TELECOMMUNICATION ENGINEER

Diploma Thesis

DESIGN AND IMPLEMENTATION AN ONLINE LAB EXERCISES FOR TEACHING THE DEVELOPMENT OF CLIENT USING SOCKET PROGRAMMING IN C

Najlae Ben Doudouh

July 2015

Supervisor: Prof. Spyros Denazis

Electrical and Computer Engineering Department
Mr. Spyros Denazis  
Professor at Electrical and Computer Engineering Department  
University of Patras, as director of diploma thesis of  
Ms. Najlae Ben Doudouh

Reports:

That the present work, entitled:

DESIGN AND IMPLEMENTATION AN ONLINE LAB EXERCISES FOR TEACHING THE DEVELOPMENT OF CLIENT USING SOCKET PROGRAMMING IN C

It has been made and written by the mentioned student under our direction and authorize her presentation date:

Patras, 06 of July 2015

Signed:
DECLARATION

I hereby certify that this diploma thesis is entirely the result of my own work and I have faithfully and properly cited all sources used in the thesis.

***********************

Najlae Ben Doudouh
Acknowledgments

The completion of the final project represents one of those moments in life where it's worth to look back and take stock of all the past. Not only means the fruit of several months of development, but also symbolizes the conclusion of a career through many years of dedication, sacrifice and hard work.

Once I get here, I can say I am very proud of this achievement and I feel full membership on this special and important profession as telecommunication engineer.

Thus, I would like to thanks my supervisors Prof. Sypros Denazis and Mr. Kostas Lampropoulos for their support over time and for their feedback provided in the annual meetings. Throughout this thesis they help me by sharing their knowledge, their competence, their availability and, most important, their friendship and their confidence in me.

I would also take this opportunity to express the profound gratitude from my deep heart to my beloved parents, grandparents, and my siblings for their love and continuous support – both spiritually and materially.

Finally, I would like to thank my dear, Mounaim Rafai. He was always there cheering me up and stood by me through the good times and bad.
Abstract

Learning by practice is essential to achieve better performance in complex areas. Nevertheless, learning is only effective if the students have at their disposal a wide range of exercises that cover the entire course syllabus and if their resolutions are quickly be evaluated and given the appropriate feedback.

Currently the teaching-learning process in complex domains, such as computer programming, is characterized by an extensive education program and a high enrolment of students. This poses a great workload for faculty and teaching assistants responsible for the creation, delivering and assessment of student exercises.

The main goal of this thesis is to foster practice-based learning in complex domains. This objective is attained with an e-learning platform - called FORGE - as a conceptual tool to organize and facilitate technical interoperability among systems and services in domains that use complex evaluation. These domains has a diversity of tools, include interactive activities combining simulations, virtual experiments and more, from the environments, students are able to edit the solutions of exercises proposed by their teacher, to compile and to execute them.

One of the unique features of this environment is the possibility of developing solutions in an incremental way by marking and unmarking which parts of the solution are due to the students and which parts should made use of the hidden teacher’s solution. In this way, students can test the program and can better understand what it is expected from them. They can also check parts of the code without waiting for the complete solution. The tool can also automatically check the validity of some exercises by means of tests, providing feedback on the attempts of students, in order to enhance interactive learning based on constructivism theory, and allow for students and teachers to learn skills for intelligent use of information and technological communication, not forgetting the authoring, management and sequencing of exercises.
Table of Contents

1. INTRODUCTION ................................................................................................................... 13
   1.1. Justification ................................................................................................................. 13
   1.2. Motivation ................................................................................................................... 13
   1.3. Objectives .................................................................................................................... 14
2. STATE OF THE ART ............................................................................................................... 15
   2.1. Introduction ................................................................................................................ 15
   2.2. E-learning in today ...................................................................................................... 16
   2.3. The future of E-learning .............................................................................................. 17
   2.4. E-learning platforms .................................................................................................... 17
   2.5. Comparison between E-learning platforms. ............................................................... 18
      2.5.1. ATutor: ............................................................................................................... 18
      2.5.2. Moodle: ............................................................................................................ 19
      2.5.3. Claroline: ......................................................................................................... 19
      2.5.4. Dokeos: .......................................................................................................... 20
      2.5.5. Sakai: ............................................................................................................. 21
      2.5.6. ILIAS: ............................................................................................................ 22
      2.5.7. Platform used: FORGE ................................................................................. 23
3. PREFACE .............................................................................................................................. 29
4. SOCKET PROGRAMMING ..................................................................................................... 30
   4.1. What is a socket? ........................................................................................................ 30
   4.2. Client-Server model ..................................................................................................... 30
   4.3. Connectionless (UDP) Vs Connection-Oriented (TCP) Servers .................................... 32
   4.4. Definition of Socket ..................................................................................................... 33
   4.5. The Socket Interface .................................................................................................... 33
   4.6. System Data Structures for Sockets ............................................................................ 35
   4.7. Active vs Passive Sockets ............................................................................................. 35
   4.8. Socket Endpoints ......................................................................................................... 35
   4.9. Creating socket ............................................................................................................ 37
   4.10. Generic socket addresses ........................................................................................ 38
5. TCP SOCKET PROGRAMMING .............................................................................................. 40
   5.1. TCP Client .................................................................................................................... 40
   5.2. TCP Server ................................................................................................................... 41
6. UDP SOCKET PROGRAMMING ............................................................................................. 44
7. APIs FOR MANAGING NAMES AND IP ................................................................. 46
   7.1. Data Type struct hostent ................................................................. 46
   7.2. Network Integers versus Host Integers .............................................. 48
   7.3. Conversion of IP numbers ............................................................... 50
8. DESIGN .............................................................................................................. 51
9. IMPLEMENTATION ............................................................................................. 57
   9.1. jFed ............................................................................................................ 57
       9.1.1. Register for an account ............................................................... 57
       9.1.2. Start up jFed ............................................................................... 58
       9.1.3. Run jFed ...................................................................................... 58
       9.1.4. Create the experiment ................................................................. 59
       9.1.5. RSpec editor ............................................................................... 61
       9.1.6. Run the experiment ................................................................. 63
       9.1.7. SSH widget ............................................................................. 64
   9.2. Lab exercise .............................................................................................. 65
10. CONCLUSION AND FUTURE WORKS ............................................................ 71
APPENDIX ............................................................................................................... 72
   A. BACKGROUNDS ..................................................................................... 72
       1. TCP vs UDP .................................................................................. 74
       2. Port numbers and services ......................................................... 75
       3. Numeric IP Addresses ................................................................. 75
       4. IP Address Classes ................................................................... 76
       5. Host Names and DNS ................................................................. 77
   B. EXERCISES ................................................................................................. 78
       1. Server.c ...................................................................................... 78
       2. Client1.c ...................................................................................... 84
       3. Client2.c ...................................................................................... 88
       4. Client3.c ...................................................................................... 92
   C. RSpec ....................................................................................................... 96
   D. API REFERENCES ..................................................................................... 103

BIBLIOGRAPHY ................................................................................................. 111
1. INTRODUCTION

This project aims to introduce an online FORGE lab course learning to facilitate the work of teaching the development of a client using socket programming in C.

1.1. Justification

As project of help to the teaching arises the idea of creating a course of e-learning on socket programming in C concerning client–server model, thus complementing the different classes related to networks.

The design of the course will be eminently practical, focused on a rapid and direct learning of every theoretical part and supporting its comprehension from the exercises.

The purpose of the project will provide to students an e-learning tool easy to use, to facilitate their work without compromising the needs of teaching, giving the first all facilities to adapt according to their needs or preferences course development that serves as an introduction to use socket and that feels a sufficiently solid base to improve learning in the future.

1.2. Motivation

Academic motivation:

The e-learning platforms are starting to be considered and taking them in account. We are in a world where distances are becoming smaller thanks to the mass media.

It is true that there have been many predictions that distance education replace traditional classes, yet this is something that has not happened. The reasons are unclear, perhaps it is the inherent trust and confidence that must exist between people, the need to relate, or improvisation of a good teacher, his anecdotes, his words and his help motivate students in a way irreplaceable by any other means.
Despite this e-learning, it is a useful tool in many situations, and allows direct control of the used time. The criticism of educators is common to the time spent to students in working a course. With these tools able to control in a practical and simple form doing their exercises that required or proposed, for example.

Despite this I think that the best asset that has this technology is not the control that the teacher can exercise on the students, but the extra motivation that students find to study a subject distantly and being able to do some activities at any time, acceding easily to multimedia content, and especially including chats and forums to share views about the subject, with greater involvement entailed.

In short, it is an excellent tool to complement the classes in person and can serve as a motivator for students.

1.3. Objectives

The objective is to create an interactive lab course on a platform of e-learning using FORGE as a tool to complement or supplement the learning of a subject on client-server socket programming in C: client side. Additionally, offering an application or series of practical exercises to improve students understanding of the subject.

- Understand the principle of client / server communication.
- Study fundamental functions of socket.
- Implement a client / server application (using sockets):
  1- Connected mode: use of the TCP protocol.
- Develop a server processing requests from several clients.
2. STATE OF THE ART

2.1. Introduction

Nowadays, E-learning is not a new word to the internet community, especially to students. There are different subjects with different type of E-learning’s methods available through the internet, computer-network, and software for PC or television. For a real-time interactive E-learning system using the internet web-based application, one of the most commonly used technologies which can be realized is the client server architecture.

For a normal client server web application, the client sends a request for a resource located on the server, and the server application responds the required resource to the client. But in this E-learning client server application, there is a high interactivity between the client and the server application. The user does not only send a normal request, but he sends the request together with his proposal solution for an exercise. And there is not only one standard solution but there are different proposal solutions of different user for one exercise. The server application has to diagnose all these different solutions and send an appropriate response for each different proposal solution. And from the responses, each user will have to decide what they want to do next differently. This means that the client and the server application have an understanding between them, they can “communicate”, “talk” and “understand” each other like human being in which the client is like a student and the server application is like a tutor. One can say that the server application is an artificial intelligent application. This is what high interactivity means for and it makes the E-learning client server application be different from other normal request response client server application.

We will begin studying the different e-learning platforms in the market, with preference for those that are partially or completely free of use.
2.2. E-learning in today

Usually it is a system that consists of three parts: the students, the server with the theoretical and practical course, and finally one or more teachers to support the evolution of the course.

It is used in many contexts, the most common university education and academic training courses distantly from the university, as much public and internal access to the formation within the company as well.

Within the university, using two different methods are proposed. One is a complement to traditional classes, offering the theory on a platform to ensure reliable access to the complement of exercises to perform certain distantly. The other method is as a substitute for attending classes, by a teacher / course manager and supplying assistance by forums or other methods of internal consultation.

Nowadays the e-learning is in clear growth. There are many companies that offer courses of all kinds (mainly of topics derived from computer science such as programming languages, web design, etc., but also of many other classes, such as cooking classes or music...), thanks to the facility of access that they provide and to multiple tools to moderate and to direct the course, although this includes a large number of students. The possibilities are there to be exploited, yet the acceptance of the courses is given timidly in heavy subjects of major academic qualifications.

Using the example of the University of Patras, one of the most compromised to new technologies, we observe that it gets to intermediate commitments as for the possibilities of online courses. Stating that this University is leading of new trends in education, especially in terms of new technologies, we see that have developed their own programs and applications to accommodate contents, sometimes with subjects forums, but in a way little centralized and excessively dependent on the wishes of the teacher. Is absolutely commendable attitude of the University in this regard, especially in light of other cases in other universities, but always can be improved.

The e-learning divides the educators into two groups, some in favor and others against. It is clear that online learning has great educational value, offering the possibility of dividing into parts and subsections content of a subject and being able to read or perform the exercises proposed at anytime from anywhere. But many educators who prefer an attended class against to some online lessons.
2.3. The future of E-learning

We can guess where e-learning evolve. First, all possible elements of Web 2.0 will be adopted: Social networking, multimedia content, blogs, etc. The possible integration with any element is indispensable that will be a jump forward in terms of educational opportunities. Concepts such as collaborative web, semantic web, take form when it comes to e-learning. It is not expected that online classes leave aside classes, personal attention needed beyond what can be offered on the web, but it will become increasingly common integration and alternation between both worlds.

2.4. E-learning platforms

An e-learning platform is a software application that integrates different management tools, communication, evaluation, monitoring, etc... with the aim of providing technological support to teachers and students to optimize the various phases of the teaching-learning process, either the educational process completely remote, classroom or mixed nature and combine both modes in different proportions.

Main Features of the e-learning platforms:

- Authentication
- Generating content
- Viewing content
- Different media with a teacher / tutor
- Carrying out activities such as tasks, group work
- Report of the activities undertaken by the pupil
- Evaluation tools

Depending on the functionalities and the goals of the e-learning platform we can classify them as:

- CMS: Content Management System

The CMS system is the most basic functionality; is used for small projects where it is needed to generate the content within the system. Among the communication tools can be found forums, email and chat. As an example we
• **LMS: Learning Management System**

The LMS is a system that is focused precisely the area of education, allows control on both the contents and individual users who interact within it. The contents are created and loaded with some external authoring tool. It has most of the tools of communication and monitoring activities of users. For example, Moodle, ATutor...

• **LCMS: Learning Content Management System**

Integrates profits from previous systems, what gives greater robustness. Generally is a LMS to be added to the form or functionality to create within the content. In addition to managing the administrative functions of online learning, some systems also provide tools to deliver and manage instructor-led synchronous and asynchronous online training based on learning object methodology. These systems are called Learning content management systems or LCMSs. An LCMS provides tools for authoring and re-using or re-purposing content as well as virtual spaces for learner interaction such as discussion forums and live chat rooms.

### 2.5. Comparison between E-learning platforms.

#### 2.5.1. ATutor:

ATutor is a LCMS open source web-based, designed with accessibility and adaptability in mind. Administrators can install or update ATutor in minutes, create new designs to change the look, and easily extend its functionality with additional modules. Educators can collect, aggregate and distribute educational content quickly, and import content created and manage online courses. Students are offered a full adaptive learning environment. As for the evaluation activities, with ATutor can generate tests that include a broad kinds of questions (multiple choice, order, true / false, open response, multiple choice ...), plus it has a text editor quite full.
ATutor is based Open Source, which makes it an effective solution for both large and small organizations to provide training materials on the web, submit fully independent online courses.

2.5.2. Moodle:

Moodle is a course management system (LCMS) that has wide acceptance and popularity among educators around the world as a tool for creating dynamic websites for their students.

The aim of the Moodle project is always to offer the teacher the best tools to manage and promote learning. It has features that allow scaling to very large environments with hundreds or thousands of students at the same time that can be used with much smaller goals.

Many institutions use it as their primary platform for hosting full courses, while others use it as an addition to classroom courses. The large number of different modules (forums, wikis, etc.) allows you to create collaborative learning communities around a particular subject or matter while offering a way to give content to students and assess their progress through exercises.

Also there is ability to custom tasks that students must perform, and later send the resulting file or files to the web course, as we mentioned in ATutor. In addition there is the ability to include interactive activities with other programs such as Hot Potatoes or JClic. Finally, regarding the evaluation activities, Moodle allows conducting through the "Quizzes" section with a range greater than ATutor, since activities can also be made to fill gaps.

Since software is distributed under the terms of the GPL, it allows the development of modules and modifications by the user community. Because the community is so extensive, Moodle has a significant advantage over the other competitors.

2.5.3. Claroline:

This platform though not reaching the flexibility of Moodle, but is better prepared than ATutor in terms of the possibility of facing collaborative work.

Claroline is a platform for learning and virtual work (e-Learning and e-Working) open source and free software that enables trainers to build effective online courses and to manage learning and collaborative activities on the web.
Translated into 35 languages, Claroline has a large community of developers and users worldwide.

Released under an open source license, Claroline can create and manage courses to hundreds of organizations from 93 different countries and online collaboration spaces. Each course site provides a list of tools enabling the teacher to:

- Write the description of a course.
- Publish documents in any format (text, PDF, HTML, video ...).
- Administer public and private forums.
- Develop learning paths.
- Creating student groups.
- Prepare online exercises.
- Manage an agenda with tasks and deadlines.
- Publish announcements (also by e-mail).
- Propose tasks handled by the network.
- See the statistics of user activity.
- Use the wiki tool to write documents collaboratively.

Adaptable to various training contexts, Claroline is not only used by schools and universities, also in training centers, associations and companies. The platform is adaptable and offers a flexible and adjustable environment work.

It has been developed according to the needs and experiences of teachers. Claroline provides an intuitive and clear spaces administration interface. Management or daily management platform requires no special technical skill. The platform is quick to install and the use any web browser can handle the various parts of the course and the admission of users fluently.

2.5.4. Dokeos:

Dokeos is an e-learning environment and an application of administration of the course contents and also is a collaboration tool. It is free software and is licensed under the GNU GPL, the development is international and
collaborative. It is also certified by the OSI and can be used as a content management system (CMS) for education and educators. This feature to manage contents includes content distribution, calendar, training process, text chat, audio and video, test management and records kept.

Dokeos main goals are to be a flexible and easy to use with a user interface very friendly. Being a learning tool, especially recommended for users with minimal computer literacy aimed concern for content.

Dokeos is written in PHP and uses MySQL databases. The current stable version is Dokeos.

Dokeos.com is also a Belgian company that provides hosting support and e-learning services, apart from the distribution of the Dokeos platform. The company collaborates with the community paying several developers to them.

2.5.5. Sakai:

The Sakai Project is a free course management system (CMS) and open source. It contains a large number of software tools designed to help educators, researchers and students to collaborate online in support of their work, teaching courses, research projects or general partnership.

For courses, Sakai provides features to support and enhance learning and teaching. There are tools for collaboration that help organize communication and collaborative work in groups on campus and around the world. Using a web browser, users can create a site with Sakai tools to suit their needs, without having to know HTML.

However, the product vision reaches beyond teaching and learning applications. Sakai many facilities are included for research work and collaboration. For this there are modules as Sakaibrary, which allows linking those library resources to Sakai.

Sakai is open source; it is distributed freely licensed under the Educational Community License. The access code is extremely important for those who want to change their platform online campus or want to develop new tools. The quality of Open Source is important for all the Sakai community, since many changes proposed and created by the community can be adopted for the base project.
2.5.6. **ILIAS**:

ILIAS is a Learning Management System (LMS) Open Source for the development and implementation of e-learning web-based. The software was developed to reduce the costs of using new media in education and training, and to ensure the highest level of customer influence in the development of software. ILIAS is published under the GPL license and free.

It allows efficient creation of courses and course materials. It provides tools and templates for learning and integrated work process, including navigation and content management and users. Every user in ILIAS has his own personal desktop, which includes all the resources that are necessary for learning and helps meet the daily tasks of the course. The desk allows the aggregation of personal characteristics such as news, personal messages, Learning Resources, notes, bookmarks, feeds (RSS) and other information. The student has these blocks of information according to your needs.

ILIAS has multiple ways to deliver learning content. You can load multiple file types, and SCORM 2004, SCORM 1.2 and AICC standards are supported. ILIAS turn includes an authoring environment to create XML in which the learning modules, which can include images, flash, applets and other media, are based. You can create glossaries and reuse in other definitions learning modules.

ILIAS is available as open source software under the GNU General Public License (GPL) code and can be used without restrictions. Schools, universities, educational institutions and anyone interested can use the system for free and contribute to its further development. Users, authors and administrators require only a browser and Internet access. Multiple operating systems such as UNIX, Linux, Mac OS X and Windows can be used to ILIAS server with PHP and MySQL. All additional software (for example, MySQL, PHP, Apache) software is available as open source as well, and can be downloaded from their website for free.

Users around the world have already contributed to the development of programs coordinated by the ILIAS team of open source at University of Cologne.
2.5.7. Platform used: FORGE

Forging Online Education through FIRE (FORGE) is a project bringing the FIRE (Future Internet Research and Experimentation) and e-Learning worlds together. FORGE relates to communications and IT, as well as to other disciplines including the physical and social sciences. It will lead to a strong connection between the learning community and existing FIRE platforms and supporting tools.

![Figure 1: Forging online education through FIRE](image)

The overall objective of FORGE is to introduce the FIRE experimental facilities into the e-Learning community, in order to promote the concept of experimentally driven research in education by using experiments as an interactive learning and training channel for both students and professionals by raising the accessibility and usability of FIRE facilities. The goal is to create an open FORGE community and ecosystem where educational resources, collaborative tools and proposed experiments are offered and contributed for free.
FORGE follows the approach outlined in the previous figure to introduce the eLearning community to the FIRE experimental facilities and bridge the gap between these two areas. FORGE expects to create an environment for introducing the eLearning community to the experimentally driven research but also to act as a training facility for FIRE in general. For this, the FORGE core, which consists of the FORGE project partners, will initiate the process of developing a number of prototype courses of which the final attainment level will also target skills related to the basic and advanced usage of FIRE facilities for creating experiments (next to content specific learning goals of the course such as e.g. understanding a certain communication protocol). The creation of these courses provides insight in new requirements that will be imposed on FIRE facilities on one hand and on the eLearning community on the other. Additionally, these prototype lab courses, which will be created within the FORGE project, will (a) serve as exemplars for the educational community (so they can have a look at how these example courses are made when creating their own course) and (b) also enhance the access of educationalists to the FIRE facilities (as the example courses explain how to use FIRE facilities). FORGE will act as a conduit facilitating the passing of lessons learned to the FIRE facility owners enabling them to enhance their offerings to provide support for educational courses.

FIRE has invested significantly in the last few years in creating its federation. FIRE strives for a harmonization of tools and APIs to facilitate experimentation and the integration of heterogeneous resources around a single experiment.
under a single account and by using any experimentation tool that the end user wants. All this effort is currently led by FIRE’s flagship project, Fed4FIRE (Integrated project, number 318389, funded by the European Commission through the 7th ICT-Framework Programme, 1 Oct. 2012 – 30 Sep. 2016). FORGE will build upon Fed4FIRE’s leadership using the project as our main channel to the FIRE facilities. FORGE will thus adopt Fed4FIRE’s tools and proposed mechanisms to aid in our role as an intermediary between the learning community and the FIRE facilities and tools. The FORGE core will create a set of processes, tools and widgets to facilitate the development of lab courses as an interactive learning and training channel for both students and professionals.

FORGE uses three different technologies to achieve its goal:

1) Social networking technologies to: increase access to learning content; enable learners and educators to easily create and publish learning content; and support collaboration amongst learners. The Open University’s Social Learn environment enables educators and learners to create learning paths from online resources to share.

2) Interactive eBooks, where rich multimedia content is intertwined with interactive pages to set up and run large-scale internet experiments.

3) Linked data to support: i) the delivery of learning materials through the provision of easy-to-use navigation schemes; and ii) the discovery and recommendation of learning materials through semantic links. For example, the DISCOU tool, from The Open University (OU), automatically displays OU courses that are related to the content of web pages as they are browsed.

As we know the experimentation is a key concept in engineering education. However, physical experimentation is often expensive and hard to maintain, and requires specific guidance through the experiment in order to avoid malfunctions or injuries to the operator. Although simulators can in some situations complement physical experimentation, physical experiments are mandatory in most engineering education areas, allowing learners to fully understand design procedures. Physical experimentation and simulation can both contribute to engineering education and can both be integrated within the same computer-based platform. Remote laboratories can provide remote access to experiments and allow students to access experiments without time and location restrictions, providing the necessary guidance and constraining
operation in order to avoid setup integrity problems. Remote experiments can be ready all the time, and thus the remote laboratory concept provides a tool to sustain a learner-centric teaching approach. Remote network laboratories support shared learner access to physical network equipment through an internet interface. Depending on the interface used to access the hardware, the experience of configuring, maintaining, and troubleshooting a network environment is close to the experience in a true campus network. The interface that mediates access to the physical hardware significantly affects the quality of the learner experience. This platform indicated the interface may improve the accessibility of the labs for learners who have had little prior network experience. Additional software is often used to augment the environment through reporting services that provide a detailed analysis of network behavior. As aforementioned, these types of laboratories, employed to teach specific and general concepts, have been found more efficient than pure network analysis-based labs.

FORGE thus offers the following labs to the community:

- Hands-on Wireless LAN connectivity labs where students will be able to obtain practical skills in configuring these wireless networks and to gain better insights into the theory, such as the hidden terminal problem. These labs will allow students to acquire the learning outcomes from the “NC/Mobility” core-tier2 component of the CS2013 curricula.

- Studying Networking implications labs will enable learners to build a simulated Internet, either individually or in collaboration with other learners. These labs will provide the learning outcomes from the “NC/Local Area Networks” core-tier1 component.

- Network architecture labs will provide a simple interface to the OneLab environment, with all its experimentation tools available over the worldwide PlanetLab testbed. These labs will provide the learning outcomes from the ”NC/Introduction” core-tier1 component.

- A hands-on experience with flow control protocols and different traffic classes will enable students to understand how transport protocols such as TCP may impact the performance of the network, and how their control mechanisms react in the face of various network conditions. These labs will allow students to acquire the learning outcomes from the “NC/Resource Allocation” core-tier1 component.
• Hands-on experiences with application layer protocols and with how to provide reliable data communication will illustrate two fundamental concepts from a typical introduction to networking lecture, namely the evolution of HTTP and the concept of Reliable Data Transfer. These labs will address the ”NC/Networked Applications” core-tier1 and “NC/Reliable Data Delivery” core-tier2 components.

• FORGEBox architecture

The Figure provides a detailed view of FORGEBox proposed architecture. The figure displays also the concept of a FORGE repository that will host any shared published items such as Lab Courses, widgets and FIRE adapters to be used by the learning community and by other organizations hosting a FORGEBox instantiation. At its simplest form the core consists of services that make some tasks easier such as creating, managing and operating Lab Courses and their content as well as widgets and adapters. FORGEBox will contain a set of managing services, a Widgets layer, a FIRE adapters layer and a local repository of hosted Lab Courses.
FORGEBox managing services

A FORGEBox instantiation will contain a set of management services. A web control panel will be the mechanism to access and configure those services. Through these management services, FORGEBox admins will be able to perform the following tasks:
- User management. Create and manage accounts for users that are Lab course assistants, or simple users that want to use FORGE technology.
- Manage FORGEBox services that can be exposed and used by other users.
- Configure repository access. The admin can configure access points (i.e. URLs) that point to other FORGE repositories or even update and configure repositories content.
- FORGEBox updates. The admin can perform updates to any core services of FORGEBox.

Widgets layer

Widgets usually provide interactive web content to learners. This means that access should be provided by web containers and web services. Therefore the widget layer will not be a specific service, but rather an abstract concept, since widgets can be provided by several web containers. As web containers can be considered for example the Apache server or Tomcat. However, we expect that a local repository will keep track of the deployed widgets.

Fire adapters layer

As widgets layer is a concept, the same also holds for the FIRE adapters’ layer. FIRE adapters can be developed in any language or hosted by some kind of container. Still, a local repository needs to be in place, to keep track of what is deployed and what can be exposed as a FIRE adapter service.

Synergies widgets adapters

Synergies between FORGE widgets, adapters and FIRE resources: when a lab course is executed by a learner, either via a LMS web page or an iBook there are some underlying interactions that happen between different components. Especially between widgets, adapters and the course related FIRE resources. Widgets, as envisioned by FORGE, are going to be the main element towards the Learner for accessing and manipulating some FIRE resource parameters through web pages, LMSs and iBooks.
3. PREFACE

Internet and WWW have emerged as global ubiquitous media for communication and changed the way we conduct science, engineering, and commerce. They are also changing the way we learn, live, enjoy, communicate, interact, engage, etc. Our modern life activities are getting now completely centered around or driven by the Internet.

Today, computers all over the world are connected to the worldwide network known as the “Internet”. The Internet enables programs running in computers thousands of miles apart to communicate and exchange information. If we have a computer connected to a network, we may have used a Web browser; a typical program that makes use of the Internet. What does such a program do to communicate with others over a network? The answer varies depending on the application and the operating system, but many programs get access to network communication services through the "sockets" programming interface. The goal of this thesis is to learn how to build and design a client/server application that communicates using sockets.
4. SOCKET PROGRAMMING

4.1. What is a socket?

An interface between application and network which is used for communication between processes. Once configured the application can pass data to the socket for network transmission as well receive data from the socket (transmitted through the network by some other host).

To the kernel, a socket is an endpoint of communication. And to an application, a socket is a file descriptor that lets the application read/write from/to the network.

Clients and servers communicate with each by reading from and writing to socket descriptors.

Sockets come in different flavors, corresponding to different underlying protocol families and different stacks of protocols within a family. In this report we deal only with the TCP/IP protocol family. The main flavors of sockets in the TCP/IP family are stream sockets and datagram sockets. Stream sockets use TCP as the end-to-end protocol and thus provide a reliable byte-stream service. Datagram sockets use UDP and thus provide a best-effort datagram service that applications can use to send individual messages up to about 65500 bytes in length. Stream and datagram sockets are also supported by other protocol suites; however, in this thesis we deal only with TCP stream sockets and UDP datagram sockets.

Throughout this report, we will introduce some basic concepts and terminology concerning computer networks, network protocols, services, terms, etc... relevant to sockets and socket programming. More information will be explained in Appendix -A.

4.2. Client-Server model

Sockets are used for interprocess communication. Most of the interprocess communication follows a Client-Server Model, where client and server are two separate processes in itself. Server and Client exchange messages over the network through a common Socket API.
The terms client and server refer to these roles: The client program initiates communication, while the server program waits passively for and then responds to clients that contact it. Together, the client and server compose the application. The terms "client" and "server" are descriptive of the typical situation in which the server makes a particular capability; for example, a database service; available to any client that is able to communicate with it using the TCP/IP protocols.

Whether a program is acting as a client or server determines the general form of its use of the sockets API to communicate with its peer. (The client is the peer of the server and vice versa.) Beyond that, the client-server distinction is important because the client needs to know the server's address and port initially, but not vice versa. With the sockets API, the server can, if necessary, learn the client's address information when it receives the initial communication from the client. This is analogous to a telephone call; the called in general does not need to know the telephone number of the caller.

How does a client find out a server's IP address and port number? Usually, the client knows the name of the server it wants, for example, from a Universal Resource Locator (URL) and uses the name resolution service to learn the corresponding Internet address.

Finding a server's port number is a different story. In principle, servers can use any port, but the client must be able to learn what it is. In the Internet, there is a convention of assigning well-known port numbers to certain applications. The Internet Assigned Number Authority (IANA) oversees this assignment. For example, port number 21 has been assigned to the File Transfer Protocol. When you run an FTP client application, it tries to contact the FTP server on that port by default. A list of all the assigned port numbers is...
maintained by the numbering authority of the Internet (see http://www.iana.org/assignments/port-numbers).

Server examples:

- Web server (port 80)
- FTP server (20, 21)
- Telnet server (23)
- Mail server (25)

4.3. **Connectionless (UDP) Vs Connection-Oriented (TCP) Servers**

A programmer can choose a connection-oriented server or a connectionless server based on their applications. In Internet Protocol terminology, the basic unit of data transfer is a datagram. This is basically a header followed by some data. The datagram socket is connectionless.

- **User Datagram Protocol (UDP):**
  1. Is a connectionless.
  2. A single socket can send and receive packets from many different computers.
  3. Best effort delivery.
  4. Some packets may be lost some packets may arrive out of order.

- **Transmission Control Protocol (TCP):**
  1. Is a connection-oriented.
  2. A client must connect a socket to a server.
  3. TCP socket provides bidirectional channel between client and server.
  4. Lost data is re-transmitted.
  5. Data is delivered in-order.
  6. Data is delivered as a stream of bytes.
  7. TCP uses flow control.

It is simple for a single UDP server to accept data from multiple clients and reply, but it is easier to cope with network problems using TCP.
4.4. Definition of Socket

A socket is one of the most fundamental technologies of computer networking. Sockets allow applications to communicate using standard mechanisms built into network hardware and operating systems. Although network software may seem to be a relatively new "Web" phenomenon, socket technology actually has been employed for roughly two decades.

Software applications that rely on the Internet and other computer networks continue to grow in popularity. Many of today's most popular software packages -- including Web browsers, instant messaging applications and peer to peer file sharing systems -- rely on sockets.

We are now ready to begin using the sockets API. Although clients and servers differ in some aspects of their use of the API, other aspects are common across clients and servers and across TCP and UDP sockets. After dealing with these common aspects, we present the details through an example client and server.

4.5. The Socket Interface

Socket is an Application Programming Interface (API) used for Interprocess Communications (IPC). It is a well-defined method of connecting two processes, locally or across a network. It is a Protocol and Language Independent, often referred to as Berkeley Sockets or BSD Sockets.

The following figure illustrates the example of client/server relationship of the socket APIs for connection-oriented protocol (TCP).
Figure 5: example of client/server relationship of the socket APIs for connection-oriented protocol (TCP)

The following figure illustrates the example of client/server relationship of the socket APIs for a connectionless protocol (UDP).

Figure 6: example of client/server relationship of the socket APIs for a connectionless protocol (UDP)
4.6. System Data Structures for Sockets

In order to use a socket, the kernel needs to keep track of several pieces of data as the following:

1. Protocol Family: a parameter to the socket call.
2. Service Type (Stream, Datagram): parameter to socket.
3. Local IP Address: can be set with `bind()`.
4. Local Port: can be set with `bind()`.
5. Remote IP Address: can be set with `connect()`.
6. Remote Port: can be set with `connect()`.

Ultimately all 6 values must be known to 'make' the communication.

4.7. Active vs Passive Sockets

- A server uses a passive socket to wait the client connections.
- A client uses an active socket to initiate a connection.
- Both start using the `socket()` call.
- Later on, servers and clients will use other calls.

4.8. Socket Endpoints

TCP/IP communication occurs between 2 endpoints. That endpoint is defined as an IP address and a port number. To allow other protocols to merge into the socket abstraction, address families are used.

- We will use `PF_INET` for internet protocol family.
- We will also use `AF_INET` for internet address family. Normally `PF_INET = AF_INET = 2`.

Socket types for `AF_INET` are listed in the following Table.

<table>
<thead>
<tr>
<th>Socket type</th>
<th>Protocol</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>STREAM</td>
<td>TCP, Systems Network Architecture (SNA-IBM), Sequenced Packet eXchange (SPX-Novell).</td>
<td>SOCK_STREAM</td>
</tr>
<tr>
<td>SEQPACKET</td>
<td>SPX.</td>
<td>SOCK_SEQPACKET</td>
</tr>
<tr>
<td>DGRAM</td>
<td>UDP, SNA, Internetwork Packet eXchange (IPX-Novell).</td>
<td>SOCK_DGRAM</td>
</tr>
<tr>
<td>RAW</td>
<td>IP.</td>
<td>SOCK_RAW</td>
</tr>
</tbody>
</table>
There are other address families such as:
1. AF_UNIX
2. AF_NS
3. AF_TELEPHONY

- **AF_UNIX address family**

  The system uses this address family for communicating between two programs that are on the same physical machine. The address is a path name to an entry that is in a hierarchical file system. Sockets with address family AF_UNIX use the `sockaddr_un` address structure:

  ```c
  struct sockaddr_un {
    short sun_family;
    char sun_path[126];
  };
  ```

  - The `sun_family` field is the address family. The `sun_path` field is the pathname. The `<sys/un.h>` header file contains the `sockaddr_un` address structure definition.
  - For the AF_UNIX address family, protocol specifications do not apply because protocol standards are not involved.
  - The communications mechanism between the two processes on the same machine is specific to that machine.

- **AF_NS address family**

  This address family uses addresses that follow Novell or Xerox NS protocol definitions. It consists of a 4-byte network, a 6-byte host (node), and a 2-byte port number. Sockets with address family AF_NS use the `sockaddr_ns` address structure:

  ```c
  struct sockaddr_ns {
    unsigned short sns_family;
    struct ns_addr sns_addr;
    char sns_zero[2];
  };
  ```

- **AF_TELEPHONY address family**

  Telephony domain sockets (sockets that use the AF_TELEPHONY address family) permit the user to initiate (dial) and complete (answer) telephone calls
The AF_TELEPHONY addresses are telephone numbers that consist of up to 40 digits (0 - 9), which are contained in sockaddr_tel address structures. The system supports AF_TELEPHONY sockets only as connection-oriented (type SOCK_STREAM) sockets. Keep in mind that a connection in the telephony domain provides no more reliability than that of the underlying telephone connection. If guaranteed delivery is desired, we must accomplish this at the application level, such as in fax applications that use this family. Sockets with address family AF_TELEPHONY use the sockaddr_tel address structure.

```
struct sockaddr_tel {
    short stel_family;
    struct tel_addr stel_addr;
    char stel_zero[4];
};
```

The telephony address consists of a 2-byte length followed by a telephone number of up to 40 digits (0 - 9).

```
struct tel_addr {
    unsigned short t_len;
    char t_addr[40];
};
```

- The stel_family field is the address family. The stel_addr field is the telephony address, and stel_zero is a reserved field. The <nettel/tel.h> header file contains the tel_addr and sockaddr_tel structure definitions.

### 4.9. Creating socket

To communicate using TCP or UDP, a program begins by asking the operating system to create an instance of the socket abstraction. The function that accomplishes this is `socket()`; its parameters specify the flavor of socket needed by the program.

```
int socket(int protocolFamily, int type, int protocol)
```

- Family specifies the protocol family (AF_INET for Internet, PF_INET for TCP/IP).
• Type specifies the type of service (SOCK_STREAM, SOCK_DGRAM).
• Protocol specifies the specific protocol (usually 0, which means the default).

The return value of socket() is actually an integer: a nonnegative value for success and -1 for failure. A nonfailure value should be treated as an opaque handle, like a file descriptor. (In reality, it is a file descriptor, taken from the same space as the numbers returned by open().) This handle, which we call a socket descriptor, is passed to other API functions to identify the socket abstraction on which the operation is to be carried out. When an application is finished with a socket, it calls close(), giving the descriptor for the socket that is no longer needed.

```
int close(int socket)
```

Close() tells the underlying protocol stack to initiate any actions required to shut down communications and deallocate any resources associated with the socket, close() returns 0 on success or -1 on failure. Once close() has been called, it is not possible to send or receive data through the socket.

### 4.10. Generic socket addresses

In this section and that follows we will discuss the socket APIs details: the structures, functions, macros and types.

- **struct sockaddr**

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

- **sockaddr** consists of the following parts:
  1. The short integer that defines the address family (the value that is specified for address family on the socket() call).
  2. Fourteen bytes that are reserved to hold the address itself.

- Originally sa_len was not there.
- Depending on the address family, sa_data could be a file name or a socket endpoint.
- **sa_family** can be a variety of things, but it’ll be **AF_INET** for everything we do in this Thesis.
- **sa_data** contains a destination address and port number for the socket. This is rather unwieldy since we don’t want to tediously pack the address in the **sa_data** by hand.
- To deal with struct sockaddr, programmers created a parallel structure: struct sockaddr_in ("in" for "Internet".)

- **struct sockaddr_in**

  ```c
  struct sockaddr_in {
    u_char sin_len;         // Address family
    u_short sin_family;     // Address family
    u_short sin_port;       // Port number
    struct in_addr sin_addr; // Internet or IP address
    char sin_zero[8];       // Same size as struct sockaddr
  };
  ```

  - The **sin_family** field is the address family (always **AF_INET** for TCP and UDP).
  - The **sin_port** field is the port number, and the **sin_addr** field is the Internet address. The **sin_zero** field is reserved, and we must set it to hexadecimal zeroes.
  - Data type struct in_addr - this data type is used in certain contexts to contain an Internet host address. It has just one field, named **s_addr**, which records the host address number as an unsigned long int.
  - **sockaddr_in** is a "specialized" sockaddr.
  - **sin_addr** could be u_long.
  - **sin_addr** is 4 bytes and 8 bytes are unused.
  - **sockaddr_in** is used to specify an endpoint.
  - The **sin_port** and **sin_addr** must be in Network Byte Order.
5. TCP SOCKET PROGRAMMING

5.1. TCP Client

The distinction between client and server is important because each uses the sockets interface differently at certain steps in the communication. We first focus on the client. Its job is to initiate communication with a server that is passively waiting to be contacted. The setup for a TCP and UDP client is very similar; however, there are differences in the semantics of sending and receiving. For now, we focus on TCP because sending and receiving over a TCP socket is most similar to file I/O. The typical TCP client goes through three basic steps:

1. Create a TCP socket using `socket()`.
2. Establish a connection to the server using `connect()`.
3. Close the connection with `close()`.

Those are the basic functions of socket in TCP client. In order to communicate with a server, we use `send()` and `recv()` functions.

We have already described the process of socket creation is:

```c
int socket(int protocolFamily, int type, int protocol)
```

And for closing:

```c
int close(int socket)
```

To get a TCP socket, we supply `AF_INET`, `SOCK_STREAM`, and `IPPROTO_TCP`; or we can use the “0” as default parameter; as the parameters to `socket()`. A TCP socket must be connected to another socket before any data can be sent through it. In this sense using TCP sockets is something like using the telephone network. Before we talk, we have to specify the number we want, and a connection must be established; if the connection cannot be established, we have to try again later. The connection establishment process is the biggest difference between clients and servers: The client initiates the connection while the server waits passively for clients to connect to it. To establish a connection with the server, we call `connect()` on the socket.

```c
int connect(int sockfd, struct sockaddr *foreignAddress, unsigned int addressLength)
```
sockfd is the descriptor created by socket(). ForeignAddress is declared to be a pointer to a sockaddr because the sockets API is generic; for our purposes, it will always be a pointer to a sockaddr_in containing the Internet address and port of the server, addressLength specifies the length of the address structure and is invariably given as sizeof(struct sockaddr_in). When connect () returns successfully, the socket is connected and communication can proceed with calls to send() and recv().

\[
\text{int send(int sockfd, const void *msg, unsigned int msgLength, int flags)}
\]

\[
\text{int recv(int sockfd, void *rcvBuffer, unsigned int bufferLength, int flags)}
\]

send() and recv() have very similar arguments, sockfd is the descriptor for the connected socket through which data is to be sent or received. For send(), msg points to the message to send, and msgLength is the length (bytes) of the message. The default behavior for send() is to block until all of the data is sent. For recv(), rcvBuffer points to the buffer; that is, an area in memory such as a character array; where received data will be placed, and bufferLength gives the length of the buffer, which is the maximum number of bytes that can be received at once. The default behavior for recv() is to block until at least some bytes can be transferred.

The flags parameter in both send() and recv() provides a way to change the default behavior of the socket call. Setting flags to 0 specifies the default behavior, send() and recv() return the number of bytes sent or received or -1 for failure. Finally, to terminate the connection, the client calls close().

### 5.2. TCP Server

We now turn our focus to constructing a server. The server's job is to set up a communication endpoint and passively wait for a connection from the client. As with clients, the setup for a TCP and UDP server is similar. For now we focus on a TCP server. There are four steps for TCP server communication:

1. Create a TCP socket using socket().
2. Assign a port number to the socket with bind().
3. Tell the system to allow connections to be made to that port, using listen().
4. Repeatedly do the following:
   - Call accept() to get a new socket for each client connection.
Those are the basic functions used in TCP server. In order to communicate with the client via that new socket, we use `send()` and `recv()` functions.

Creating the socket by `int socket(int protocolFamily, int type, int protocol)`, sending by `int send(int sockfd, const void *msg, unsigned int msgLength, int flags)`, receiving by `int recv(int sockfd, void *rcvBuffer, unsigned int bufferLength, int flags)`, and closing by `int close(int socket)` are the same as in the client. The differences in the server have to do with binding an address to the socket and then using the socket as a channel to "receive" other sockets that are connected to clients. For the client to contact the server, the server's socket must have an assigned local address and port; the function that accomplishes this is `bind()`. We notice that while the client has to supply the server's address to connect, the server has to specify its own address to `bind()`. It is this piece of information (i.e., the server's address and port) that they have to agree on to communicate; neither one really needs to know the client's address.

```
int bind(int sockfd, struct sockaddr *localAddress, unsigned int addressLength)
```

The first parameter is the descriptor returned by an earlier call to `socket()`. As with `connect()`, the address parameter is declared as a pointer to a `sockaddr`, but for TCP/IP applications, it will actually point to a `sockaddr_in` containing the Internet address of the local interface and the port to listen on. `addressLength` is the length of the address structure, invariably passed as `sizeof(struct sockaddr_in)`, `bind()` returns 0 on success and -1 on failure. If successful, the socket identified by the given descriptor (and no other) is associated with the given Internet address and port. The Internet address can be set to the special wildcard value `INADDR_ANY`, which means that connections to the specified port will be directed to this socket, regardless of which Internet address they are sent to; this practice can be useful if the host happens to have multiple Internet addresses. Now that the socket has an address (or at least a port), we need to instruct the underlying TCP protocol implementation to listen for connections from clients by calling `listen()` on the socket.

```
int listen(int sockfd, int queueLimit)
```

The function `listen()` causes internal state changes to the given socket, so that incoming TCP connection requests will be handled and then queued for
acceptance by the program. The queueLimit parameter specifies an upper bound on the number of incoming connections that can be waiting at any time. Listen() returns 0 on success and -1 on failure. At first it might seem that a server should now wait for a connection on the socket that it has set up, send and receive through that socket, close it, and then repeat the process. However, that is not the way it works. The socket that has been bound to a port and marked "listening" is never actually used for sending and receiving. Instead, it is used as a way of getting new sockets, one for each client connection; the server then sends and receives on the new sockets. The server gets a socket for an incoming client connection by calling accept().

```
int accept(int sockfd, struct sockaddr *clientAddress, unsigned int *addressLength)
```

The accept() dequeues the next connection on the queue for socket. If the queue is empty, accept() blocks until a connection request arrives. When successful, accept() fills in the sockaddr structure, pointed to by clientAddress, with the address of the client at the other end of the connection, addressLength specifies the maximum size of the clientAddress address structure and contains the number of bytes actually used for the address upon return. If successful, accept() returns a descriptor for a new socket that is connected to the client. The socket sent as the first parameter to accept() is unchanged (not connected to the client) and continues to listen for new connection requests. On failure, accept() returns -1. The server communicates with the client using send() and recv(); when communication is complete, the connection is terminated with a call to close().
6. UDP SOCKET PROGRAMMING

The previous section talked about functions used the TCP sockets. As already said, TCP guarantees the delivery of packets and preserves their order on destination. Sometimes these features are not required and since they do not come without performance costs, it would be better to use a lighter transport protocol. This kind of service is accomplished by the UDP protocol which conveys datagram packets. Datagram packets are used to implement a connectionless packet delivery service supported by the UDP protocol. Each message is transferred from source machine to destination based on information contained within that packet. That means, each packet needs to have destination address and each packet might be routed differently, and might arrive in any order. Packet delivery is not guaranteed.

There are a few steps involved in using sockets:

1. Create the socket.
2. Identify the socket (name it).
3. On the server, wait for a message.
4. On the client, send a message.
5. Send a response back to the client (optional).
6. Close the socket.

![Figure 7: client-server model for UDP socket](image-url)
A UDP socket can be used to send/receive messages to/from any address and to/from many different addresses in succession. To allow the destination address to be specified for each message, the sockets API provides a different sending routine that is generally used with UDP sockets: `sendto()`. Similarly, the `recvfrom()` routine returns the source address of each received message in addition to the message itself.

```c
int sendto(int sockfd, const void *buffer, size_t length, int flags, const struct sockaddr *dest_addr, socklen_t dest_len);
```

The first parameter, `sockfd`, is the socket that was created with the socket system call and named via `bind`. The second parameter, `buffer`, provides the starting address of the message we want to send. Length is the number of bytes that we want to send. The flags parameter is 0 and not useful for UDP sockets. The `dest_addr` defines the destination address and port number for the message. It uses the same `sockaddr_in` structure that we used in `bind` to identify our local address. As with `bind`, the final parameter is simply the length of the address structure: `sizeof(struct sockaddr_in)`.

```c
int recvfrom(int sockfd, void *restrict buffer, size_t length, int flags, struct sockaddr *restrict src_addr, socklen_t *restrict *src_len);
```

The first parameter, `sockfd`, is a socket that we created ahead of time and used `bind`. The port number assigned to that socket via the `bind` call tells us on what port `recvfrom` will wait for data. The incoming data will be placed into the memory at `buffer` and no more than `length` bytes will be transferred (that's the size of your buffer). We will ignore flags here. We can look at the man page for `recvfrom` for details on this. This parameter allows us to process out-of-band data, peek at an incoming message without removing it from the queue, or block until the request is fully satisfied. We can safely ignore these and use 0. The `src_addr` parameter is a pointer to a `sockaddr` structure that we allocate and will be filled in by `recvfrom` to identify the sender of the message. The length of this structure will be stored in `src_len`. If we do not care to identify the sender, we can set both of these to zero but we will then have no way to reply to the sender. The `recvfrom` call returns the number of bytes that were read into `buffer`. 
7. APIs FOR MANAGING NAMES AND IP

We next consider a number of auxiliary APIs:

- The `hostent` structure: describes IP, hostname pairs.
- `gethostbyname`: hostent of a specified machine.
- `htons, htonl, ntohs, ntohl`: byte ordering.
- `inetpton, inetntop`: conversion of IP numbers between presentation and strings.

7.1. Data Type struct hostent

This data type is used to represent an entry in the host database. It has the following members:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char *h_name</td>
<td>This is the official name of the host.</td>
</tr>
<tr>
<td>Char **h_aliases</td>
<td>These are alternative names for the host, represented as a null-terminated vector of strings.</td>
</tr>
<tr>
<td>Int h_addrtype</td>
<td>This is the host address type; in practice, its value is always AF_INET. In principal other kinds of addresses could be represented in the database as well as Internet addresses; if this were done, you might find a value in this field other than AF_INET.</td>
</tr>
<tr>
<td>Int h_length</td>
<td>This is the length, in bytes, of each address.</td>
</tr>
<tr>
<td>Char **h_addr_list</td>
<td>This is the vector of addresses for the host. Recall that the host might be connected to multiple networks and have different addresses on each one. The vector is terminated by a null pointer.</td>
</tr>
<tr>
<td>Char *h_addr</td>
<td>This is a synonym for h_addr_list[0]; in other words, it is the first host address.</td>
</tr>
</tbody>
</table>

- As far as the host database is concerned, each address is just a block of memory h_length bytes long.
- But in other contexts there is an implicit assumption that we can convert this to a `struct in_addr` or an unsigned long int. Host addresses in a `struct hostent` structure are always given in network byte order.
We can use `gethostbyname()` or `gethostbyaddr()` to search the hosts database for information about a particular host. The information is returned in a statically-allocated structure.

The `gethostbyname()` function returns information about the host named name. If the lookup fails, it returns a null pointer.

```
struct hostent * gethostbyaddr(const char *addr, int length, int format)
```

The `gethostbyaddr()` function returns information about the host with Internet address addr. The length argument is the size (in bytes) of the address at addr.

Format specifies the address format; for an Internet address, specify a value of AF_INET.

If the lookup fails, `gethostbyaddr()` returns a null pointer.

If the name lookup by `gethostbyname()` or `gethostbyaddr()` fails, we can find out the reason by looking at the value of the variable h_errno.

Before using h_errno, we must declare it like this:
```c
extern int h_errno;
```

Here are the error codes that may find in h_errno:

<table>
<thead>
<tr>
<th>H_errno</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST_NOT_FOUND</td>
<td>No such host is known in the database.</td>
</tr>
<tr>
<td>TRY_AGAIN</td>
<td>This condition happens when the same server could not be contacted. If we try again later, we may succeed then.</td>
</tr>
<tr>
<td>NO_RECOVERY</td>
<td>A non-recoverable error occurred.</td>
</tr>
<tr>
<td>NO_ADDRESS</td>
<td>The host database contains an entry for the name, but it doesn’t have an associated Internet address.</td>
</tr>
</tbody>
</table>

We can also scan the entire hosts database one entry at a time using `sethostent()`, `gethostent()`, and `endhostent()`. But in using these functions, they are not reentrant.

**Function void sethostent(int stayopen)**

This function opens the hosts database to begin scanning it. We can then call `gethostent()` to read the entries.

If the stayopen argument is nonzero, this sets a flag so that subsequent calls to `gethostbyname()` or `gethostbyaddr()` will not close the database (as they usually would).

This makes for more efficiency if we call those functions several times, by avoiding reopening the database for each call.
• Function struct hostent * gethostent()

This function returns the next entry in the hosts database. It returns a null pointer if there are no more entries.

• Function void endhostent()

This function closes the host database.

7.2. Network Integers versus Host Integers

Little Endian and big Endian issue regarding the use of the different processor architectures. Usually integers are either most-significant byte first or least-significant byte first. On Intel based machines the hex value 0x01020304 would be stored in 4 successive bytes as: 04, 03, 02, 01. This is a little endian. On a Most Significant Bit (MSB)-first (big endian) machine (IBM RS6000), this would be: 01, 02, 03, 04.

It is important to use network byte order (MSB-first) and the conversion functions available for this task are listed below:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>htons()</td>
<td>Host to network short.</td>
</tr>
<tr>
<td>ntohs()</td>
<td>Network to host short.</td>
</tr>
<tr>
<td>htonl()</td>
<td>Host to network long.</td>
</tr>
<tr>
<td>ntohl()</td>
<td>Network to host long.</td>
</tr>
</tbody>
</table>

We use these functions to write portable network code.
First, let’s say we have a:

```c
struct sockaddr_in ina
```

And we have an IP address "10.12.110.57" that we want to store into it. The function we want to use, `inet_addr()`, converts an IP address in numbers-and-dots notation into an unsigned long. The assignment can be made as follows:

```c
Ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

Notice that `inet_addr()` returns the address in Network Byte Order already so we don’t have to call `htonl()`.
Now, the above code snippet isn’t very robust because there is no error checking. `inet_addr()` returns -1 on error. For binary numbers (unsigned)-1 just
happens to correspond to the IP address 255.255.255.255! That’s the broadcast address!
Actually, there’s a cleaner interface we can use instead of inet_addr(): it’s called
inet_aton() ("aton" means "ascii to network"):

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
```

And here’s a sample usage, while packing a struct sockaddr_in is shown below:

```c
struct sockaddr_in my_addr;
/* host byte order */
my_addr.sin_family = AF_INET;
/* short, network byte order */
my_addr.sin_port = htons(MYPORT);
inet_aton("10.12.110.57", &(my_addr.sin_addr));
/* zero the rest of the struct */
memset(&(my_addr.sin_zero), 0, 8);
```

The function inet_aton(), unlike practically every other socket-related function,
returns non-zero on success, and zero on failure. And the address is passed
back in inp. Unfortunately, not all platforms implement inet_aton() so, although
its use is preferred, normally the older more common inet_addr() is used.

All right, now we can convert string IP addresses to their binary representations. What about the other way around? What if we have a
struct in_addr and we want to print it in numbers-and-dots notation? In this
case, we’ll want to use the function inet_ntoa() ("ntoa" means "network to ascii") something like this:

```c
printf("%s", inet_ntoa(ina.sin_addr));
```

That will print the IP address. That inet_ntoa() takes a struct in_addr as an
argument, not a long. Also that it returns a pointer to a char.
This points to a statically stored char array within inet_ntoa() so that each time
we call inet_ntoa() it will overwrite the last IP address we asked for.

For example:

```c
char *a1, *a2;
...
a1 = inet_ntoa(ina1.sin_addr); /* this is 192.168.4.1 */
a2 = inet_ntoa(ina2.sin_addr); /* this is 10.11.110.55 */
printf("address 1: %s\n", a1);
printf("address 2: %s\n", a2);

Will print:
address 1: 10.11.110.55
address 2: 10.11.110.55

7.3. Conversion of IP numbers

- **Inet_ntop**

  \[
  \text{int inet_ntop ( int family, const char *addrPtr, char *strPtr, size_t len );}
  \]

  That Function returns 1 if OK, 0 if presentation error, -1 error. Where family is either AF_INET or AF_INET6. The strPtr is the return IP address as a dotted string. Finally, addrPtr points to either the 32 bit (AF_INET) or 128 bit (AF_INET6). And the length is the size of destination.

- **Inet_pton**

  \[
  \text{int inet_pton ( int family, const char *addrPtr, char *strPtr);}
  \]

  That Function returns 1 if OK, 0 if presentation error, -1 error. Where family is either AF_INET or AF_INET6. The strPtr is the return IP address as a dotted string. Finally, addrPtr points to either the 32 bit (AF_INET) or 128 bit (AF_INET6).
8. DESIGN

The objective of this phase is the design and implementation an online lab exercises of developing a client server model using socket programming in C, thus jFed and other tools such as SSH widget and html are used to facilitate the development of lab course as an interactive learning and training channel for both students and professionals.

For the design, our objective is to create a course lab in a FORGE in order to facilitate to student to get the role of basic functions of socket. The idea is not learning C programming but leading students to get familiar with client server model and to focus only on very specific piece of code concerning those basic function of socket in C. It consists of three parts:

- Creation of the course as a module giving all the backgrounds needed to proceed with exercises.
SOCKET PROGRAMMING IN C - CLIENT SIDE

SOCKET PROGRAMMING IN C - CLIENT SIDE - THEORY

1. Objectives.

The objectives of this laboratory are getting familiar with the client-server communication model, learning the most important functions used for the design of the client-server applications using socket programming in C, designing and implementing simple client and server applications for a stream communication-TCP socket.

2. Backgrounds.

1.1. Definition of a Socket

Sockets provide a standard interface between the network and application. It exists two types of socket: a stream socket that provides a reliable byte stream transport service, while a datagram socket provides unreliable delivery of individual data packets. When used in the Internet environment, stream sockets correspond to TCP/IP connections and datagram sockets to UDP/IP datagrams. This lab exercise will only consider TCP/IP stream sockets, as they are by far the most widely used.

A socket is bound to a port number, which is very important, so that the TCP layer can identify the application that data is destined to be sent.

As mentioned before, “PORT” is an important parameter, as identifier used for making the difference between sockets found at the same address. There can be used addresses between 1024 and 49151 (these are registered ports) and other addresses between 49152 and 65535, called also dynamic ports; nevertheless the addresses between 1 and 1023 are reserved for the system.

1.2. TCP/IP and UDP/IP communications

There are two communication protocols used for socket programming: Datagram communication and Stream communication.

1.2.1. UDP Datagram communication

The datagram communication protocol is known as UDP (User Datagram Protocol). It is a connectionless protocol, means for each time you send datagrams, you need also to send the local socket descriptor and the receiving socket’s address. As you can tell, additional data must be sent each time a communication is made.

1.2.2. TCP Stream communication

The stream communication protocol is known as TCP (Transfer Control Protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a connection has first to be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

The selection of the protocol (UDP or TCP) depends on the client-server application you are writing.

In brief, TCP is useful for implementing network services such as remote login (login), telnet, and the file transfer (FTP) which require data of indefinite length to be transferred. UDP is less complex and incurs fewer overheads.
3. Sockets using TCP connection:

3.1. Client-Server model

Any operation inside a network can be seen as a client process that communicates with a server process. The server process creates a socket, associates it with the socket as an address and launches a mechanism for listening to the connection requests of the clients. The client process creates a socket and asks for a connection to the server. After accepting the request by the server and after the connection has been established, a communication between the sockets can be established.

The following figure illustrates the example of client-server relationship of the socket APIs for connection-oriented protocol (TCP).

The usual functions from the socket library are:

- socket() - creates a new socket descriptor;
- bind() - makes the binding of an address and of the corresponding port to the socket;
- connect() - is the establishment of a connection with a remote server;
- listen() - listens for the connection requests; this function is used in a passive socket;
- accept() - allows the creation of a new socket, corresponding to a connection request;
- send(), recv(), sendto(), recvfrom() - transmits/receives streams or datagrams;
- close(), shutdown() - closes a connection;
- gethostname(), gethostbyname() - determines the name and the address from the specified host.

The communication between the client processes and the server is based on the implement of the socket() function, that returns a socket descriptor. This descriptor is used in the calls of different functions specialized in the data transmissions (as send() and recv()).
The system call to open a socket is socket():

   int fd = socket(domain, type, protocol);

The arguments can be described as follows:

   domain is the addressing domain of the socket, which for our purposes is an internet socket or AF_INET.
   type is the type of socket describing which transport protocol we are using. If the traffic is over TCP, so that would be SOCK_STREAM. If we wanted to open a UDP socket, the keyword would be SOCK_DGRAM.
   protocol is additional information about the protocol of the socket, but we will not need to use this option, so we'll set it to 0.

The return value of socket() is a socket file descriptor (fd) on success, and -1 on failure. All further socket operations, connect(), read(), write(), and so on, require that integer file descriptor.

Opening a socket isn’t the same as connecting the socket to a foreign address. To act as a client and connect to a server using the socket, you use the connect() system call:

   int connect(fd, const struct sockaddr *address, int address_len);

which takes a socket file descriptor, a pointer to a socket address, and the length of that address. The return value of connect() indicates a successful or failed connection.

Note that if your computer has multiple interfaces, you must first bind() your socket to one of the interfaces using the bind() system call. It has the following function definition:

   int bind(fd, struct sockaddr *address, int address_len);

where the address indicates which of the severals interfaces, identified by IP address, this socket should use for listen() (or connect()).

Listening and Accepting a Connection:

On the server end, a socket must be set up so that it can accept incoming connections. This occurs in two parts, first it requires a call to listen(), and second, a call to accept().

The function definition for listen() is as follows:

   int listen(fd, int backlog);

Of course, listen() first argument is the socket file descriptor that the will be listened on. The second argument, backlog indicates the number of queued or backlogged incoming connections that can be pending waiting on an accept() call before a connection refused message is sent to the connecting client.

The accept() function is the key server side mechanism of socket programming. Let’s start by inspecting its function definition:

   int accept(fd, struct sockaddr *address, int *address_len);

Essentially, given a socket that is listening to incoming connections, accept will block until a client connects, filling in the address of the client in address and the length of the address in address_len.

The return value of accept is very important. It returns a new socket file descriptor for the newly accepted connection. The information about this socket is encoded in address, and all further communication with this client occurs over the new socket file descriptor. Don’t forget to close the socket when done communicating with the client.

Reading and Writing:

Reading and writing from a stream socket occurs very much like reading and writing from any standard file descriptor. There are a number of functions to choose from, I suggest that you use the standard read() and write() system calls.

Writing is so simple:

   write(fd, data, datalen);

which, given a file descriptor (fd) and a data, write() will write data lengths to the destination described in the file descriptor and return the number of bytes written. Reading from a socket is slightly more complicated because you cannot be certain how much data is going to be sent ahead of time. First consider the function definition of read():

   read(fd, buffer, buflen)

Similar to write(), it will read from the give file descriptor (fd), and place up to buflen into the buffer pointed to by buffer, returning the number of bytes read.

Closing a Socket:

To close a socket, you simply use the standard file descriptor close() function:

   int close(fd);
3.2 Data types used by the socket interface

The next structure stores the socket address for different types of sockets:

```c
struct sockaddr
{
    unsigned short sa_family; /* the family of addresses, AF_INET */
    char sa_data[14]; /* 14 bytes - the protocol address */
};
```

In order to be able to use in programs the structure `struct sockaddr`, a parallel structure `struct sockaddr_in` has been designed:

```c
struct sockaddr_in{
    short int sin_family; /* the family of addresses */
    unsigned short int sin_port; /* the port number */
    struct in_addr sin_addr; /* the internet address */
    char sin_zero[8]; /* unused */
};
```

and

```c
/* the internet address */
struct in_addr{
    unsigned long s_addr;
};
```

This structure allows a simple reference to the elements of a socket address. `sin_zero` will be set to 0, using `bzero()` or `memset()` functions. The pointer to the `struct sockaddr_in` maps the pointer to the structure `struct sockaddr`. We must highlight that `sin_family` corresponds to `sa_family` in the structure `struct sockaddr`, and it will be set to `AF_INET`, value used for the TCP/IP protocol family.

3.3 Host Names - Translating host names to host IP numbers

Besides the standard numbers and dots notation for Internet addresses, you can also refer to a host by a symbolic name. The advantage of a symbolic name is that it is usually easier to remember.

Internally, the system uses a database to keep track of the mapping between host names and host numbers. This database is usually referred to as the `hosts` file or an equivalent provided by a name-DNS server. The functions and other symbols for accessing this database are declared in `netdb.h`. They are BSD features, defined unconditionally if you include `netdb.h`.

The IP address and vice versa is called name resolution. It is done by Domain Name Service. Other than the hosts file, in Windows platforms it is called DNS (Domain Name Service).

The complete process or steps taken for name resolution quite complex but normal users normally use DNS service and UNIX/Linux normally use DNS.

* Data Type struct hostent

This data type is used to represent an entry in the hosts database. It has the following members:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *h_name</td>
<td>This is the official name of the host.</td>
</tr>
<tr>
<td>char **h_aliases</td>
<td>These are alternative names for the host represented as a null-terminated vector of strings.</td>
</tr>
<tr>
<td>int h_addrtype</td>
<td>This is the host address type. In practice, its value is always AF_INET. In principal, other kinds of addresses could be represented in the data base as well as Internet addresses; if this were done, you might find a value in this field other than AF_INET.</td>
</tr>
<tr>
<td>int h_length</td>
<td>This is the length, in bytes, of each address.</td>
</tr>
<tr>
<td>char **h_addr_list</td>
<td>This is the vector of addresses for the host. Recall that the host might be connected to multiple networks and have different addresses on each one. The vector is terminated by a null pointer.</td>
</tr>
<tr>
<td>char *h_name</td>
<td>This is a synonym for h_addr_list[0]; in other words, it is the first host address.</td>
</tr>
</tbody>
</table>

As far as the hosts database is concerned, each address is just a block of memory h_length bytes long. But in other contexts there is an implicit assumption that you can convert this to struct in_addr or an unsigned long int. Host addresses in a struct hostent structure are always given in network byte order.

You can use `gethostbyname()` or `gethostbyaddr()` to search the hosts database for information about a particular host. The information is retained in a statically-allocated structure. You must free the information if you need to save it across calls.

```c
struct hostent * gethostbyname(const char *name)
```
The gethostbyname() function returns information about the host name. If the lookup fails, it returns a null pointer.

```
struct hostent * gethostbyname(const char *host, int format = AF_INET);
```

The gethostbyname() function returns information about the host with Internet address addr. The length argument is the size (in bytes) of the address at addr. The format specifies the address format; for an Internet address, specify a value of AF_INET.

If the lookup fails, gethostbyname() returns a null pointer. And if the name lookup by gethostbyname() or gethostbyaddr() fails, you can find out the reason by looking at the value of the variable h_errno.

<table>
<thead>
<tr>
<th>h_errno</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST_NOT_FOUND</td>
<td>No such host is known in the database.</td>
</tr>
<tr>
<td>TRY_AGAIN</td>
<td>This condition happens when the same server could not be contacted. If we try again later, we may succeed then.</td>
</tr>
<tr>
<td>NO_RECOVERY</td>
<td>A non-recoverable error occurred.</td>
</tr>
<tr>
<td>NO_ADDRESS</td>
<td>The host database contains an entry for the name, but it doesn’t have an associated Internet address.</td>
</tr>
</tbody>
</table>

- Creation of two interactives part, one concerning connection to the terminal of the server and the other one concerning connection to the client. Both accesses are by the SSHweb widget. All the information concerning the design of this part is at the section 9.2 of lab exercises.

- Creation of an application form using HTML as programming language in order to facilitate to student to save their answers concerning problems found during executing those files and their solutions for good connectivity between client and server.
9. IMPLEMENTATION

9.1. jFed

jFed was developed at the iMinds research institute and Ghent University, and was partially funded by the European Commission through the FP7 Project Fed4FIRE.

Fed4FIRE works with X.509 certificates to authenticate and authorize experimenters on testbeds. Before starting the experiment using jFed tool, we should create an account and a certificate. Testbeds can decide which certificates from which authorities they accept and as such federations of testbeds and authorities are established. Fed4FIRE is such a federation.

In the end, we will have a file on our local PC which contains this certificate information and this can then be used in experimenter tools. This procedure has to be run only once.

9.1.1. Register for an account

The current authority which can be used in the Fed4FIRE federation is located at http://authority.ilabt.iminds.be.

Figure 8: iMinds authority interface
To request a new account, we click on Sign Up. The authority has the concept of Projects which bundle multiple experimenters. In a Project, the PIs (Principal Investigators) can approve new experimenters for that project, without Fed4FIRE administrators needing to approve this.

Here we will start with creating a new project by clicking Start New Project. Then we have to fill in our account information. If all goes well, we will receive an email in our mailbox with a code to verify the account which we have to fill in the dialog box.

Now, it can take some time to have a Fed4FIRE administrator or project administrator approved our request. We will receive an email when that is done. After that, we can login on the authority portal and start jFed immediately or we can download the approved certificate.

9.1.2. **Start up jFed**

When we have a Fed4FIRE account, we can go on with the experiment to show how we can access testbed resources. For this we will use the jFed tool.

We need to have a recent java 7 or java 8 (preferred) running. Since we have already an account from [https://authority.ilabt.iminds.be](https://authority.ilabt.iminds.be), we can login and click the green button to start the jFed tool. Alternatively, we can download the certificate, and store it locally on pc, then go to [http://jfed.iminds.be](http://jfed.iminds.be) and click the green button Quickstart jFed experimenter tool.

A dialog box will pop up saying that the Java application is signed by iMinds. Only we need to click on Run and optionally we can tick the box ‘Do not show this again’.

9.1.3. **Run jFed**

When jFed starts up for the first time, we will get the dialog box in order to fill in the necessary information.
If we started jFed through https://authority.ilabt.iminds.be, the right information will be filled in, and we only need to type in our password.

But, if we started jFed from http://jfed.iminds.be, we should browse to the pem file that we downloaded from the authority portal.

Now jFed should automatically show our Username and authority as shown below.

![jFed login dialogue box](image)

*Figure 9: jFed login dialogue box*

Then, we should type our password for this certificate and click on Login. A dialog box will pop up to say that we have to configure jFed for this initial run.

**9.1.4. Create the experiment**

When we have logged in, and filled in our preferences, we see jFed with no experiments loaded.
When we click on New, we get a blank canvas where we can draw our experiment. Let’s drag in a generic node from the left side to the canvas. For this specific experiment we can right click and configure the node.
For our experiment, our objective is to create 5 clients and 5 servers so we can make sure that 5 students at the same time can have access to those resources independently to each other.

![Figure 12: design five node of clients and 5 nodes of servers in jFed tool](image)

### 9.1.5. RSpec editor

It is now possible to toggle between a graphical editor and a raw RSpec editor, which makes it possible to add extra information or to put in new RSpecs. With the Format Code button we can optimize the XML view on the RSpec. With Rspec editor, a user can create a jFed experiment/test with two different ways. The first -visual- is the easier way and is generally preferred by the new users, as shown in this figure.
The second, as it is shown in the following figure, which is preferred by the advanced users is the manual way which we use for this experiment. With the manual way a user describes his experiment/test by creating an XML formatted script (file extension: rspec). With this script the user can activate more features for his test (that are not possible using the visual tool). Such capabilities are the insertion of additional certification keys, the installation of extra software in the test nodes etc. (See the appendix-C for the Rspec code).
9.1.6. Run the experiment

Let’s run this experiment, by clicking the tab General at the top, and then the Run button. We will now have to choose a name for the experiment at a slicename: `uopclients` and choose a maximum duration.
It will now take a couple of minutes to get nodes prepared. Now, when nodes become green, we can right click on the node, and click on Open SSH terminal. To release the resources before the end time of our experiment, we can click on the terminate button at the top in jFed. After that the nodes will become black and if our ssh connection is still open, we can see that the nodes will be shutdown.

9.1.7. SSH widget

One of the most used means to access machines residing in FIRE facilities are through terminal utilities. Planetlab, VirtualWall, BonFIRE cloud facilities, OFELIA Openflow testbeds, wireless access nodes and almost any other facility that offers full access to resources, offer the for experimentation via terminal access. The most known protocol is Secure Shell SSH, since it enables an experimenter to access a resource residing in FIRE facility and fully manage it via the command line. SSH client tools exist for almost any operating system, either natively in the system or via external downloaded tools.

*Figure 16: access to machines residing in FIRE facilities through terminals utilities*
Having such an ssh client is the primary requirement for accessing most of FIRE resources. The next requirement is authenticating to the target system. FIRE facilities usually offer the two most common usages, via username and password combination or through public/private keys. The latter need from experimenter some more special configuration for their clients, especially for generating the key pairs.

9.2. Lab exercise

The objective of creating an online lab course exercises (remote laboratory) on a FORGE is to complement the learning of a subject about client-server model using socket programming in C: client side. Offering series of practical exercises, three different clients with connection problems with one correct server, to improve students understanding of the subject and getting familiar with the basic functions of socket. Giving the first, all facilities to adapt according to their needs or preferences course development that serves as an introduction to use socket and that feels a sufficiently solid base to improve learning in the future.

Thanks to FORGE that has a diversity of tools, which include interactive activities combining simulations, virtual experiments and more, from the environments, students are able to edit the solutions of exercises proposed by the teacher, to compile and to execute them.

In the FORGE page, after installing and uploading everything (codes) in the server, students can start doing their exercises following some steps that a teacher described and explained before. Below in figures, we will explain how to use that platform to proceed with those set of exercises.
SOCKET PROGRAMMING IN C - CLIENT SIDE-IMPLEMENTATION

1. Implementation of server-client model.

In this exercise you will make 3 clients test programs which are problematic: client1.c, client2.c, client3.c with a correct server server1.c, that are connected through a logic pipeline to get familiar to the basic socket functions.

You will see in each test what kind of messages appears during the connection, and if any error message occurs, modify the code using the VIM editor.

In the first part you need to compile and run the server.

In the second part you compile and run first the client 1. Then you should check the connectivity between the client 1 and server 1. The same procedure will be for client 2 and client 3.

In the third part you should write and describe problems found in each client and their solutions in a form and submit them by hitting a 'Submit' button.

Please make sure to read all the instructions before to start exercises.

Important Information regarding this course!

To access this course interactive parts please use:
username = ucpe1111
password=1111
Server Machine: server1.uoeclients.walldl.labt.iminds.be
Client Machine: client1.uoeclients.walldl.labt.iminds.be

Please see above the provided credentials

2. As soon as you are successfully connected, run the following scripts (located in the login directory).

   cd /home/clients

3. To get the IP address of the server, type:

   ifconfig

4. First run the server.

   gcc server1.c

   server

5. After the completion of the tasks and collection of your output, stop the server.

   gcc client1.c

Important Information regarding this course!

To access this course interactive parts please use:
username = ucpe1111
password=1111
Server Machine: server1.uoeclients.walldl.labt.iminds.be
Client Machine: client1.uoeclients.walldl.labt.iminds.be

Please see above the provided credentials

ssh2web - forgebox - Login

Username
Password
Machine

Use remote machine credentials
Login

ForgeBox - ssh2web 1.02.02 - http://www.forgebox.eu/ - 2014
After using those credentials in SSH widget boxes, students have access to the shell of each terminal server/client, so they can proceed for their exercises. For the correct running, we should run the server then the client at the terminal. (See the appendix-B- for exercises).

Following those steps, the student gets face to some problems which have to be corrected. Bellow, we will show some figures giving idea about how the terminal is and how the problems are shown.
• First running the server.

• Then running the client using the ip address of the server.

• As we see, it is shown an error concerning the connectivity. Here students should edit the code using the VIM editor and recompile it until they get the successful message of connectivity.

• Of course, when the client is running correctly, the server sends a message to the client and vice versa.
In the following, we will show other kinds of error messages in client2.c and client3.c which have to be corrected.

In the end, we implemented an HTML form in order to facilitate students to describe and explain problems found and corrected in each client.

HTML forms are one of the main points of interaction between a user and a web site or application. They allow users to send data to the web site. Most of the time that data is sent to the web server, but the web page can also intercept it to use it on its own.

An HTML Form is made of one or more widgets. Those widgets can be text fields (single line or multiline), select boxes, buttons, checkboxes, or radio buttons.
Most of the time, those widgets are paired with a label that describes their purpose.

In our design, we used text fields and a submit button to submit the form. Below, the figure shows how we created that part.
10. CONCLUSION AND FUTURE WORKS

Starting from the general objective of providing an online FORGE lab course of teaching the development of client-server model using socket programming in C, our intention was not to learn C coding but to highlight to students the basic function of sockets. Students will better understand the role of each function during the connection of client and server by correcting those set of exercises. The course is running now on the server, prepared to receive students and integrating them into e-classes to join the course.

Through FORGE, traditional online courses will be complemented with interactive laboratory courses, supplying an in-depth and hands-on educational experience. In addition, as future work, FORGE will redesign eLearning tools and enhance existing Learning Management Systems (LMSs) with new functionalities to enable a seamless interactive experience when accessing FIRE facilities, thus providing a hands-on experience using the latest networking equipment, with evidence of innovative and advanced practice for learners. FORGE will also allow educators to efficiently create, use and re-use FIRE-based learning experiences through tools and techniques. And, most importantly, it will enable equity of access to the latest ICT systems and tools independent of location and at low cost, strengthening the culture of online experimentation tools and remote facilities.

Currently the FORGE project is in a progress stage. The design of the courses and teaching materials and the enabling of the facilities are tasks that are ongoing. In the near future, FORGE will have an evaluation of the experience from the learners and educators point of view. This feedback will provide valuable information to improve and reinforce the educational model that FORGE proposes.
APPENDIX

A. BACKGROUNDS

A computer network consists of machines called hosts and routers interconnected by communication channels. Hosts are computers that run applications such as Web browser; the application programs running in hosts are really the "users" of the network. Routers are machines whose job is to relay or forward information from one communication channel to another. They may run programs but typically do not run application programs. For our purposes, a communication channel is a means of conveying sequences of bytes from one host to another; it may be a broadcast technology like Ethernet, a dial-up modem connection, or something more sophisticated.

The Routers machines are important simply because it is not practical to connect every host directly to every other host. Instead, a few hosts connect to a router, which connects to other routers, and so on to form the network. This arrangement lets each machine get by with a relatively small number of communication channels; most hosts need only one. Programs that exchange information over the network, however, do not interact directly with routers and generally remain blissfully unaware of their existence.

That exchange of information, here we mean sequences of bytes that are constructed and interpreted by programs, these byte sequences are generally called packets. A packet contains control information that the network uses to do its job and sometimes also includes user data. An example is information about the packet's destination. Routers use such control information to figure out how to forward each packet.

Figure 17: TCP/IP network
In this thesis, we will concentrate more on the Transport and Network layer of the TCP/IP stack. More detail TCP/IP stack with typical applications is shown below.

The common applications that we encounter in our everyday use are:

1. FTP (file transfer protocol).
2. SMTP (simple mail transfer protocol).
3. telnet (remote logins).
4. rlogin (simple remote login between UNIX machines).
5. World Wide Web (built on http) and https (secure http).
6. NFS (network filing system – originally for Sun Microsystems).
7. TFTP (trivial file transfer protocol – used for booting).
8. SNMP (simple network management protocol).

A protocol is a standard rules that enables and controls the connection, communication and data transfer between two computing endpoints. Protocols may be implemented by hardware, software, or a combination of the two. At the lowest level, a protocol defines the behavior of a hardware connection. In term of controls, protocol may provide data transfer reliability, resiliency and integrity. An actual communication is defined by various communication protocols. In the context of data communication, a network protocol is a formal set of rules, conventions and data structure that governs how computers and other network devices exchange information over a network. A wide variety of network protocols exist, which are defined by many standard organizations worldwide and technology vendors over years of technology evolution and developments. One of the most popular network protocol suites is TCP/IP, which is the heart of internetworking communications.

1. TCP vs UDP

- Both use port numbers
  - application-specific construct serving as a communication endpoint
  - 16-bit unsigned integer, thus ranging from 0 to 65535
  - to provide end-to-end transport
- UDP: User Datagram Protocol
  - no acknowledgements
  - no retransmissions
  - out of order, duplicates possible
  - connectionless, i.e., app indicates destination for each packet
- TCP: Transmission Control Protocol
  - reliable byte-stream channel (in order, all arrive, no duplicates)
  - similar to file I/O
  - flow control
  - connection-oriented
  - bidirectional
TCP is used for services with a large data capacity, and a persistent connection, while UDP is more commonly used for quick lookups, and single use query-reply actions. Some common examples of TCP and UDP with their default ports:

<table>
<thead>
<tr>
<th>Service</th>
<th>Protocol</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS lookup</td>
<td>UDP</td>
<td>53</td>
</tr>
<tr>
<td>FTP</td>
<td>TCP</td>
<td>21</td>
</tr>
<tr>
<td>HTTP</td>
<td>TCP</td>
<td>80</td>
</tr>
<tr>
<td>POP3</td>
<td>TCP</td>
<td>110</td>
</tr>
<tr>
<td>Telnet</td>
<td>TCP</td>
<td>23</td>
</tr>
</tbody>
</table>

2. Port numbers and services

It is 16 bit integers. So we have $2^{16} = 65536$ ports maximum. It is unique within a machine/IP address. Every service or application will have their own port number.

To make a connection we need an IP address and port number of the protocol. The connection defined by:

**IP address & port of server + IP address & port of client**

Normally, server port numbers are low numbers in the range 1 – 1023, normally called well known port number and normally assign for root (Administrator) only. It is used for authentication e.g. rlogin. And normally, client port numbers are higher numbers starting at 1024.

A server running on a well-known port lets the OS know what port it wants to listen on. Whereas a client normally simply lets the operating system picks a new port that isn’t already in use.

3. Numeric IP Addresses

IPv4 (Internet Protocol version 4) Internet address is 32 bit integers. The IP stand for Internet Protocol. For convenience they are displayed in "dotted decimal" format. Each byte is presented as a decimal number. Dots separate the bytes, for example: 127.0.0.1
4. IP Address Classes

To simplify packet routing, internet addresses are divided into classes. An IP address has two parts: The network portion and the host portion.

The network portion is unique to each company/organization/domain/group/network, and the host portion is unique to each network device (host) in the network. Where the network portion ends and the host portion begin is different for each class of IP address.

We can determine this by looking at the two high-order bits in the IP address.

<table>
<thead>
<tr>
<th>Class and Network size</th>
<th>Range (decimal)</th>
<th>Network ID</th>
<th>Host ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class A (Large)</td>
<td>1 -127</td>
<td>Byte 1</td>
<td>Bytes 2, 3, 4</td>
</tr>
<tr>
<td>Class B (Medium)</td>
<td>128 – 191</td>
<td>Bytes 1, 2</td>
<td>Bytes 3, 4</td>
</tr>
<tr>
<td>Class C (Small)</td>
<td>192 – 223</td>
<td>Bytes 1, 2</td>
<td>Bytes 4</td>
</tr>
</tbody>
</table>

- The first four bits (bits 0-3) of an address determine its class:
  0xxx = Class A which bits 1-7 define a network, bits 8-31 define a host on that network. So we have 128 networks with 16 million hosts.
  10xx = Class B which bits 2-15 define a network, bits 16-31 define a host on that network. So we have 16384 networks with 65536 hosts.
  110x = Class C which bits 2-23 define a network, bits 24-31 define a host on that network. So we have 2 million networks with 256 hosts.

- The IP network portion can represent a very large network that may span multiple geographic sites. To make this situation easier to manage, we can use sub networks, which use the two parts of the address to define a set of IP addresses that are treated as group. The sub netting divides the address into smaller networks.
- We configure a sub network by defining a mask, which is a series of bits. Then, the system performs a logical AND operation on these bits and the IP address.
- The 1 bit defines the sub network portion of the IP address (which must include at least the network portion). The 0 bits define the host portion.
- Class D is a multicast address and class E is reserved.
As a summary:

Nowadays we use classless IP address. That means we subnet the class type IP into smaller subnet or smaller group of IP addresses creating smaller networks. The example can be found in Classless Inter-Domain Routing (CIDR) and the private domain normally uses the private IP range. The private IP range cannot be routed in the Internet domain.

A network technology that deploys the private IP is Virtual Private Network (VPN) while the advanced subnetting technology using private IP can be found in Virtual LAN (VLAN). Before the IPV4 run out of the IP addresses, now we have IPV6 with 128 bits.

5. Host Names and DNS

People need names to make it simpler to use the Internet instead of the dotted decimal. The Domain Name System (DNS) can translate from name (domain name) to number (IP address) or from number to name. This is called name resolution.

Name resolution done by Domain Name Service (DNS – although the term is same as the Domain Name System and same acronym, this is Microsoft implementation of the Domain Name System) in Windows and in Unix/Linux it is implemented using Berkeley Internet Name Domain (BIND).
B. EXERCISES

1. Server.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

/* the port users will be connecting to */
#define MYPORT 5001

/* how many pending connections queue will hold */
#define BACKLOG 10

/* BufferLength 100*/
#define BufferLength 100

void sigchld_handler(int s)
{
    while(wait(NULL) > 0);
}

/* BufferLength 100*/
#define BufferLength 100

void sigchld_handler(int s)
int main(int argc, char *argv[])  
/* listen on sock_fd, new connection on new_fd */
int sockfd, new_fd, rc;
int totalcnt = 0;
char buffer[BufferLength];
/* my address information */
struct sockaddr_in my_addr;
/* connector’s address information */
struct sockaddr_in their_addr;
int sin_size;
struct sigaction sa;
int yes = 1;
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    perror("Server-socket() error lol!
");
    exit(1);
}
else
    printf("Server-socket() is OK...
");
/*the setsockopt() is used to allow the local address to be reused*/
/*when the server is restarted before the required wait time expires.*/
if ((rc = setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))) == -1)
{
    perror("Server-setsockopt() error lol!");
exit(1);
}

/* host byte order */
my_addr.sin_family = AF_INET;

/* short, network byte order */
my_addr.sin_port = htons(MYPORT);

/* automatically fill with server IP */
my_addr.sin_addr.s_addr = INADDR_ANY;

/* zero the rest of the struct */
memset(&(my_addr.sin_zero), '\0', 8);

/* After the socket descriptor is created, a bind() gets a unique*/
/* name for the socket. */
if((rc = bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))) == -1)
{
    perror("Server-bind() error");
    exit(1);
}
else

    printf("Server-bind() is OK...
");

/* the listen() function allows the server to accept*/
/* incoming client connections. Here the backlog is set to 10 */
/* the system can queue up to 10 connection requests before the system*/
/* starts rejecting incoming requests*/
if((rc = listen(sockfd, BACKLOG)) == -1)
{
    perror("Server-listen() error");
}
exit(1);
}

printf("Server-listen() is OK...Ready for Listening\n\n");

/* clean all the dead processes */
sa.sa_handler = sigchld_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if(sigaction(SIGCHLD, &sa, NULL) == -1)
{
    perror("Server-sigaction() error");
    exit(1);
}

/* accept() loop */
/*with this accept() function provided the connection request does the following:*/
/*-is part of the same address family*/
/*-uses streams socket TCP*/
/*attempts to connect to the specified port*/

while(1)
{
    sin_size = sizeof(struct sockaddr_in);
    if((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1)
    {
        perror("Server-accept() error");
        continue;
    }

    exit(1);
printf("Server-accept() is OK...\n");

/*Client IP*/
printf("Server: Got connection from %s\n\n", inet_ntoa(their_addr.sin_addr));

/*Read data from client*/
totalcnt = 0;

while(totalcnt < BufferLength)
{
    rc = read(new_fd,&buffer[totalcnt],(BufferLength - totalcnt));
    if(rc < 0)
    {
        perror("Server-read() error");
        close(sockfd);
        close(new_fd);
        exit(-1);
    }
    else if (rc == 0)
    {
        printf("Client program has issued a close ()\n");
        close(sockfd);
        close(new_fd);
        exit(-1);
    }
    else
    {
}
totalcnt += rc;

printf("Server-read() is ok \n");

/*Show the data*/

printf("Recieved data from the client : %s\n",buffer);

/* this is the child process */

if(!fork())
{
  /* child doesn’t need the listener */
  close(sockfd);
  if(send(new_fd, "This is a test string from server!\n", 37, 0) == -1)
    perror("Server-send() error lol!");
  close(new_fd);
  exit(0);
}
else
  printf("Server-send is OK...!\n\n");

/* parent doesn’t need this*/
close(new_fd);

return 0;
2. Client1.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

// the port client will be connecting to
#define PORT 5001

// max number of bytes we can get at once
#define MAXDATASIZE 300

('//BufferLength is 100 Bytes
#define BufferLength 100

int main(int argc, char *argv[])
{
  int sockfd, numbytes, rc;
  char buf[MAXDATASIZE];
  char buffer[BufferLength];
  struct hostent *he;
  char data[100] = "this is a test string from client :)!");
// connector’s address information
struct sockaddr_in their_addr;

// if no command line argument supplied
if(argc != 2)
{
    fprintf(stderr, "Client-Usage: %s the_client_hostname\n", argv[0]);
    // just exit
    exit(1);
}

// get the host information
if((he=gethostbyname(argv[1])) == NULL)
{
    perror("gethostbyname()");
    exit(1);
}
if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{
    perror("socket()");
    exit(1);
}
else

    printf("Client-The socket() is OK...\n");

    // host byte order

    their_addr.sin_family = AF_INET;
// short, network byte order
their_addr.sin_port = htons(PORT);

their_addr.sin_addr = *((struct in_addr *)he->h_addr);

// zero the rest of the struct
memset(&(their_addr.sin_zero), '\0', 8);

if((rc = connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1)
{
    perror("connect()");
    exit(1);
}
else
{
    printf("Client-The connect() is OK...

");
/*send string to server */
printf("Sending some string to the %s...
",argv[1]);
rc = write(sockfd,data,sizeof(data));

if(rc < 0)
{
    perror("Client-write() error");
    if (rc == 0)
    {
        perror("SO_ERROR was");
    }
    close(sockfd);
}
exit(-1);

}

else
{

printf("Client-write() is OK...

");
}

if((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1)
{
    perror("recv()");
    exit(1);
}

else
    printf("Client-The recv() is OK...
");

buf[numbytes] = '\0';

printf("Client-Received: %s", buf);

printf("Client-Close() is OK! 

")

close(sockfd);

return 0;

}
3. Client2.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

// the port client will be connecting to
#define PORT 5000

// max number of bytes we can get at once
#define MAXDATASIZE 300

//BufferLength is 100 Bytes
#define BufferLength 100

int main(int argc, char *argv[])
{
    int sockfd, numbytes, rc;
    char buf[MAXDATASIZE];
    char buffer[BufferLength];
    struct hostent *he;
    char data[100] = "this is a test string from client :)!");
// connector’s address information
struct sockaddr_in their_addr;

// if no command line argument supplied
if(argc != 2)
{
    fprintf(stderr, "Client-Usage: %s the_client_hostname\n", argv[0]);
    // just exit
    exit(1);
}

// get the host information
if((he=gethostbyname(argv[1])) == NULL)
{
    perror("gethostbyname()");
    exit(1);
}

if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    perror("socket()");
    exit(1);
}
else
{
    printf("Client-The socket() is OK...\n");
    // host byte order
    their_addr.sin_family = AF_INET;
}
// short, network byte order
their_addr.sin_port = htons(PORT);
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
// zero the rest of the struct
memset(&their_addr.sin_zero, '\0', 8);

if((rc = connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1)
{
    perror("connect()");
    exit(1);
}
else
{
    printf("Client-The connect() is OK...

");
/*send string to server */
printf("Sending some string to the %s...
",argv[1]);
rc = write(sockfd,data,sizeof(data));
if(rc < 0)
{
    perror("Client-write() error");
    if (rc == 0)
    {
        perror("SO_ERROR was");
    }
}
close(sockfd);
exit(-1);

}

else
{
printf("Client-write() is OK...

");
}

if((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1)
{
 perror("recv()");
 exit(1);
}

else
{
 printf("Client-The recv() is OK...

");
    buf[numbytes] = '\0';
 printf("Client-Received: %s", buf);
 printf("Client-Close() is OK! \n\n");
 close(sockfd);
 return 0;
}

4. Client3.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

// the port client will be connecting to
#define PORT 5001

// max number of bytes we can get at once
#define MAXDATASIZE 300

// BufferLength is 100 Bytes
#define BufferLength 100

int main(int argc, char *argv[])
{
    int sockfd, numbytes, rc;
    char buf[MAXDATASIZE];
    char buffer[BufferLength];
    argv[2] = "1";
    struct hostent *he;
```
char data[100] = "this is a test string from client :)!";

// connector’s address information

struct sockaddr_in their_addr;

// if no command line argument supplied

if(argc != 2)
{
    fprintf(stderr, "Client-Usage: %s the_client_hostname\n\n", argv[0]);
    // just exit
    exit(1);
}

// get the host information

if((he = gethostbyname(argv[2]))==NULL)
{
    perror("gethostbyname()");
    exit(1);
}

if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    perror("socket()");
    exit(1);
}

else
{
    printf("Client-The socket() is OK...\n");

    // host byte order
their_addr.sin_family = AF_INET;

// short, network byte order

their_addr.sin_port = htons(PORT);

their_addr.sin_addr = *((struct in_addr *)he->h_addr);

// zero the rest of the struct

memset(&(their_addr.sin_zero), '\0', 8);

if((rc = connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1)
{
    perror("connect()");
    exit(1);
}

else

    printf("Client-The connect() is OK...\n");

/*send string to server */

printf("Sending some string to the %s...\n",argv[1]);

rc = write(sockfd,data,sizeof(data));

if(rc < 0)
{
    perror("Client-write() error");
    if (rc == 0)
    {
        perror("SO_ERROR was");
    }
}
close(sockfd);
exit(-1);
}

else
{
printf("Client-write() is OK..

");
}

if((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1)
{
    perror("recv()");
    exit(1);
}

else
{
    printf("Client-The recv() is OK...
");
    buf[numbytes] = '\0';
    printf("Client-Received: %s", buf);
    printf("Client-Close() is OK! 

");
    close(sockfd);
    return 0;
}

C. RSPEC

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
    <node client_id="client1" component_manager_id="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm" exclusive="true">
        <sliver_type name="raw-pc"/>
        <services>
            <execute shell="sh" command="sudo apt-get install dstat"/>
            <install url="http://nam.ece.upatras.gr/sites/default/files/files/clients.tar.gz" install_path="/local"/>
        </services>
        <jFed:location x="189.0" y="120.0"/>
        <jFed:nodeDescription>physical-node</jFed:nodeDescription>
        <interface client_id="client1:if0"/>
    </node>
    <node client_id="server1" component_manager_id="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm" exclusive="true">
        <sliver_type name="raw-pc"/>
        <services>
        </services>
        <jFed:location x="189.0" y="120.0"/>
        <jFed:nodeDescription>physical-node</jFed:nodeDescription>
        <interface client_id="server1:if0"/>
    </node>
</rspec>
<execute shell="sh" command="sudo apt-get install dstat"/>

<install url="http://nam.ece.upatras.gr/sites/default/files/files/clients.tar.gz"
install_path="/local"/>

</services>

<jFed:location x="520.0" y="201.0"/>

<jFed:nodeDescription>physical-node</jFed:nodeDescription>

@interface client_id="server1:if0"/>

</node>

<link client_id="link1">

<component_manager
name="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm"/>

<link_type name="lan"/>

<property source_id="client1:if0" dest_id="server1:if0" latency="2"
capacity="100000"/>

<property source_id="server1:if0" dest_id="client1:if0" latency="2"
capacity="100000"/>

@interface_ref client_id="client1:if0"/>

@interface_ref client_id="server1:if0"/>

</link>

<node client_id="client2"
component_manager_id="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm"
exclusive="true">

<sliver_type name="raw-pc"/>

</services>

<execute shell="sh" command="sudo apt-get install dstat"/>

<install url="http://nam.ece.upatras.gr/sites/default/files/files/clients.tar.gz"
install_path="/local"/>

</services>

<jFed:location x="189.0" y="120.0"/>
<jFed:nodeDescription>physical-node</jFed:nodeDescription>

<interface client_id="client2:if0"/>

</node>

<node client_id="server2" component_manager_id="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm" exclusive="true">
  <sliver_type name="raw-pc"/>
  
  <services>
    <execute shell="sh" command="sudo apt-get install dstat"/>
    <install url="http://nam.ece.upatras.gr/sites/default/files/files/clients.tar.gz" install_path="/local"/>
  </services>
  
  <jFed:location x="520.0" y="201.0"/>
  
  <jFed:nodeDescription>physical-node</jFed:nodeDescription>

  <interface client_id="server2:if0"/>

</node>

<link client_id="link2">
  <component_manager name="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm"/>
  
  <link_type name="lan"/>
  
  <property source_id="client2:if0" dest_id="server2:if0" latency="2" capacity="100000"/>
  
  <property source_id="server2:if0" dest_id="client2:if0" latency="2" capacity="100000"/>
  
  <interface_ref client_id="client2:if0"/>
  
  <interface_ref client_id="server2:if0"/>

</link>
<node client_id="client3"
component_manager_id="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm"
exclusive="true">
  <sliver_type name="raw-pc"/>
  <services>
    <execute shell="sh" command="sudo apt-get install dstat"/>
    <install url="http://nam.ece.upatras.gr/sites/default/files/files/clients.tar.gz"
      install_path="/local"/>
  </services>
  <jFed:location x="189.0" y="120.0"/>
  <jFed:nodeDescription>physical-node</jFed:nodeDescription>
  <interface client_id="client3:if0"/>
</node>

<node client_id="server3"
component_manager_id="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm"
exclusive="true">
  <sliver_type name="raw-pc"/>
  <services>
    <execute shell="sh" command="sudo apt-get install dstat"/>
    <install url="http://nam.ece.upatras.gr/sites/default/files/files/clients.tar.gz"
      install_path="/local"/>
  </services>
  <jFed:location x="520.0" y="201.0"/>
  <jFed:nodeDescription>physical-node</jFed:nodeDescription>
  <interface client_id="server3:if0"/>
</node>

<link client_id="link3">
  <component_manager
name="urn::publicid:IDN+wall1.ilabt.iminds.be+authority+cm"/>
</link>
<link_type name="lan"/>

<property source_id="client3:if0" dest_id="server3:if0" latency="2" capacity="100000"/>

<property source_id="server3:if0" dest_id="client3:if0" latency="2" capacity="100000"/>

<interface_ref client_id="client3:if0"/>

<interface_ref client_id="server3:if0"/>

</link>

<node client_id="client4" component_manager_id="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm" exclusive="true">
  <sliver_type name="raw-pc"/>
  <services>
    <execute shell="sh" command="sudo apt-get install dstat"/>
    <install url="http://nam.ece.upatras.gr/sites/default/files/files/clients.tar.gz" install_path="/local"/>
  </services>
  <jFed:location x="189.0" y="120.0"/>
  <jFed:nodeDescription>physical-node</jFed:nodeDescription>
  <interface client_id="client4:if0"/>
</node>

<node client_id="server4" component_manager_id="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm" exclusive="true">
  <sliver_type name="raw-pc"/>
  <services>
    <execute shell="sh" command="sudo apt-get install dstat"/>
    <install url="http://nam.ece.upatras.gr/sites/default/files/files/clients.tar.gz" install_path="/local"/>
  </services>
</node>
<node client_id="server5"
    component_manager_id="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm"
    exclusive="true">
    <sliver_type name="raw-pc"/>
    <services>
        <execute shell="sh" command="sudo apt-get install dstat"/>
        <install url="http://nam.ece.upatras.gr/sites/default/files/files/clients.tar.gz"
            install_path="/local"/>
    </services>
    <jFed:location x="520.0" y="201.0"/>
    <jFed:nodeDescription>physical-node</jFed:nodeDescription>
    <interface client_id="server5:if0"/>
</node>

<link client_id="link5">
    <component_manager
        name="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm"/>
    <link_type name="lan"/>
    <property source_id="client5:if0" dest_id="server5:if0" latency="2"
        capacity="100000"/>
    <property source_id="server5:if0" dest_id="client5:if0" latency="2"
        capacity="100000"/>
    <interface_ref client_id="client5:if0"/>
    <interface_ref client_id="server5:if0"/>
</link>

<jfed-ssh-keys:user-ssh-keys
    user="urn:publicid:IDN+wall2.ilabt.iminds.be+user+{username}">
    <jfed-ssh-keys:sshkey>{Key}</jfed-ssh-keys:sshkey>
</jfed-ssh-keys:user-ssh-keys></rspec>
D. API REFERENCES

Data Structures
- **sockaddr**
  Generic (non-address-family-specific) address structure

```c
#include <sys/socket.h> /* Generic socket header file */
struct sockaddr
{
  unsigned short sa_family; /* Address family (e.g., AF_INET) */
  char sa_data[14];          /* Protocol-specific address information */
};
```

- **sockaddr_in**
  Internet protocol address structure

```c
#include <netinet/in.h> /* Internet protocol socket specifics */
struct in_addr
{
  unsigned long s_addr; /* IP address (32 bits) */
};
struct sockaddr_in
{
  unsigned short sin_family; /* Internet protocol (AF_INET) */
  unsigned short sin_port; /* Address port (16 bits) */
  struct in_addr sin_addr; /* IP address (32 bits) */
  char sin_zero[8]; /* Not used */
};
```

All values must be in network byte order.

Socket Setup
- **socket()**
  Creates a TCP or UDP socket, which may then be used as an endpoint of communication for sending and receiving data using the specified protocol. Specify TCP and UDP with socket type / protocol pair SOCK_STREAM / IPPROTO_TCP and SOCK_DGRAM / IPPROTO_UDP, respectively.

```c
#include <sys/types.h>
#include <sys/socket.h>
int socket(int protocolFamily, int type, int protocol)
```

  - **protocolFamily:** Always PF_INET for TCP/IP sockets
  - **type:** Type of socket (SOCK_STREAM or SOCK_DGRAM)
  - **protocol:** Socket protocol (IPPROTO_TCP or IPPROTO_UDP)
Socket () returns the descriptor of the new socket if no error occurs and -1 otherwise.

- **Bind()**
  Assigns the local Internet address and port for a socket. The port number must be specified. The call will fail if the specified port number is the local port of some other socket.

```c
#include <sys/types.h>
#include <sys/socket.h>
int bind(int socket, struct sockaddr *localAddress, unsigned int addressLength)
```

- **Getsockname()**
  Returns the local information for a socket in a sockaddr structure. All multibyte fields in the structure are in network byte order.

```c
#include<sys/socket.h>
int getsockname(int socket, struct sockaddr *localAddress, unsigned int *addressLength )
```

Sockum connection
- **Connect()**
  Establishes a connection between the given socket and the remote socket associated with the foreign address, if any. Upon returning successfully, the given socket's local and remote IP address and port information have been filled in. If the socket was not previously bound to a local port, one is assigned randomly. For TCP sockets, connect() returns successfully only after completing a handshake with the remote TCP implementation; success implies the existence of a reliable channel to that socket.

```c
#include<sys/types.h>
#include<sys/socket.h>
int connect(int socket, Struct sockaddr *foreignAddress, int addressLength)
```
Connect() returns 0 if no error occurs and -1 otherwise.

- **Listen() (tcp streams socket only)**
  Indicates that the given socket is ready to accept incoming connections. The socket must already be associated with a local port (i.e., bind() must have been called previously). After this call, incoming TCP connection requests addressed to the given local will be completed and queued until they are passed to the program via accept().

```c
#include<sys/types.h>
#include<sys/socket.h>
int listen(int socket, int backlog)

  socket:    Socket (returned from socket ())
  backlog:  Maximum number of new connections (sockets) waiting
```

Listen() returns 0 if no error occurs and -1 otherwise.

- **Accept() (tcp stream socket only)**
  Blocks waiting for connections addressed to the IP address and port to which the given socket is bound. When a connection arrives and the TCP handshake is successfully completed, a new socket is returned. The local and remote address and port numbers of the new socket have been filled in with the local port number of the new socket, and the remote address information has been returned in the sockaddr_in structure (in network byte order).

```c
#include<sys/types.h>
#include<sys/socket.h>
int accept(int socket, struct sockaddr * clientAddress, int * addressLength )

  socket:                  Socket (listen() already called)
  clientAddress:     Originating socket IP address and port
  addressLength:   Length of sockaddr buffer (in), returned address (out)
```

Accept() returns the newly connected socket descriptor if no error occurs and -1 otherwise.

**Socket communication**

- **Send()**
  Sends the bytes contained in the buffer over the given socket. The socket must be in a connected state. When the call returns, the data has been queued for transmission over the connection. Semantics depends on the type of socket. For a stream socket, the data will eventually be transmitted, provided the connection closes normally. For a
A datagram socket, there are no guarantees of delivery. However, the data from a single send() call will never be split across multiple recv() calls. The return value indicates the number of bytes actually transmitted. The flags argument allows various special protocol features, such as out-of-band data, to be accessed.

```c
#include<sys/types.h>
#include<sys/socket.h>
int send(int socket, const void *msg, unsigned int msgLength, int flags)
```

- **socket:** Socket (must be in connected state)
- **msg:** Pointer to data to be transmitted
- **msgLength:** Number of bytes to be sent
- **flags:** Control flags (0 in most cases)

Send() returns the number of bytes sent if no error occurs and -1 otherwise.

- **Sendto()**

Sends the bytes contained in the buffer over the given socket. To use sendto() on a TCP socket requires the socket to be in the connected state. The semantics for sendto() are the same as the semantics of send().

```c
#include<sys/types.h>
#include<sys/socket.h>
int sendto(int socket, char *msg, int msgLength, int flags, struct sockaddr *destAddr, int destAddrLen)
```

- **socket:** Socket (must be in connected state)
- **msg:** Pointer to data to be transmitted
- **msgLength:** Number of bytes to be sent
- **flags:** Control flags (0 in most cases)
- **destAddr:** Destination address for data (network byte order)
- **destAddrLen:** Length of destination address structure

Sendto() returns the number of bytes sent if no error occurs and -1 otherwise.

- **Recv()**

Copies up to a specified number of bytes, received on the socket, into a specified location. The given socket must be in the connected state. Normally, the call blocks until either at least one byte is returned or the connection closes. The return value indicates the number of bytes actually copied into the buffer starting at the pointed-to location. Semantics depends on the type of socket. For a stream socket, the bytes are delivered in the same order as they were transmitted, without omissions. For a datagram socket, each recv() returns the data from at most one send(), and order is not necessarily preserved. If the buffer provided to recv() is not big enough for the next available datagram, the datagram is silently truncated to the size of the buffer.
#include<sys/types.h>
#include<sys/socket.h>

int recv(int socket, void *rcvBuffer, int bufferLength, int flags)

- **socket:** Socket (must be in connected state)
- **rcvBuffer:** Where to put the data
- **bufferLength:** Maximum number of bytes to put in buffer
- **flags:** Control flags, 0 in most cases

recv() returns the number of bytes received if no error occurs and -1 otherwise.

- **Recvfrom()**
Copies up to a specified number of bytes, received on the socket, into a specified location. To use recvfrom() on a TCP socket requires the socket to be in the connected state. The semantics for recvfrom() are the same as the semantics of recv().

#include<sys/types.h>
#include<sys/socket.h>

int recvfrom(int socket, char *buffer, int bufferSize, int flags, struct sockaddr *fromAddr, unsigned int *fromAddrLen)

- **socket:** Socket (must be in connected state)
- **buffer:** Where to put the data
- **bufferSize:** Max number of bytes to put in buffer
- **flags:** Control flags (0 in most cases)
- **fromAddr:** Address data of sender (network byte order)
- **fromAddrLen:** Length of sender address structure

recv() returns the number of bytes received if no error occurs and -1 otherwise.

- **Close()**
Terminates communication on a socket. The socket is marked to disallow further sends and receives.

#include<unistd.h>

int close(int socket)

- **socket:** Socket (must be in connected state)

Close() returns 0 if no error occurs and -1 otherwise.

- **Shutdown()**
Terminates communication on a socket. The socket is marked to disallow further sends, receives, or both, according to the second parameter: If it is 0, further receives will be disallowed. If it is 1, further sends are disallowed. If it is 2, both sends and receives will be disallowed. The socket must be in the connected state.
#include <sys/socket.h>
int shutdown(int socket, int how)  

- socket: Socket (must be in connected state)
- how: 0 = done receiving, 1 = done sending, 2 = done sending and receiving

shutdown() returns 0 if no error occurs and -1 otherwise.

**Binary/String conversion**
- **Inet_ntoa()**
  Converts an IP address in binary notation (network byte order) to the corresponding string in dotted notation (e.g., "169.1.1.2").

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr address)
```

- address: Structure containing 32-bit representation of IP address

inet_ntoa() returns a pointer to a string. The string is a statically allocated buffer whose value changes with every call; therefore, the buffer should be copied before subsequent calls.

- **Inet_addr()**
  Converts an IP address in dotted notation to the corresponding binary notation (network byte order).

```c
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
char *inet_addr(const char *address)
```

- address: Pointer to character string containing dotted representation of IP address

inet_addr() returns the unsigned long, binary representation of the IP address if no error occurs and -1 otherwise.

- **Htons(), htonl(), ntohs(), ntohl()**
  Converts host to network byte order and vice versa.

```c
#include<netinet>
short int htons(short int hostShort)
long int htonl(long int hostLong)
```
short int ntohs(short int netShort)
long int ntohl(long int netLong)

  hostShort: Short integer in host byte order
  hostLong: Long integer in host byte order
  netShort: Short integer in network byte order
  netLong: Long integer in network byte order

htons(), htonl(), ntohs(), and ntohl() return the converted value. These functions have no failure return value.

**Host and service information**

- **Gethostname()**
  gethostname() returns the local host name in the specified buffer.

  int gethostname(char *hostName, unsigned int length)

  **hostname**: Buffer to hold the host name
  **length**: Length of hostName buffer

  gethostname() returns -1 for failure; 0 otherwise.

- **Gethostbyname()**
  Given the name of a host, gethostbyname() returns a hostent structure containing a description of the named host.

  ```c
  #include<netdb.h>
  struct hostent *gethostbyname (const char * hostName)
  ```

  **hostName**: Name of host to get information about

  gethostbyname() returns NULL on error and a pointer to a hostent structure on success:

  ```c
  struct hostent
  {  
    char *h_name; /* Official name of host */
    char **h_aliases; /* List of alias names (strings) */
    int h_addrtype; /* Type of host address (AF_INET) */
    int h_length; /* Address length */
    char **h_addr_list; /* List of addresses (binary in network byte order) */
  }
  ```

  If hostName is an IP address instead of a name, gethostbyname() copies the string to h_name and places an entry for the IP address (binary) in h_addr_list.

- **Gethostbyaddr()**
  Given an IP address, gethostbyaddr() returns a hostent structure containing a description of the host with the given IP address.
#include<netdb.h>
#include<sys/socket.h>

struct hostent *gethostbyaddr(const char *address, int addressLength, int addressFamily)

  address: Address (in binary, network-byte ordered representation) of host to get information about
  addressLength: Length of given address (in bytes)
  addressFamily: Family of given address (AF_INET)

gethostbyaddr() returns NULL on error and a hostent structure on success.


About Moodle, https://docs.moodle.org/en/About_Moodle


About Dokeos, http://www.dokeos.com

About Atutor, http://www.atutor.ca/

About SAKAI, https://sakaiproject.org/


