, don't try to achieve the final result in one try

It is a very sensible thing to do, but somehow people keep on forgetting this.
In the beginning of the project, sometimes you're so full of vigor you think you can take on the world.

We made the mistake of trying to digest everything in one take. Bad move. We were overwhelmed with the new knowledge we encountered. In the end, we could hardly make sense of anything. Then, we feared that we will never be able to complete this project, it reduced the morale of the team's members.

Then we carefully reviewed of what we had done. At the time we realise that we had underestimated the scope of our project. We then devised a new strategy in order to tackle this project. First, we need to learn things one by one, but as fast as possible. Second, we need to seek external help - which we found in abundance on the Internet.

Thankfully, we realised this just in time, and got this project finished on time.

7.4 CONCLUSIONS

This project has borne its first fruit - a working network application simulator.

Whenever we show the product, everyone's impressed, so we think we must have hit the right spot with this project. Sure there are still some more things to do, but with the first step done right, the road to the future is now easier to be traveled.

Although it's been going well, we could've been better. We made so many mistakes, we're rather embarrassed when we looked back.

But, to think that this is the first time we tackle a project with a scope this big, and so technical in nature, with inexperienced team members, and yet still managed to wrap it up on time - it's actually quite beyond expectations.

All in all, this project has been a very rewarding educational experience for us.

8.0 REFERENCES

Please note that not all references are used explicitly on this dissertation, many of them are used for our brainstorming sessions, or to inspire ideas on how to solve particular programming problem and/or how to implement certain features, etceteras.

Arsham, Hossein (1999), *Systems Simulation course*, http://ubmail.ubalt.edu/~harsham/simulation/sim.htm

Brooks, Frederick P. (1975) (Ed.), *The Mythical Man-Month*, North Carolina, Addison-Wesley.

Cartwright, Dave (1999), *Ganymede Chariot 2.2: Network application testing and simulation tool*, http://www.networkweek.com/applications/story/NWW19990316S0002

Currier, Robert (1999), *Review: Network simulation tools*, http://www.idg.net/crd_comparison_9-131380.html, idg.net

Cusamano and Selby (1995), *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*, New York, Simon & Schuster

Feldmann *et al*. (1998), *Dynamics of IP Traffic: A study of the role of variability and the impact of control*, http://www.acm.org/sigcomm/sigcomm99/papers/session8-3.pdf,

Fishwick, Paul A. (1994), *COMPUTER SIMULATION: GROWTH THROUGH EXTENSION*, http://www.cis.ufl.edu/~fishwick/paper/paper.html

Ganymede Software Inc. (1999), *Chariot Product Information*, http://www.ganymede.com/html/chariot.htm

gar@caciast.com (1999), *ComNet III Overview*, http://www.caciasl.com/comnet/netperform/comnet_main.cfm

Greis, Marc (1997), *NS - Network Simulator*, http://www-mash.cs.berkeley.edu/ns/

Helmy, Ahmed *et al.* (1997), *VINT - Virtual InterNetwork Testbed*, http://netweb.usc.edu/vint/

Incera, Jose. A (1999), *Network simulation bookmarks*, http://www.rennes.enst-bretagne.fr/~incera/Simulation/SimLinks.html

McDonald, Chris (1999), *The cnet network simulator*, http://renoir.vill.edu/~cassel/cnet/doc

Mercury Interactive Corp. (1999), *LoadRunner 6 Guide*, http://www.merc-int.com/products/loadrunguide.html

NetReference (1997), *Service Level Agreements*, http://www.netreference.com/PublishedArchive/WhitePapers/SLA_wp/SLA_white_paper.html

Nusenoff, Ron (1996), *Software Success: Microsoft's Method*, USA, IEEE

Piette, Francois (1997), *A TCP/UDP primer*, http://www.rtfm.be/fpiette/docsuk.htm

Piette, Francois (1997), *Event-drived vs Multithreading programming*, http://www.rtfm.be/fpiette/docsuk.htm

Quinn and Shute (1996), *Windows Sockets Network Programming*, Reading, Addison-Wesley.

Siegel and Passmore (1997), *Network Quality of Service - What's Good Enough?*, http://www.netreference.com/PublishedArchive/WhitePapers/QoS_wp/QoS_wp.html

Young, Warren (1999), *Winsock FAQ (Frequently Asked Questions)*, http://www.cyberport.com/%7Etangent/programming/winsock/

9.0 APPENDIX

9.1 NetSim scripting language - documentation

Creating a script to be used by NUS simulator is very easy. Below is a sample script, which simulate HTTP traffic, that will give an idea of how the script is constructed:

```
;-------------------------------------------------------------------------------
[NUS_simulator_script]
;
;loop_timingrecord

        delay_before_start=0
        timing_record=100

;       loop_transaction
                delay_before_transaction=0
                ;--- will send 10 packets in each transaction
                transaction_per_record=50
                ;--- number of packet sent during each transaction
                number_of_send=3

                delay_before_sending1=0
                ;--- each packet will by 300 bytes in size
                send_data1=300
                ;--- in return, request client to send 1000 bytes packet
                instruct_client_to_send1=1000
                ;--- with certain delay
                instruct_client_to_delay1=0

                delay_before_sending2=0
                ;--- each packet will by 300 bytes in size
                send_data2=300
                ;--- in return, request client to send 1000 bytes packet
                instruct_client_to_send2=1000
                ;--- with certain delay
                instruct_client_to_delay2=0

                delay_before_sending3=0
                ;--- each packet will by 300 bytes in size
                send_data3=300
                ;--- in return, request client to send 1000 bytes packet
                instruct_client_to_send3=1000
                ;--- with certain delay
                instruct_client_to_delay3=0

                delay_after_transaction=0
                disconnect_after_transaction=true
;       end_transactionloop
```

```
;end_timingloop
;-------------------------------------------------------------------------------
```

The script is consisted of two major loops.

The first loop is the timing loop.
Timing loop is where NUS does the timing process, that's it, measuring time elapsed and bandwidth used.

The second loop is the transaction loop.
This is where everything happened. In it, there are *send-sequence* that will simulate actual communications between client and server.
Send-sequence is consisted of delay before sending, data transfer to client, another delay, and finally data transfer to server.

The normal practice is that you specify transaction loop as one, and timing loop as many as possible, in order to get better average of the results.
But there are some exceptions in where the application does the same transaction several times - and this is where you will specify something else than one to the transaction loop. Or, you may also specify transaction loop as many as possible to put extra load to observe the result, etceteras.

Listed below are the syntax descriptions of statements in the script. All script have default values, so in case of the user forgetting to specify one or some of them, the software will not crash unexpectedly. Also, actually you don't need to construct the script in the order as shown above, because the software is able to find and do it in the correct sequence - further error-proofing the software. However, constructing the script by following the example above will make it easier to debug the script should something wrong happened.

; (semicolon)
Every line that started with this character will be treated as remark/comment and will not be processed.
You may put as many of them as you wish in a script. However, you may not mix instruction with remark in the same line.

delay_before_start
Before we start the whole simulation, the software will wait for the duration specified in this parameter. (in miliseconds)
Default value: 0

timing_record

This is where NUS gather the statistical data. After each completion of a timing record, NUS will gather the data of time elapsed for that particular timing record, and also the number of bytes transferred. In the end of the simulation run, those data will be reported to the user.
Default value: 1

delay_before_transaction

If anything positive specified as the value of this parameter, the software will pause for the length of its duration before proceeding to the next transaction. (in miliseconds)
Default value: 0

transaction_per_record

This parameter specify how many times we will repeat the transaction for each timing record.
Default value: 10

number_of_send

This parameter specify how many send-sequence will be executed during each transaction.
Default value: 1

delay_before_sending<n>

If anything positive specified as the value of this parameter, the software will pause for the length of its duration before sending data from server to the client. (in miliseconds)
Substitute "<n>" with the current send sequence number.
Default value: 0

send_data<n>

This parameter instruct the software to send a packet from server to the client with the size as specified (in bytes).
Substitute "<n>" with the current send sequence.
Default value: 100

instruct_client_to_send<n>

This parameter instruct the software to send an instruction to the client, to send a packet from client to the server, with the size as specified (in bytes).
Substitute "<n>" with the current send sequence.
Default value: 100

instruct_client_to_delay<n>

If anything positive specified as the value of this parameter, the client will pause for the length of its duration before sending data to the server. (in miliseconds)
Substitute "<n>" with the current send sequence.

Default value: 0

delay_after_transaction

If anything positive specified as the value of this parameter, the software will pause for the length of its duration after completing a transaction. (in miliseconds)

Default value: 0

disconnect_after_transaction

If set as True, then the software will simulate the client's disconnection from the server.

Default value: False

9.2 System Design Document

Available as separate document, handed in together with the dissertation.

9.3 Network Application Simulator - Source code

9.3.1 SERVER source code

MAIN_FORM.PAS

```
unit main_form;
{

 SIMULATOR Server code for NUS project
 http://welcome.to/nus_project

 ------------------------------------------------------

 Code based on MiniChat software version 1.0 beta by Steve Williams
 Utilising TWsocket component by F. Piette

 ------------------------------------------------------


BUGS:
1. After running the simulation for the first time, the result won't show up on status
column
Only after running it again (and so on) it will show up properly there.
--->>> FIXED
2. Progress bar doesn't end properly
--->>> FIXED
3. User can select menus when simulation is running, e.g.: Exit --> application
crashed
--->>> FIXED
4. Still simulating only 1 application client, need to implement multithreading

}


interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ComCtrls, Menus, WSocket, StdCtrls, Winsock, ExtCtrls, About_dialogbox, Options;

const
  WM_CLIENTQUIT = WM_USER + 10;

type
  TUser = class(TObject)
  protected
    FBuffer: array [0..1023] of Char;
    FBufLen: Integer;
    FSocket: TWSocket;
```

```
    FName: String;              { Name of the client }
    FNick: String;              { client's IP address, as reported by the client
itself }
    FAddress: String;           { client's IP address as well, from
TWSocket.GetPeerAddr}
    FStatus: String;            { client's status }
    FPing: Integer;
    procedure SocketSessionClosed(Sender: TObject; Error: Word);
    procedure SocketDataAvailable(Sender: TObject; Error: Word);
    procedure Send(Line: String);
    procedure ProcessLine(Line: String);
  public
    ListItem: TListItem;
    FirstTimeAccess: Boolean;
    constructor Create(ASocket: TSocket);
    destructor Destroy; override;
    property Nick: String read FNick write FNick;
    property Address: String read FAddress write FAddress;
    property Status: String read FStatus write FStatus;
  end;

  TfrmMain = class(TForm)
    mnuMain: TMainMenu;
    miServer: TMenuItem;
    miServerExit: TMenuItem;
    sbrMain: TStatusBar;
    miView: TMenuItem;
    miHelp: TMenuItem;
    miViewToolbar: TMenuItem;
    miViewStatusbar: TMenuItem;
    miViewSep1: TMenuItem;
    miViewOptions: TMenuItem;
    miHelpTopics: TMenuItem;
    miHelpSep1: TMenuItem;
    miHelpAbout: TMenuItem;
    sktMain: TWSocket;
    lvwUsers: TListView;
    SEND1: TMenuItem;
    ProgressBar1: TProgressBar;
    Run2FinishStatBar: TStatusBar;
    miReport: TMenuItem;
    Chart1: TMenuItem;
    Statistic1: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure miServerExitClick(Sender: TObject);
    procedure miViewClick(Sender: TObject);
    procedure miViewStatusbarClick(Sender: TObject);
    procedure miHelpAboutClick(Sender: TObject);
    procedure sktMainSessionAvailable(Sender: TObject; Error: Word);
    procedure FormShow(Sender: TObject);
    procedure lvwUsersInsert(Sender: TObject; Item: TListItem);
    procedure lvwUsersDeletion(Sender: TObject; Item: TListItem);
    procedure miViewOptionsClick(Sender: TObject);
```

```
    procedure SEND1Click(Sender: TObject);
    procedure sktMainDataSent(Sender: TObject; Error: Word);
    procedure Chart1Click(Sender: TObject);
  private
    FStarted: Boolean;
    FPort: String;
    FShuttingDown: Boolean;
    procedure WMClientQuit(var Msg: TMessage); message WM_CLIENTQUIT;
  public
    function  UserNick(User: TUser; Nick: String; Name: String): Boolean;
    procedure SendMessage(From: TUser; ToNick, Msg: String);
    procedure Ping(From: TUser; ToNick, TickCount: String);
    procedure Pong(From: TUser; ToNick, TickCount: String);
    function FindNick(Nick: String): TUser;
    procedure SetCaption(Count: Integer);
    function GetNextToken(var Line: String): String;
    function DupChars(WhatChar: Char; HowMany: Integer): String;
    procedure Delay(WaitFor: Integer);
    procedure Disconnect();
  end;


var
  frmMain  : TfrmMain;
  Counter: Integer;
  RunTimed, Run2Finish: boolean;
  RunDuration: Integer;
  Script_filename: string;
  Number_of_simulated_clients: Integer;
  CurrentlyRunning: boolean;
  Finished: boolean;
  AllDataInBufferSent: Boolean;

   { variables that controls the running of a script }
   timing_record, delay_before_start, delay_before_transaction,
   transaction_per_record, number_of_send, delay_after_transaction: Integer;
   disconnect_after_transaction: Boolean;
   delay_before_sending, send_data, client_send, client_delay: array of Integer;

   { will be used to measure performance between each timing record }
   start_timing, end_timing: Integer;
   time_elapsed, bytes_transferred: array of Integer;
   throughput_stat: array of Real;



implementation

{$R *.DFM}

uses
  IniFiles, Notify1, Report1_chart;

const
```

```
  AppName = 'Network Application Simulator - Server';
  AppVersion = '1.0 beta';

var
  dirApplication: String;

{ TUser list section ------------------------------------- }

{ Create a client socket and assign it some values. }
constructor TUser.Create(ASocket: TSocket);
begin
  inherited Create;
  FSocket := TWSocket.Create(nil);
  FSocket.HSocket := ASocket;
  FSocket.Tag := Integer(Self);
  FSocket.OnDataAvailable := SocketDataAvailable;
  FSocket.OnSessionClosed := SocketSessionClosed;
  FBufLen := 0;
  FPing := 0;
  FAddress := FSocket.GetPeerAddr;
  FirstTimeAccess := True;
  { Ping client immediately. }
  Send(Format('001 server %d', [GetTickCount]));
end;

{ Close the socket and free it. }
destructor TUser.Destroy;
begin
  FSocket.Abort;
  FSocket.Free;
  inherited Destroy;
end;

{ Send data to the client. }
procedure TUser.Send(Line: String);
begin
  FSocket.SendStr(Line + #13#10);
end;

{ The connection was closed. }
procedure TUser.SocketSessionClosed(Sender: TObject; Error: Word);
begin
  { Invoke clean-up routine by sending WM_CLIENTQUIT message
    to frmMain. The clean-up routine can be seen in frmMain.WMClientQuit procedure }
  PostMessage(frmMain.Handle, WM_CLIENTQUIT, 0, Integer(Self));
end;

{ Received some data. }
procedure TUser.SocketDataAvailable(Sender: TObject; Error: Word);
var
  Len, i: Integer;
begin
  { Retrieve the data, and place it at the end of the buffer. }
  Len := TWSocket(Sender).Receive(@FBuffer[FBufLen], SizeOf(FBuffer) - FBufLen - 1);
```

```
  if Len <= 0 then
    Exit;
  { Add the data length to the buffer count. }
  Inc(FBufLen, Len);
  { Place a null byte at the end of the buffer. }
  FBuffer[FBufLen] := #0;
  { Scan the buffer for complete lines. }
  while True do
  begin
    { Find the terminating line feed. }
    i := StrScan(@FBuffer, #10) - FBuffer;
    if i < 0 then
      { Incomplete line, so break out of loop. }
      Break;
    { Replace the carriage return (before the line feed) by a null character,
      terminating the line. }
    FBuffer[i - 1] := #0;



    { Process the incoming line. }
    ProcessLine(StrPas(FBuffer));



    { Restore the carriage return. }
    FBuffer[i - 1] := #13;
    { Was it the last line in the buffer? }
    if i >= FBufLen - 1 then
    begin
      FBufLen := 0;
      Break;
    end;
    { Not the last line, so move the data to the front of the buffer. }
    Move(FBuffer[i + 1], FBuffer, FBufLen - i);
    Dec(FBufLen, i + 1);
  end;
end;

{ Process the command. }
procedure TUser.ProcessLine(Line: String);
var
  Numeric: Integer;
  ToNick: String;
  i: Integer;
begin
  { Convert numeric to an integer. }
  try
    Numeric := StrToInt(Copy(Line, 1, 3));
    Line := Copy(Line, 5, 65535);
  except
    Send('500 ERROR Unknown command');
    Exit;
  end;
```

```
  i := Pos(' ', Line);
  if i = 0 then
  begin
    ToNick := Line;
    Line := '';
  end
  else
  begin
    ToNick := Copy(Line, 1, i - 1);
    Line := Copy(Line, i + 1, 65535);
  end;
  case Numeric of
    1:    Send(Format('002 %s %s', [ToNick, Line]));
    2:    begin
            FPing := GetTickCount - StrToIntDef(Line, 0);
            ListItem.SubItems[2] := IntToStr(FPing);
          end;
    3:    frmMain.Ping(Self, ToNick, Line);
    4:    frmMain.Pong(Self, ToNick, Line);
    10:   begin
            frmMain.UserNick(Self, ToNick, Line);
          end;
    100:  frmMain.SendMessage(Self, ToNick, Line);
  else
    Send('500 ERROR Unknown command');
  end;
end;

{ TfrmMain section ---------------------------------------------- }

{ Set some initial values. }
procedure TfrmMain.FormCreate(Sender: TObject);
var
  Ini: TIniFile;
  tmpstr: string;
begin
  dirApplication := ExtractFilePath(Application.ExeName);
  SetCaption(0);
  Ini := TIniFile.Create(dirApplication + 'nus.ini');
  try
    Left := Ini.ReadInteger('Window', 'Left', Left);
    Top := Ini.ReadInteger('Window', 'Top', Top);
    Width := Ini.ReadInteger('Window', 'Width', Width);
    Height := Ini.ReadInteger('Window', 'Height', Height);
    FPort := Ini.ReadString('Server', 'Port', FPort);

    tmpstr := Ini.ReadString('Simulator Options', 'Run Timed', 'disabled');
    if tmpstr = 'enabled' then RunTimed := True else RunTimed := False;
    RunDuration := Ini.ReadInteger('Simulator Options', 'Run Duration', 5);
    tmpstr := Ini.ReadString('Simulator Options', 'Run to Finish', 'enabled');
    if tmpstr = 'enabled' then Run2Finish := True else Run2Finish := False;
    Number_of_simulated_clients := Ini.ReadInteger('Simulator Options', 'Number of
simulated clients', 1);
    Script_filename := Ini.ReadString('Simulator Options', 'Script filename', '');
```

```
  except
    Ini.Free;
  end;

  FPort := '4000';
  sktMain.Addr := '0.0.0.0';
  sktMain.Port := FPort;
  FStarted := False;
  FShuttingDown := False;
  CurrentlyRunning := False;
  Finished := False;
  AllDataInBufferSent := False;
end;

{ If server is not listening, then start listening. }
procedure TfrmMain.FormShow(Sender: TObject);
begin
  if not FStarted then
  begin
    FStarted := True;
    sktMain.Listen;
  end;
  frmMain.WindowState := wsMaximized;
end;

{ Free up resources. }
procedure TfrmMain.FormClose(Sender: TObject; var Action: TCloseAction);
var
  Ini: TIniFile;
  i: Integer;
  tmpstr: String;
begin
  FShuttingDown := True;
  for i := 0 to lvwUsers.Items.Count - 1 do
    TUser(lvwUsers.Items[i].Data).Free;
  Ini := TIniFile.Create(dirApplication + 'nus.ini');
  try
    Ini.WriteInteger('Window', 'Left', Left);
    Ini.WriteInteger('Window', 'Top', Top);
    Ini.WriteInteger('Window', 'Width', Width);
    Ini.WriteInteger('Window', 'Height', Height);
    Ini.WriteString('Server', 'Port', FPort);

    if RunTimed = True then tmpstr := 'enabled' else tmpstr := 'disabled';
    Ini.WriteString('Simulator Options', 'Run Timed', tmpstr);
    if Run2Finish = True then tmpstr := 'enabled' else tmpstr := 'disabled';
    Ini.WriteString('Simulator Options', 'Run to Finish', tmpstr);
    Ini.WriteInteger('Simulator Options', 'Run Duration', RunDuration);
    Ini.WriteInteger('Simulator Options', 'Number of simulated clients',
Number_of_simulated_clients);
    Ini.WriteString('Simulator Options', 'Script filename', Script_filename);
  except
    Ini.Free;
  end;
```

```
end;

{ Exit the application. }
procedure TfrmMain.miServerExitClick(Sender: TObject);
begin
  Close;
end;

{ Show current status of toolbar and status bar. }
procedure TfrmMain.miViewClick(Sender: TObject);
begin
  miViewStatusbar.Checked := sbrMain.Visible;
end;

{ Toggle visibility of status bar. }
procedure TfrmMain.miViewStatusbarClick(Sender: TObject);
begin
  sbrMain.Visible := not sbrMain.Visible;
end;

{ Show some info about the program. }
procedure TfrmMain.miHelpAboutClick(Sender: TObject);
begin
  About.Show;
{ The following code will center the About dialogbox form }
  About.Left := (Screen.Width div 2) - (About.Width div 2);
  About.Top := (Screen.Height div 2) - (About.Height div 2);
end;


function TfrmMain.UserNick(User: TUser; Nick: String; Name: String): Boolean;
begin
  Result := False;
  { Probably just a change of case, so allow it. }
  if StrIComp(PChar(Nick), PChar(User.Nick)) = 0 then
  begin
    { Change listview caption. }
    User.Nick := Nick;
    User.ListItem.Caption := User.FName;
    { Send success message. }
    User.Send(Format('011 %s', [Nick]));
    Exit;
  end;
  { See if anyone else already has the nick. }
  Result := FindNick(Nick) = nil;
  if Result then
  begin
    { Change listview caption. }
    User.Nick := Nick;
    User.FName := Name;
    User.ListItem.Caption := User.FName;
    { Send success message. }
    User.Send(Format('011 %s', [Nick]));
  end
```

```
  else
    { Someone else already has the nick. }
    User.Send(Format('511 %s Nick already in use', [Nick]));
end;


{ Send a message to the person. }
procedure TfrmMain.SendMessage(From: TUser; ToNick, Msg: String);
var
  U: TUser;
begin
  U := FindNick(ToNick);
  if U = nil then
    From.Send(Format('501 %s No such nick', [ToNick]))
  else
    U.Send(Format('100 %s %s', [From.Nick, Msg]));
end;

{ Send a ping to the person. }
procedure TfrmMain.Ping(From: TUser; ToNick, TickCount: String);
var
  U: TUser;
begin
  U := FindNick(ToNick);
  if U = nil then
    From.Send(Format('501 %s No such nick', [ToNick]))
  else
    U.Send(Format('003 %s %s', [From.Nick, TickCount]));
end;

{ Send a pong to the person. }
procedure TfrmMain.Pong(From: TUser; ToNick, TickCount: String);
var
  U: TUser;
begin
  U := FindNick(ToNick);
  if U = nil then
    From.Send(Format('501 %s No such nick', [ToNick]))
  else
    U.Send(Format('004 %s %s', [From.Nick, TickCount]));
end;


{ Find a user with the specified IP address }
function TfrmMain.FindNick(Nick: String): TUser;
var
  i: Integer;
begin
  Result := nil;
  for i := 0 to lvwUsers.Items.Count - 1 do
    if StrIComp(PChar(TUser(lvwUsers.Items[i].Data).Nick), PChar(Nick)) = 0 then
    begin
      Result := TUser(lvwUsers.Items[i].Data);
      Exit;
```

```
    end;
end;

{ A client is connecting. }
procedure TfrmMain.sktMainSessionAvailable(Sender: TObject; Error: Word);
var
  User: TUser;
begin
  { Create a new user object to handle the connection. }
  User := TUser.Create(TWSocket(Sender).Accept);
  { Add to the list. }
  User.ListItem := lvwUsers.Items.Add;
  with User.ListItem do
  begin
    Caption := '<unknown>';
    SubItems.Add(User.Address);
    SubItems.Add('n/a');
    SubItems.Add('0');
    Data := User;
  end;
end;

{ Update caption to show number of users connected. }
procedure TfrmMain.lvwUsersInsert(Sender: TObject; Item: TListItem);
begin
  SetCaption(lvwUsers.Items.Count);
end;

{ Update caption to show number of users connected. }
procedure TfrmMain.lvwUsersDeletion(Sender: TObject; Item: TListItem);
begin
  { The user is about to be deleted, but not actually gone yet, so adjust for it. }
  SetCaption(lvwUsers.Items.Count - 1);
end;

{ Set caption and application title. }
procedure TfrmMain.SetCaption(Count: Integer);
begin
  Caption := Format('%d users - %s %s', [Count, AppName, AppVersion]);
  Application.Title := Caption;
end;

{ Received a message that a client quit.  So we free that client's TUser
  object. }
procedure TfrmMain.WMClientQuit(var Msg: TMessage);
var
  User: TUser;
begin
  if not FShuttingDown then
  begin
    User := TUser(Msg.lParam);
    if User <> nil then
    begin
      User.ListItem.Delete;
```

```
      User.Free;
    end;
  end;
end;

procedure TfrmMain.miViewOptionsClick(Sender: TObject);
begin
  Options_dialogbox.Show;
{ The following code will center the Options dialogbox form }
  Options_dialogbox.Left := (Screen.Width  div 2) - (Options_dialogbox.Width div 2);
  Options_dialogbox.Top := (Screen.Height div 2) - (Options_dialogbox.Height div 2);
end;

{ Start the simulation }
procedure TfrmMain.SEND1Click(Sender: TObject);
var
   U: TUser;
   i: Integer;
   Ini: TIniFile;

   n, m, o: Integer;
   tmpstr,tmpstr2: string;
   y,z: TDateTime;
   dirApplication: string;

   StartTime, CurrentTime: TDateTime;
   TestString: String;
   TotalDataSent, TotalDataReceived: Integer;
   Throughput: Real;

begin
if CurrentlyRunning then
   begin
        Finished := True;
        sbrMain.SimpleText := 'SIMULATION CANCELLED BY USER - Please wait while
freeing up used resources';
   end
else
 begin
   CurrentlyRunning := True;

   { change the menus to reflect the fact that currently a simulation is in progress }
   SEND1.Caption := '[Cancel_Simulation]';
   miHelp.Enabled := False;
   miReport.Enabled := False;
   miView.Enabled := False;
   miServer.Enabled := False;

   try
      i := lvwUsers.ItemFocused.Index;
      U := TUser(lvwUsers.Items[i].Data);
    except
      on EAccessViolation do
      begin
```

```
           MessageDlg(#13 + 'Please select at least one client before executing
process', mtInformation, [mbOk], 0);
           CurrentlyRunning := False;
           SEND1.Caption := '&Run';
           miHelp.Enabled := True;
           miReport.Enabled := True;
           miView.Enabled := True;
           miServer.Enabled := True;
      end;
    end;

   Counter := 0;
   y := 0;    { to store previous second, and to be compared with
CurrentTime/GetTickCount }
   z := 0;    { to track Progress Bar's ... well, progress }
   n := 0;    { store number of timing_records already completed }
   m := 1;    { store number of transactions already completed }
   o := 0;    { store number of send routine already completed }
   TotalDataSent     := 0;
   TotalDataReceived := 0;

   { Prepare the Progress Bar }
   ProgressBar1.Min := 0;
   ProgressBar1.Max := RunDuration * 10;     { it will be updated every 1/10th
second }
   ProgressBar1.Step := 1;
   Finished := False;
   { enable the StatusBar specific for Run2Finish simulation }
   if (Run2Finish) then
   begin
        Run2FinishStatBar.Enabled := True;
        Run2FinishStatBar.Visible := True;
   end;

   { Get starting time }
   StartTime := GetTickCount;
   { Get parameters from the script file }
   Ini := TIniFile.Create(Script_filename);
   try
          timing_record            := Ini.ReadInteger('NUS_simulator_script',
'timing_record', 1);
          delay_before_start       := Ini.ReadInteger('NUS_simulator_script',
'delay_before_start', 0);
          delay_before_transaction := Ini.ReadInteger('NUS_simulator_script',
'delay_before_transaction', 0);
          transaction_per_record   := Ini.ReadInteger('NUS_simulator_script',
'transaction_per_record', 10);
          number_of_send           := Ini.ReadInteger('NUS_simulator_script',
'number_of_send', 1);
          delay_after_transaction  := Ini.ReadInteger('NUS_simulator_script',
'delay_after_transaction', 0);
          tmpstr := Ini.ReadString('NUS_simulator_script',
'disconnect_after_transaction', 'false');
          if (tmpstr = 'true') then disconnect_after_transaction := True else
```

```
disconnect_after_transaction := False;
   except
   end;
   { Prepare the arrays to store results }
   SetLength(time_elapsed, timing_record);
   SetLength(bytes_transferred, timing_record);
   SetLength(throughput_stat, timing_record);
   SetLength(delay_before_sending, number_of_send);
   SetLength(send_data, number_of_send);
   SetLength(client_send, number_of_send);
   SetLength(client_delay, number_of_send);

   { clear previous' simulation run results }
   repeat
         time_elapsed[n] := 0;
         bytes_transferred[n] := 0;
         throughput_stat[n] := 0;
         inc(n);
   until (n >= timing_record);
   n := 0;

   repeat   { read every send routine's details, and put it into the arrays }
         Str(o+1, tmpstr);
         delay_before_sending[o] := Ini.ReadInteger('NUS_simulator_script',
'delay_before_sending'+tmpstr, 0);
         send_data[o]            := Ini.ReadInteger('NUS_simulator_script',
'send_data'+tmpstr, 100);
         client_send[o]          := Ini.ReadInteger('NUS_simulator_script',
'instruct_client_to_send'+tmpstr, 100);
         client_delay[o]         := Ini.ReadInteger('NUS_simulator_script',
'instruct_client_to_delay'+tmpstr, 0);
         inc(o);
   until (o >= number_of_send);


   { --- Go! --- }
   Delay(delay_before_start);
   repeat   { loop for timing_records }
         Delay(delay_before_transaction);
         m := 0;
         start_timing := GetTickCount;
         repeat { loop for transactions }
                o := 0;
                repeat { loop for each send routine }
                       Delay(delay_before_sending[o]);
                       { send data to client }
                       Str(o, tmpstr);
                       TestString := DupChars('X', send_data[o]);
                       frmMain.SendMessage(U, U.Nick, 'Data: ' + tmpstr + TestString);
                       { instruct client to send certain size of data packet & delay }
                       U.Send(Format('120 %d %d %d', [1, client_send[o],
client_delay[o]]));

                       { for user information - Increase Progress Bar every 1/10
```

```
second }
                        if (CurrentTime > z + 100) and (RunTimed) then
                        begin
                             z := CurrentTime;
                             ProgressBar1.Stepit;
                        end;
                        { for user information - Inform user how long we have been
running this simulation }
                        Str(Counter, tmpstr);
                        sbrMain.SimpleText := 'Time Elapsed: ' + tmpstr + ' seconds';
                        { for user information - Increase Counter every 1 second }
                        CurrentTime := GetTickCount;
                        if (CurrentTime > y + 1000) then
                        begin
                             y := CurrentTime;
                             if (Counter < RunDuration) and (RunTimed) then
inc(Counter);
                             if (Run2Finish) then inc(Counter);
                        end;

                        inc(TotalDataSent, send_data[o]);
                        inc(TotalDataReceived, client_send[o]);
                        if (RunTimed) then
                           bytes_transferred[Counter] := bytes_transferred[Counter] +
(send_data[o] + client_send[o])
                        else
                           bytes_transferred[n] := bytes_transferred[n] + (send_data[o]
+ client_send[o]);
                        inc(o);
                until (o >= number_of_send) or (Finished);
                inc(m);
         until  (m >= transaction_per_record) or (Finished);
         Delay(delay_after_transaction);

         if disconnect_after_transaction = True then disconnect;
         Application.ProcessMessages;

         { get statistics for this timing record }
         end_timing := GetTickCount;
         time_elapsed[n] := end_timing - start_timing;
         { time_elapsed is in miliseconds, we need to convert it to second }
         throughput_stat[n] := bytes_transferred[n] / (time_elapsed[n] / 1000);

         { give user the prediction of how long this simulation will run }
         Run2FinishStatBar.SimpleText := 'Time Elapsed: ' + tmpstr + ' seconds';

         { check for simulation's end }
         if (RunTimed = True) then
         begin
              if (CurrentTime >= StartTime + (RunDuration * 1000)) then Finished :=
True;
         end;
         if (Run2Finish = True) then inc(n);
   until (n >= timing_record) or (Finished);
```

```
   CurrentlyRunning := False;
   SEND1.Caption := '&Run';
   SEND1.Enabled := False;

   { report result to user }
   ProgressBar1.Position := RunDuration * 10;
   if (RunTimed = True) then Throughput := (TotalDataSent + TotalDataReceived) /
RunDuration;
   if (Run2Finish = True) then
   begin
        n := 0;
        m := 0;
        repeat
              Throughput := Throughput + bytes_transferred[o];
              m          := m + time_elapsed[o];
              inc(n);
        until (n >= timing_record);
        { m is in milliseconds,
          so in the end we need to multiply by 1000 to get the Throughput in seconds }
        Throughput := (Throughput / m) * 1000;
        { disable again the StatusBar specific for Run2Finish simulation }
        Run2FinishStatBar.Enabled := True;
        Run2FinishStatBar.Visible := True;
   end;
   { multiply by 8 to get results in Mbps (Mega bit per second) }
   Str((Throughput * 8):15:2, tmpstr); Str(((Throughput /  1048576) *8):6:2, tmpstr2);
   MessageDlg('Simulation finished,' + #13'Throughput: ' + tmpstr + ' bytes/second
(' +  tmpstr2 + ' Mbps)', mtInformation, [mbOk], 0);

   U.ListItem.SubItems.Delete(1);
   { workaround for ListItem bug }
   if (U.FirstTimeAccess) then
   begin
        U.FirstTimeAccess := False;
        U.ListItem.SubItems.Delete(1);
   end;

   U.ListItem.SubItems.Add('Network throughput: ' + tmpstr + ' bytes/second   (' +
tmpstr2 + ' Mbps)');

   { clean up }
   ProgressBar1.Position := 0;
   sbrMain.SimpleText := '';

   { will instantly stop data transmission, without disconnecting the client }
   frmNotify1.Show;
   frmNotify1.Left := (Screen.Width  div 2) - (frmNotify1.Width div 2);
   frmNotify1.Top := (Screen.Height div 2) - (frmNotify1.Height div 2);
   U.FSocket.DeleteBufferedData; { <------------------------ }
   frmNotify1.Close;

   { re-enable all menu items }
   SEND1.Enabled := True;
   miHelp.Enabled := True;
```

```
   miReport.Enabled := True;
   miView.Enabled := True;
   miServer.Enabled := True;
 end;
end;

{ Extracts the next token from the line.  Returns the remaining line. }
function TfrmMain.GetNextToken(var Line: String): String;
var
  i: Integer;
begin
  { Find first space character. }
  i := Pos(' ', Line);
  { No space? }
  if i = 0 then
  begin
    Result := Line;
    Line := '';
  end
  else
  { Just extract the first token. }
  begin
    Result := Copy(Line, 1, i - 1);
    Line := Copy(Line, i + 1, 65535);
  end;
end;

function TfrmMain.DupChars(WhatChar: Char; HowMany: Integer): String;
var
   x: Integer;
   TmpString: String;
begin
   x := 1;
   repeat
         TmpString := TmpString + WhatChar;
         inc(x);
   until x > HowMany;
   Result := TmpString;
end;

procedure TfrmMain.Disconnect();
var U: TUser;
begin
    U.FSocket.DeleteBufferedData;
end;

{ the delay (WaitFor) is in miliseconds }
procedure TfrmMain.Delay(WaitFor: Integer);
var
   StartTime, CurrentTime: Integer;
begin
     StartTime := GetTickCount;
     repeat
           CurrentTime := GetTickCount;
```

```
     until (CurrentTime >= StartTime);
end;

procedure TfrmMain.sktMainDataSent(Sender: TObject; Error: Word);
begin
     AllDataInBufferSent := True;
end;

procedure TfrmMain.Chart1Click(Sender: TObject);
begin
   ChartReport.Show;
   ChartReport.Left := (Screen.Width  div 2) - (ChartReport.Width div 2);
   ChartReport.Top := (Screen.Height div 2) - (ChartReport.Height div 2);
end;

end.
```

ABOUT_DIALOGBOX.PAS

```
unit About_dialogbox;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TAbout = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    procedure CreateParams(var Params: TCreateParams); override;
  end;

var
  About: TAbout;

implementation

uses main_form;
{$R *.DFM}
```

```
procedure TAbout.Createparams(var Params: TCreateParams);
begin
     inherited CreateParams(Params);
     with Params do
          Style := (Style or WS_POPUP) and (not WS_DLGFRAME);
end;

procedure TAbout.Button1Click(Sender: TObject);
begin
     About.Close;
     frmMain.Show;
end;

end.
```

OPTIONS.PAS

```
unit Options;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TOptions_dialogbox = class(TForm)
    Button1: TButton;
    GroupBox1: TGroupBox;
    RunToFinish: TRadioButton;
    RunFor: TRadioButton;
    Edit1: TEdit;
    Label1: TLabel;
    Edit2: TEdit;
    Label2: TLabel;
    OpenDialog1: TOpenDialog;
    Edit3: TEdit;
    Label3: TLabel;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure RunToFinishClick(Sender: TObject);
    procedure RunForClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
```

```
  end;

var
  Options_dialogbox: TOptions_dialogbox;

implementation

uses main_form;

{$R *.DFM}


procedure TOptions_dialogbox.Button1Click(Sender: TObject);
var
   tmpint: integer;
begin
     Val(Edit1.Text, RunDuration, tmpint);
     Script_filename := Edit2.Text;
     Val(Edit3.Text, Number_of_simulated_clients, tmpint);

     if RunFor.Checked = True then
     begin
          Run2Finish := False;
          RunTimed := True;
     end
     else
     begin
          Run2Finish := True;
          RunTimed := False;
     end;

     Options_dialogbox.Close;
     frmMain.Show;
end;


procedure TOptions_dialogbox.RunToFinishClick(Sender: TObject);
begin
     Edit1.Enabled := False;
     RunFor.Checked := False;
end;

procedure TOptions_dialogbox.RunForClick(Sender: TObject);
begin
     Edit1.Enabled := True;
     RunToFinish.Checked := False;
end;

procedure TOptions_dialogbox.FormShow(Sender: TObject);
var
   tmpstr: string;
begin
     Str(RunDuration, tmpstr);
     Edit1.Text := tmpstr;
```

```
     Edit2.Text := Script_filename;
     Str(Number_of_simulated_clients, tmpstr);
     Edit3.Text := tmpstr;

     if Run2Finish then
     begin
          RunToFinish.Checked := True;
          RunFor.Checked := False;
     end
     else
     begin
          RunFor.Checked := True;
          RunToFinish.Checked := False;
     end;
end;

procedure TOptions_dialogbox.Button2Click(Sender: TObject);
begin
     if OpenDialog1.Execute then
     begin
          Edit2.Text := OpenDialog1.FileName;
     end;
end;


end.
```

REPORT1_CHART.PAS

```
unit Report1_chart;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, TeEngine, Series, ExtCtrls, TeeProcs, Chart;

type
  TChartReport = class(TForm)
    Chart1: TChart;
    Series1: TLineSeries;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
```

```
    { Private declarations }
  public
    { Public declarations }
  end;

var
  ChartReport: TChartReport;

implementation

uses main_form;

{$R *.DFM}

procedure TChartReport.Button1Click(Sender: TObject);
begin
     ChartReport.Close;
     frmMain.Show;
end;

procedure TChartReport.FormShow(Sender: TObject);
var
   n: Integer;
   tmpint: integer;
   tmpreal: real;
begin
     n := 0;
     Series1.Clear;

     If (RunTimed) then
     begin
          repeat
           { multiply by 8 to get results in Mbps (Mega bit per second) }
           Series1.AddY((bytes_transferred[n] / 1048576) * 8,' ',clTeeColor );
           inc(n);
          until (n >= Counter+1);
     end
     else
     begin
          Series1.AddY(0,' ',clTeeColor );
          repeat
           { multiply by 8 to get results in Mbps (Mega bit per second) }
           Series1.AddY((throughput_stat[n] / 1048576) * 8,' ',clTeeColor );
           inc(n);
          until (n >= timing_record);
     end;
end;

procedure TChartReport.Button2Click(Sender: TObject);
begin
        Chart1.PrintLandscape;
        ChartReport.Close;
        frmMain.Show;
end;
```

```
end.
```

9.3.2 CLIENT source code

```
unit main_form;

{

 SIMULATOR client code for NUS project
 http://welcome.to/nus_project

 ------------------------------------------------------

 Code based on MiniChat software version 1.0 beta by Steve Williams
 Utilise TWsocket component by F. Piette

 ------------------------------------------------------

}

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ComCtrls, StdCtrls, IniFiles, WSocket, ExtCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    WorkstationName: TEdit;
    Label2: TLabel;
    ServerIPaddress: TEdit;
    Button1: TButton;
    StatusBar1: TStatusBar;
    WorkstationIPaddress: TEdit;
    Label3: TLabel;
    sktMain: TWSocket;
    Button2: TButton;
    Timer1: TTimer;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure sktMainDnsLookupDone(Sender: TObject; Error: Word);
    procedure sktMainDataAvailable(Sender: TObject; Error: Word);
    procedure sktMainSessionConnected(Sender: TObject; Error: Word);
    procedure sktMainSessionClosed(Sender: TObject; Error: Word);
    procedure Timer1Timer(Sender: TObject);
  private
    FNick: String;               { workstation's IP address }
    FName: String;               { workstation's name }
    FServer: String;
    FPort: String;
    FBuffer: array [0..8191] of Char;
    FBufLen: Integer;
    FListing: Boolean;
```

```
    FConnected: Boolean;
    FNickSet: Boolean;
    FStatus: TForm;
  public
        procedure Send(Line: String);
        procedure ProcessLine(Line: String);
        function GetNextToken(var Line: String): String;
  end;

var
  Form1: TForm1;
  dirApplication: String;

const
  AppName = 'Network Application Simulator - client';
  AppVersion = '1.0 beta';

implementation
{$R *.DFM}


{ Set some initial values. }
procedure TForm1.FormShow(Sender: TObject);
var
  Ini: TIniFile;
begin
  Application.Title := 'Network Application Simulator - Client';

  Caption := Format('%s %s', [AppName, AppVersion]);
  Application.Title := Caption;
  dirApplication := ExtractFilePath(Application.ExeName);
  { Read profile from ini file. }
  Ini := TIniFile.Create(dirApplication + 'nus.ini');
  try
    Left := Ini.ReadInteger('Window', 'Left', Left);
    Top := Ini.ReadInteger('Window', 'Top', Top);
    Width := Ini.ReadInteger('Window', 'Width', Width);
    Height := Ini.ReadInteger('Window', 'Height', Height);

    FNick := Ini.ReadString('Profile', 'Nick', FNick);
    FName := Ini.ReadString('Profile', 'Name', FName);
    FServer := Ini.ReadString('Profile', 'Server', FServer);
    FPort := Ini.ReadString('Profile', 'Port', FPort);
  finally
    Ini.Free;
  end;

  WorkstationName.Text := FName;
  WorkstationIPaddress.Text := FNick;
  ServerIPaddress.Text := FServer;

  FPort := '4000';
  FBufLen := 0;
  FListing := False;
```

```
  FConnected := False;
  FStatus := nil;
  { The following code will center the form ! }
    Form1.Left := (Screen.Width  div 2) - (Form1.Width div 2);
    Form1.Top := (Screen.Height div 2) - (Form1.Height div 2);
end;


{ Connect to server. }
procedure TForm1.Button1Click(Sender: TObject);
var
  Ini: TIniFile;
begin
  StatusBar1.SimpleText := '';

  FName := WorkstationName.Text;
  FNick := WorkstationIPaddress.Text;
  FServer := ServerIPaddress.Text;

  { Set FNickSet to False because we haven't secured the nick yet. }
  FNickSet := False;
  { Check for any null values. }
  if (FNick = '') or (FServer = '') or (FPort = '') then
  begin
    StatusBar1.SimpleText := 'ERROR: Null values not allowed in Workstation name or
Server IP address';
    Exit;
  end;
  StatusBar1.SimpleText := Format('>>> Connecting to %s', [FServer]);
  Ini := TIniFile.Create(dirApplication + 'nus.ini');
  try
    Ini.WriteInteger('Window', 'Left', Left);
    Ini.WriteInteger('Window', 'Top', Top);
    Ini.WriteInteger('Window', 'Width', Width);
    Ini.WriteInteger('Window', 'Height', Height);

    Ini.WriteString('Profile', 'Nick', FNick);
    Ini.WriteString('Profile', 'Name', FName);
    Ini.WriteString('Profile', 'Server', FServer);
    Ini.WriteString('Profile', 'Port', FPort);
  finally
    Ini.Free;
  end;
  { Don't actually connect yet, but do a DNS lookup.  The connection will be
    made when the DNS lookup returns. }
  sktMain.Port := '4000';
  sktMain.DNSLookup(FServer);
  Timer1.Enabled := True;   { start timer for automatic reconnect on lost connection }
end;


{ The DNS lookup is done. If successful, then do the connect. }
procedure TForm1.sktMainDnsLookupDone(Sender: TObject; Error: Word);
begin
```

```
  { Check for errors. }
  if Error <> 0 then
  begin
    { Show message in status bar. }
    StatusBar1.SimpleText := 'ERROR: Could not resolve host';
    Exit;
  end;
  { Transfer the IP address from DNSResult to Addr. }
  sktMain.Addr := sktMain.DnsResult;
  sktMain.Port := FPort;
  { Initiate the connection. }
  sktMain.Connect;
end;


{ The connection was established. }
procedure TForm1.sktMainSessionConnected(Sender: TObject; Error: Word);
begin
  if Error <> 0 then
    StatusBar1.SimpleText := '*** Could not connect to server'
  else
  begin
    FConnected := True;
    StatusBar1.SimpleText := '*** Connected';
    Application.minimize;
    { Register ourself to the server }
    Send(Format('010 %s %s', [FNick, FName]));
  end;
end;


{ Send some data to the server. }
procedure TForm1.Send(Line: String);
begin
  if FConnected then
    sktMain.SendStr(Line + #13#10)
  else
    StatusBar1.SimpleText := '*** Not connected to server';
end;


{ Process the command. }
procedure TForm1.ProcessLine(Line: String);
var
  Numeric: Integer;
  FromNick: String;
begin
  { Convert numeric to an integer. }
  try
    Numeric := StrToInt(GetNextToken(Line));
  except
    Exit;
  end;
  FromNick := GetNextToken(Line);
```

```
  case Numeric of
    1:
          begin
            { Automatically send reply. }
            Send(Format('002 %s %s', [FNick, Line]));
            StatusBar1.SimpleText := '*** Received server ping';
          end;
    2:    StatusBar1.SimpleText := Format('*** Ping response from server: %d ms',
[GetTickCount - StrToIntDef(Line, 0)]);
    3:
          begin
            Send(Format('004 %s %s', [FromNick, Line]));
            StatusBar1.SimpleText := Format('*** Received ping from %s', [FromNick]);
          end;
    4:    StatusBar1.SimpleText := Format('*** Ping response from %s: %d ms',
[FromNick, GetTickCount - StrToIntDef(Line, 0)]);
    11:
          begin
            StatusBar1.SimpleText := Format('*** Your IP address is %s', [FromNick]);
            FNick := FromNick;
          end;
    100:  StatusBar1.SimpleText := Format('%s> %s', [FromNick, Line]);
  else
    StatusBar1.SimpleText := Format('%d %s %s', [Numeric, FromNick, Line]);
  end;
end;


{ Extracts the next token from the line.  Returns the remaining line. }
function TForm1.GetNextToken(var Line: String): String;
var
  i: Integer;
begin
  { Find first space character. }
  i := Pos(' ', Line);
  { No space? }
  if i = 0 then
  begin
    Result := Line;
    Line := '';
  end
  else
  { Just extract the first token. }
  begin
    Result := Copy(Line, 1, i - 1);
    Line := Copy(Line, i + 1, 65535);
  end;
end;


{ Received some data.  This could consist of one line, multiple lines, or
  sections of lines.  Based on code from the TWSChat example in ICS. }
procedure TForm1.sktMainDataAvailable(Sender: TObject; Error: Word);
var
```

```
  Len, i: Integer;
begin
  { Retrieve the data, and place it at the end of the buffer. }
  Len := sktMain.Receive(@FBuffer[FBufLen], SizeOf(FBuffer) - FBufLen - 1);
  if Len <= 0 then
    Exit;
  { Add the data length to the buffer count. }
  Inc(FBufLen, Len);
  { Place a null byte at the end of the buffer. }
  FBuffer[FBufLen] := #0;
  { Scan the buffer for complete lines. }
  while True do
  begin
    { Find the terminating line feed. }
    i := StrScan(@FBuffer, #10) - FBuffer;
    if i < 0 then
      { Incomplete line, so break out of loop. }
      Break;
    { Replace the carriage return (before the line feed) by a null character,
      terminating the line. }
    FBuffer[i - 1] := #0;
    { Process the line. }
    ProcessLine(StrPas(FBuffer));
    { Restore the carriage return. }
    FBuffer[i - 1] := #13;
    { Was it the last line in the buffer? }
    if i >= FBufLen - 1 then
    begin
      FBufLen := 0;
      Break;
    end;
    { Not the last line, so move the data to the front of the buffer. }
    Move(FBuffer[i + 1], FBuffer, FBufLen - i);
    Dec(FBufLen, i + 1);
  end;
end;


{ Disconnect from server & quit application }
procedure TForm1.Button2Click(Sender: TObject);
begin
  { Cancel any DNS lookup that is happening. }
  sktMain.CancelDNSLookup;
  { Close the socket. }
  sktMain.Close;
  { Exit from application }
  Close;
end;

{ The connection was closed. }
procedure TForm1.sktMainSessionClosed(Sender: TObject; Error: Word);
begin
  FConnected := False;
  { Show message in display. }
```

```
  StatusBar1.SimpleText := '*** Disconnected';
  { Set caption and application title to normal. }
  Caption := Format('%s %s', [AppName, AppVersion]);
  Application.Title := Caption;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
     if (sktMain.State = wsClosed) and (Fserver <> '') then
     begin
          sktMain.Port := '4000';
          sktMain.DNSLookup(FServer);
     end;
end;


end.
```

9.4 NUS Basecode - Source code

NUS basecode is the code on which all other projects are built on. Refer to section 7.1.1 for more background information on it.

9.4.1 SERVER source code

MAIN.PAS

```
unit Main;
{

 SERVER base code for NUS project
 http://welcome.to/nus_project

 ----------------------------------------------------------

 Code based on MiniChat software version 1.0 beta by Steve Williams
 Utilise TWsocket component by F. Piette

 ----------------------------------------------------------

 30Oct99:
 # Finished building the base code

}


interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ComCtrls, Menus, WSocket, StdCtrls, Winsock, ExtCtrls, About_dialogbox, Options;

const
  WM_CLIENTQUIT = WM_USER + 10;

type
  TUser = class(TObject)
  protected
    FBuffer: array [0..1023] of Char;
    FBufLen: Integer;
    FSocket: TWSocket;
    FName: String;              { Name of the workstation }
    FNick: String;              { client's IP address, as reported by the client
itself }
    FAddress: String;           { client's IP address as well, from
TWSocket.GetPeerAddr}
    FPing: Integer;
```

```
  procedure SocketSessionClosed(Sender: TObject; Error: Word);
  procedure SocketDataAvailable(Sender: TObject; Error: Word);
  procedure Send(Line: String);
  procedure ProcessLine(Line: String);
public
  ListItem: TListItem;
  constructor Create(ASocket: TSocket);
  destructor Destroy; override;
  property Nick: String read FNick write FNick;
  property Address: String read FAddress write FAddress;
end;

TfrmMain = class(TForm)
  mnuMain: TMainMenu;
  miServer: TMenuItem;
  miServerExit: TMenuItem;
  sbrMain: TStatusBar;
  miView: TMenuItem;
  miHelp: TMenuItem;
  miViewToolbar: TMenuItem;
  miViewStatusbar: TMenuItem;
  miViewSep1: TMenuItem;
  miViewOptions: TMenuItem;
  miHelpTopics: TMenuItem;
  miHelpSep1: TMenuItem;
  miHelpAbout: TMenuItem;
  sktMain: TWSocket;
  lvwUsers: TListView;
  SEND1: TMenuItem;
  procedure FormCreate(Sender: TObject);
  procedure FormClose(Sender: TObject; var Action: TCloseAction);
  procedure miServerExitClick(Sender: TObject);
  procedure miViewClick(Sender: TObject);
  procedure miViewStatusbarClick(Sender: TObject);
  procedure miHelpAboutClick(Sender: TObject);
  procedure sktMainSessionAvailable(Sender: TObject; Error: Word);
  procedure FormShow(Sender: TObject);
  procedure lvwUsersInsert(Sender: TObject; Item: TListItem);
  procedure lvwUsersDeletion(Sender: TObject; Item: TListItem);
  procedure miViewOptionsClick(Sender: TObject);
  procedure SEND1Click(Sender: TObject);
private
  FStarted: Boolean;
  FPort: String;
  FShuttingDown: Boolean;
  procedure WMClientQuit(var Msg: TMessage); message WM_CLIENTQUIT;
public
  function  UserNick(User: TUser; Nick: String; Name: String): Boolean;
  procedure SendMessage(From: TUser; ToNick, Msg: String);
  procedure Ping(From: TUser; ToNick, TickCount: String);
  procedure Pong(From: TUser; ToNick, TickCount: String);
  function FindNick(Nick: String): TUser;
  procedure SetCaption(Count: Integer);
  function GetNextToken(var Line: String): String;
```

```
  end;


var
  frmMain  : TfrmMain;


implementation

{$R *.DFM}

uses
  IniFiles;

const
  AppName = 'Network Application Simulator - Server';
  AppVersion = '1.0 beta';

var
  dirApplication: String;

{ TUser list section ------------------------------------- }

{ Create a client socket and assign it some values. }
constructor TUser.Create(ASocket: TSocket);
begin
  inherited Create;
  FSocket := TWSocket.Create(nil);
  FSocket.HSocket := ASocket;
  FSocket.Tag := Integer(Self);
  FSocket.OnDataAvailable := SocketDataAvailable;
  FSocket.OnSessionClosed := SocketSessionClosed;
  FBufLen := 0;
  FPing := 0;
  FAddress := FSocket.GetPeerAddr;
  { Ping client immediately. }
  Send(Format('001 server %d', [GetTickCount]));
end;

{ Close the socket and free it. }
destructor TUser.Destroy;
begin
  FSocket.Abort;
  FSocket.Free;
  inherited Destroy;
end;

{ Send data to the client. }
procedure TUser.Send(Line: String);
begin
  FSocket.SendStr(Line + #13#10);
end;

{ The connection was closed. }
```

```
procedure TUser.SocketSessionClosed(Sender: TObject; Error: Word);
begin
  { Invoke clean-up routine by sending WM_CLIENTQUIT message
    to frmMain. The clean-up routine can be seen in frmMain.WMClientQuit procedure }
  PostMessage(frmMain.Handle, WM_CLIENTQUIT, 0, Integer(Self));
end;

{ Received some data. }
procedure TUser.SocketDataAvailable(Sender: TObject; Error: Word);
var
  Len, i: Integer;
begin
  { Retrieve the data, and place it at the end of the buffer. }
  Len := TWSocket(Sender).Receive(@FBuffer[FBufLen], SizeOf(FBuffer) - FBufLen - 1);
  if Len <= 0 then
    Exit;
  { Add the data length to the buffer count. }
  Inc(FBufLen, Len);
  { Place a null byte at the end of the buffer. }
  FBuffer[FBufLen] := #0;
  { Scan the buffer for complete lines. }
  while True do
  begin
    { Find the terminating line feed. }
    i := StrScan(@FBuffer, #10) - FBuffer;
    if i < 0 then
      { Incomplete line, so break out of loop. }
      Break;
    { Replace the carriage return (before the line feed) by a null character,
      terminating the line. }
    FBuffer[i - 1] := #0;



    { Process the incoming line. }
    ProcessLine(StrPas(FBuffer));



    { Restore the carriage return. }
    FBuffer[i - 1] := #13;
    { Was it the last line in the buffer? }
    if i >= FBufLen - 1 then
    begin
      FBufLen := 0;
      Break;
    end;
    { Not the last line, so move the data to the front of the buffer. }
    Move(FBuffer[i + 1], FBuffer, FBufLen - i);
    Dec(FBufLen, i + 1);
  end;
end;

{ Process the command. }
```

```
procedure TUser.ProcessLine(Line: String);
var
  Numeric: Integer;
  ToNick: String;
  i: Integer;
begin
  { Convert numeric to an integer. }
  try
    Numeric := StrToInt(Copy(Line, 1, 3));
    Line := Copy(Line, 5, 65535);
  except
    Send('500 ERROR Unknown command');
    Exit;
  end;
  i := Pos(' ', Line);
  if i = 0 then
  begin
    ToNick := Line;
    Line := '';
  end
  else
  begin
    ToNick := Copy(Line, 1, i - 1);
    Line := Copy(Line, i + 1, 65535);
  end;
  case Numeric of
    1:    Send(Format('002 %s %s', [ToNick, Line]));
    2:    begin
            FPing := GetTickCount - StrToIntDef(Line, 0);
            ListItem.SubItems[2] := IntToStr(FPing);
          end;
    3:    frmMain.Ping(Self, ToNick, Line);
    4:    frmMain.Pong(Self, ToNick, Line);
    10:   begin
            frmMain.UserNick(Self, ToNick, Line);
          end;
    100:  frmMain.SendMessage(Self, ToNick, Line);
  else
    Send('500 ERROR Unknown command');
  end;
end;

{ TfrmMain section ------------------------------------------- }

{ Set some initial values. }
procedure TfrmMain.FormCreate(Sender: TObject);
var
  Ini: TIniFile;
begin
  dirApplication := ExtractFilePath(Application.ExeName);
  SetCaption(0);
  Ini := TIniFile.Create(dirApplication + 'simulator.ini');
  try
    Left := Ini.ReadInteger('Window', 'Left', Left);
```

```
    Top := Ini.ReadInteger('Window', 'Top', Top);
    Width := Ini.ReadInteger('Window', 'Width', Width);
    Height := Ini.ReadInteger('Window', 'Height', Height);
    FPort := Ini.ReadString('Server', 'Port', FPort);
  except
    Ini.Free;
  end;

  FPort := '4000';
  sktMain.Addr := '0.0.0.0';
  sktMain.Port := FPort;
  FStarted := False;
  FShuttingDown := False;
end;

{ If server is not listening, then start listening. }
procedure TfrmMain.FormShow(Sender: TObject);
begin
  if not FStarted then
  begin
    FStarted := True;
    sktMain.Listen;
  end;
  frmMain.WindowState := wsMaximized;
end;

{ Free up resources. }
procedure TfrmMain.FormClose(Sender: TObject; var Action: TCloseAction);
var
  Ini: TIniFile;
  i: Integer;
begin
  FShuttingDown := True;
  for i := 0 to lvwUsers.Items.Count - 1 do
    TUser(lvwUsers.Items[i].Data).Free;
  Ini := TIniFile.Create(dirApplication + 'simulator.ini');
  try
    Ini.WriteInteger('Window', 'Left', Left);
    Ini.WriteInteger('Window', 'Top', Top);
    Ini.WriteInteger('Window', 'Width', Width);
    Ini.WriteInteger('Window', 'Height', Height);
    Ini.WriteString('Server', 'Port', FPort);
  except
    Ini.Free;
  end;
end;

{ Exit the application. }
procedure TfrmMain.miServerExitClick(Sender: TObject);
begin
  Close;
end;

{ Show current status of toolbar and status bar. }
```

```
procedure TfrmMain.miViewClick(Sender: TObject);
begin
  miViewStatusbar.Checked := sbrMain.Visible;
end;

{ Toggle visibility of status bar. }
procedure TfrmMain.miViewStatusbarClick(Sender: TObject);
begin
  sbrMain.Visible := not sbrMain.Visible;
end;

{ Show some info about the program. }
procedure TfrmMain.miHelpAboutClick(Sender: TObject);
begin
  About.Show;
{ The following code will center the About dialogbox form }
  About.Left := (Screen.Width div 2) - (About.Width div 2);
  About.Top := (Screen.Height div 2) - (About.Height div 2);
end;


{ Workstation registration handling routine }
function TfrmMain.UserNick(User: TUser; Nick: String; Name: String): Boolean;
begin
  Result := False;
  if StrIComp(PChar(Nick), PChar(User.Nick)) = 0 then
  begin
    { Change listview caption. }
    User.Nick := Nick;
    User.ListItem.Caption := User.FName;
    { Send success message. }
    User.Send(Format('011 %s', [Nick]));
    Exit;
  end;
  { See if anyone else already has the IP address -> fatal }
  Result := FindNick(Nick) = nil;
  if Result then
  begin
    { Change listview caption. }
    User.Nick := Nick;
    User.FName := Name;
    User.ListItem.Caption := User.FName;
    { Send success message. }
    User.Send(Format('011 %s', [Nick]));
  end
  else
    { Someone else already has the IP address }
    User.Send(Format('511 %s IP address already in used!', [Nick]));
end;


{ Send a message to the workstation. }
procedure TfrmMain.SendMessage(From: TUser; ToNick, Msg: String);
var
```

```
  U: TUser;
begin
  U := FindNick(ToNick);
  if U = nil then
    From.Send(Format('501 %s No such nick', [ToNick]))
  else
    U.Send(Format('100 %s %s', [From.Nick, Msg]));
end;

{ Send a ping to the workstation }
procedure TfrmMain.Ping(From: TUser; ToNick, TickCount: String);
var
  U: TUser;
begin
  U := FindNick(ToNick);
  if U = nil then
    From.Send(Format('501 %s No such nick', [ToNick]))
  else
    U.Send(Format('003 %s %s', [From.Nick, TickCount]));
end;

{ Send a pong to the workstation }
procedure TfrmMain.Pong(From: TUser; ToNick, TickCount: String);
var
  U: TUser;
begin
  U := FindNick(ToNick);
  if U = nil then
    From.Send(Format('501 %s No such nick', [ToNick]))
  else
    U.Send(Format('004 %s %s', [From.Nick, TickCount]));
end;


{ Find a workstation with the specified IP address }
function TfrmMain.FindNick(Nick: String): TUser;
var
  i: Integer;
begin
  Result := nil;
  for i := 0 to lvwUsers.Items.Count - 1 do
    if StrIComp(PChar(TUser(lvwUsers.Items[i].Data).Nick), PChar(Nick)) = 0 then
    begin
      Result := TUser(lvwUsers.Items[i].Data);
      Exit;
    end;
end;

{ A client is connecting. }
procedure TfrmMain.sktMainSessionAvailable(Sender: TObject; Error: Word);
var
  User: TUser;
begin
  { Create a new user object to handle the connection. }
```

```
  User := TUser.Create(TWSocket(Sender).Accept);
  { Add to the list. }
  User.ListItem := lvwUsers.Items.Add;
  with User.ListItem do
  begin
    Caption := '<unknown>';
    SubItems.Add(User.Address);
    SubItems.Add('n/a');
    SubItems.Add('0');
    Data := User;
  end;
end;

{ Update caption to show number of clients connected. }
procedure TfrmMain.lvwUsersInsert(Sender: TObject; Item: TListItem);
begin
  SetCaption(lvwUsers.Items.Count);
end;

{ Update caption to show number of clients connected. }
procedure TfrmMain.lvwUsersDeletion(Sender: TObject; Item: TListItem);
begin
  { The client is about to deleted, but not actually gone yet, so adjust for it. }
  SetCaption(lvwUsers.Items.Count - 1);
end;

{ Set caption and application title. }
procedure TfrmMain.SetCaption(Count: Integer);
begin
  Caption := Format('%d clients - %s %s', [Count, AppName, AppVersion]);
  Application.Title := Caption;
end;

{ Received a message that a client quit.  So we free that client's TUser
  object. }
procedure TfrmMain.WMClientQuit(var Msg: TMessage);
var
  User: TUser;
begin
  if not FShuttingDown then
  begin
    User := TUser(Msg.lParam);
    if User <> nil then
    begin
      User.ListItem.Delete;
      User.Free;
    end;
  end;
end;

procedure TfrmMain.miViewOptionsClick(Sender: TObject);
begin
  Options_dialogbox.Show;
{ The following code will center the Options dialogbox form }
```

```
  Options_dialogbox.Left := (Screen.Width  div 2) - (Options_dialogbox.Width div 2);
  Options_dialogbox.Top := (Screen.Height div 2) - (Options_dialogbox.Height div 2);
end;

{ Provided to make it easy to test network connection between server - client }
procedure TfrmMain.SEND1Click(Sender: TObject);
var
   U: TUser;
   i: Integer;
begin
   i := lvwUsers.ItemFocused.Index;
   U := TUser(lvwUsers.Items[i].Data);
   frmMain.SendMessage(U, U.Nick, 'Data transmission from server');
end;

{ Extracts the next token from the line.  Returns the remaining line. }
function TfrmMain.GetNextToken(var Line: String): String;
var
  i: Integer;
begin
  { Find first space character. }
  i := Pos(' ', Line);
  { No space? }
  if i = 0 then
  begin
    Result := Line;
    Line := '';
  end
  else
  { Just extract the first token. }
  begin
    Result := Copy(Line, 1, i - 1);
    Line := Copy(Line, i + 1, 65535);
  end;
end;


end.
```

ABOUT_DIALOGBOX.PAS

```
unit About_dialogbox;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
```

```
type
  TAbout = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    procedure CreateParams(var Params: TCreateParams); override;
  end;

var
  About: TAbout;

implementation

uses Main;
{$R *.DFM}


procedure TAbout.Createparams(var Params: TCreateParams);
begin
     inherited CreateParams(Params);
     with Params do
          Style := (Style or WS_POPUP) and (not WS_DLGFRAME);
end;

procedure TAbout.Button1Click(Sender: TObject);
begin
     About.Close;
     frmMain.Show;
end;

end.
```

OPTIONS.PAS

```
unit Options;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
```

```
  TOptions_dialogbox = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public

  end;

var
  Options_dialogbox: TOptions_dialogbox;

implementation

uses Main;

{$R *.DFM}


procedure TOptions_dialogbox.Button1Click(Sender: TObject);
begin
     Options_dialogbox.Close;
     frmMain.Show;
end;


end.
```

9.4.2 CLIENT source code

```
unit main_form;

{

 CLIENT base code for NUS project
 http://welcome.to/nus_project

 ------------------------------------------------------

 TO-DO LIST:

# Find out how to obtain workstation's IP address
--> done, use TWSocket.GetPeerAddr

 ------------------------------------------------------

 30Oct99:
 # Finished building the base code.

}

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ComCtrls, StdCtrls, IniFiles, WSocket;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    WorkstationName: TEdit;
    Label2: TLabel;
    ServerIPaddress: TEdit;
    Button1: TButton;
    StatusBar1: TStatusBar;
    WorkstationIPaddress: TEdit;
    Label3: TLabel;
    sktMain: TWSocket;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure sktMainDnsLookupDone(Sender: TObject; Error: Word);
    procedure sktMainDataAvailable(Sender: TObject; Error: Word);
    procedure sktMainSessionConnected(Sender: TObject; Error: Word);
    procedure sktMainSessionClosed(Sender: TObject; Error: Word);
  private
    FNick: String;               { workstation's IP address }
    FName: String;               { workstation's name }
    FServer: String;
    FPort: String;
```

```
    FBuffer: array [0..8191] of Char;
    FBufLen: Integer;
    FListing: Boolean;
    FConnected: Boolean;
    FNickSet: Boolean;
    FStatus: TForm;
  public
        procedure Send(Line: String);
        procedure ProcessLine(Line: String);
        function GetNextToken(var Line: String): String;
  end;

var
  Form1: TForm1;
  dirApplication: String;

const
  AppName = 'Simulator - client';
  AppVersion = '1.0 beta';

implementation
{$R *.DFM}


{ Set some initial values. }
procedure TForm1.FormShow(Sender: TObject);
var
  Ini: TIniFile;
begin
  Application.Title := 'Network Application Simulator - Client';

  Caption := Format('%s %s', [AppName, AppVersion]);
  Application.Title := Caption;
  dirApplication := ExtractFilePath(Application.ExeName);
  { Read profile from ini file. }
  Ini := TIniFile.Create(dirApplication + 'sim_clnt.ini');
  try
    Left := Ini.ReadInteger('Window', 'Left', Left);
    Top := Ini.ReadInteger('Window', 'Top', Top);
    Width := Ini.ReadInteger('Window', 'Width', Width);
    Height := Ini.ReadInteger('Window', 'Height', Height);

    FNick := Ini.ReadString('Profile', 'Nick', FNick);
    FName := Ini.ReadString('Profile', 'Name', FName);
    FServer := Ini.ReadString('Profile', 'Server', FServer);
    FPort := Ini.ReadString('Profile', 'Port', FPort);
  finally
    Ini.Free;
  end;

  WorkstationName.Text := FName;
  WorkstationIPaddress.Text := FNick;
  ServerIPaddress.Text := FServer;
```

```
  FPort := '4000';
  FBufLen := 0;
  FListing := False;
  FConnected := False;
  FStatus := nil;
  { The following code will center the form ! }
    Form1.Left := (Screen.Width  div 2) - (Form1.Width div 2);
    Form1.Top := (Screen.Height div 2) - (Form1.Height div 2);
end;


{ Connect to server. }
procedure TForm1.Button1Click(Sender: TObject);
var
  Ini: TIniFile;
begin
  application.minimize;
  StatusBar1.SimpleText := '';

  FName := WorkstationName.Text;
  FNick := WorkstationIPaddress.Text;
  FServer := ServerIPaddress.Text;

  { Set FNickSet to False because we haven't secured the nick yet. }
  FNickSet := False;
  { Check for any null values. }
  if (FNick = '') or (FServer = '') or (FPort = '') then
  begin
    StatusBar1.SimpleText := 'ERROR: Null values not allowed in Workstation name or
Server IP address';
    Exit;
  end;
  StatusBar1.SimpleText := Format('>>> Connecting to %s', [FServer]);
  Ini := TIniFile.Create(dirApplication + 'sim_clnt.ini');
  try
    Ini.WriteInteger('Window', 'Left', Left);
    Ini.WriteInteger('Window', 'Top', Top);
    Ini.WriteInteger('Window', 'Width', Width);
    Ini.WriteInteger('Window', 'Height', Height);

    Ini.WriteString('Profile', 'Nick', FNick);
    Ini.WriteString('Profile', 'Name', FName);
    Ini.WriteString('Profile', 'Server', FServer);
    Ini.WriteString('Profile', 'Port', FPort);
  finally
    Ini.Free;
  end;
  { Don't actually connect yet, but do a DNS lookup.  The connection will be
    made when the DNS lookup returns. }
  sktMain.Port := '4000';
  sktMain.DNSLookup(FServer);
end;
```

```
{ The DNS lookup is done. If successful, then do the connect. }
procedure TForm1.sktMainDnsLookupDone(Sender: TObject; Error: Word);
begin
  { Check for errors. }
  if Error <> 0 then
  begin
    { Show message in status bar. }
    StatusBar1.SimpleText := 'ERROR: Could not resolve host';
    Exit;
  end;
  { Transfer the IP address from DNSResult to Addr. }
  sktMain.Addr := sktMain.DnsResult;
  sktMain.Port := FPort;
  { Initiate the connection. }
  sktMain.Connect;
end;


{ The connection was established. }
procedure TForm1.sktMainSessionConnected(Sender: TObject; Error: Word);
begin
  if Error <> 0 then
    StatusBar1.SimpleText := '*** Could not connect to server'
  else
  begin
    FConnected := True;
    StatusBar1.SimpleText := '*** Connected';
    { Register ourself to the server }
    Send(Format('010 %s %s', [FNick, FName]));
  end;
end;


{ Send some data to the server. }
procedure TForm1.Send(Line: String);
begin
  if FConnected then
    sktMain.SendStr(Line + #13#10)
  else
    StatusBar1.SimpleText := '*** Not connected to server';
end;


{ Process the command. }
procedure TForm1.ProcessLine(Line: String);
var
  Numeric: Integer;
  FromNick: String;
begin
  { Convert numeric to an integer. }
  try
    Numeric := StrToInt(GetNextToken(Line));
  except
    Exit;
```

```
  end;
  FromNick := GetNextToken(Line);
  case Numeric of
    1:
          begin
            { Automatically send reply. }
            Send(Format('002 %s %s', [FNick, Line]));
            StatusBar1.SimpleText := '*** Received server ping';
          end;
    2:    StatusBar1.SimpleText := Format('*** Ping response from server: %d ms',
[GetTickCount - StrToIntDef(Line, 0)]);
    3:
          begin
            Send(Format('004 %s %s', [FromNick, Line]));
            StatusBar1.SimpleText := Format('*** Received ping from %s', [FromNick]);
          end;
    4:    StatusBar1.SimpleText := Format('*** Ping response from %s: %d ms',
[FromNick, GetTickCount - StrToIntDef(Line, 0)]);
    11:
          begin
            StatusBar1.SimpleText := Format('*** Your IP address is %s', [FromNick]);
            FNick := FromNick;
          end;
    100:  StatusBar1.SimpleText := Format('%s> %s', [FromNick, Line]);
  else
    StatusBar1.SimpleText := Format('%d %s %s', [Numeric, FromNick, Line]);
  end;
end;


{ Extracts the next token from the line.  Returns the remaining line. }
function TForm1.GetNextToken(var Line: String): String;
var
  i: Integer;
begin
  { Find first space character. }
  i := Pos(' ', Line);
  { No space? }
  if i = 0 then
  begin
    Result := Line;
    Line := '';
  end
  else
  { Just extract the first token. }
  begin
    Result := Copy(Line, 1, i - 1);
    Line := Copy(Line, i + 1, 65535);
  end;
end;


{ Received some data.  This could consist of one line, multiple lines, or
  sections of lines.  Based on code from the TWSChat example in ICS. }
```

```
procedure TForm1.sktMainDataAvailable(Sender: TObject; Error: Word);
var
  Len, i: Integer;
begin
  { Retrieve the data, and place it at the end of the buffer. }
  Len := sktMain.Receive(@FBuffer[FBufLen], SizeOf(FBuffer) - FBufLen - 1);
  if Len <= 0 then
    Exit;
  { Add the data length to the buffer count. }
  Inc(FBufLen, Len);
  { Place a null byte at the end of the buffer. }
  FBuffer[FBufLen] := #0;
  { Scan the buffer for complete lines. }
  while True do
  begin
    { Find the terminating line feed. }
    i := StrScan(@FBuffer, #10) - FBuffer;
    if i < 0 then
      { Incomplete line, so break out of loop. }
      Break;
    { Replace the carriage return (before the line feed) by a null character,
      terminating the line. }
    FBuffer[i - 1] := #0;
    { Process the line. }
    ProcessLine(StrPas(FBuffer));
    { Restore the carriage return. }
    FBuffer[i - 1] := #13;
    { Was it the last line in the buffer? }
    if i >= FBufLen - 1 then
    begin
      FBufLen := 0;
      Break;
    end;
    { Not the last line, so move the data to the front of the buffer. }
    Move(FBuffer[i + 1], FBuffer, FBufLen - i);
    Dec(FBufLen, i + 1);
  end;
end;


{ Disconnect from server & quit application }
procedure TForm1.Button2Click(Sender: TObject);
begin
  { Cancel any DNS lookup that is happening. }
  sktMain.CancelDNSLookup;
  { Close the socket. }
  sktMain.Close;
  { Exit from application }
  Close;
end;

{ The connection was closed. }
procedure TForm1.sktMainSessionClosed(Sender: TObject; Error: Word);
begin
```

```
  FConnected := False;
  { Show message in display. }
  StatusBar1.SimpleText := '*** Disconnected';
  { Set caption and application title to normal. }
  Caption := Format('%s %s', [AppName, AppVersion]);
  Application.Title := Caption;
end;

end.
```

9.5 CD containing the latest build of NUS software

Submitted together with the dissertation.