Course code	Course Name	L-T-P -Credits	Year of Introduction
RLMCA232	System Design Lab	0-0-4-	2016

Course Objectives
To Introduce Shell Scripting.
To sensitize the need for Version control
To do network programming using Socket Programs.

Syllabus
Shell Scripts - GIT - Socket Programming

Expected Outcome
The students will be able to
i. Develop Shell Programs for system administration
ii. Use GIT and gain knowledge in using version control
iii. Develop programs for client- server communications using various network protocols(TCPIUDP)

References
1. B. M. Harwani, Unix and Shell programming", Oxford University Press(2013)
2. James F Kurose and Keith W Ross, "Computer Networking: A Top - Down Approach", Pearson Education; 5 th Edition (2012).
3. Richard Stevens, "UNIX Network Programming : Inter process Communications", Prentice Hall, Second Edition
4. Richard Stevens, "UNIX Network Programming,: Networking APIs: Sockets and XTI", Prentice Hall, Second Edition.
5. Travis Swicegood, " Pragmatic Guide to Git", Pragmatic Bookshelf. Pub. Date: November 15, 2010

Experiments/Exercises

Administration Level
Introduction to Shell scripting - Experiment with shell scripts mainly for administrative tasks like user creation in bulk, changing file permissions recursively, creating files in bulk, deleting folders and sub folders etc…

1. Commands
1. echo, read
2. more, less
3. man
4. chmod, chown
5. cd, mkdir, pwd, ls, find
6. cat, mv, cp, rm
7. wc, cut, paste
8. head, tail, grep, expr
9. Redirections & Piping
10. useradd, usermod, userdel, passwd
11. tar

2. Scripting
    1. Environment variables

2. If statement
3. For statement
4. While statement

3. Remote access
1. ssh, scp, ssh-keygen, ssh-copy-id

4. Scheduling Using cron and at

Experiments to supplement RLMCA202 - Application Development and Maintenance

GIT

1. git init - Initializing an empty git repository git init --bare
2. git status - Knowing the status of your repository
3. git add <artifact> StaginglAdding artifacts(files) to repository {3.1 git status}*
4. git commit -m "Message for the commit" - Details on how to commit changes to the local repository
5. git add <pattern> - Bulk addinglstaging artifacts to repository {5.1 git commit -m "Commit the changes"}*
6. git log - Git Activity logs
7. git remote add <origin_name> <remote repository URL>
- Attaching a remote repository <remote repository URL> - username@127.0.0.1:lpathltolrepository
8. git push -u <origin_name> <branch> (git push -u origin master) -Pushing to the master
9. git pull origin master - Pulling from a master
10. git diff options
11. resetting a stagedladded file.
12. git checkout
13. git branch <branch_name> - Creating the branches
14. git checkout <branch_name> - switching between branches
15. git rm 'pattern' - removing the fileslartifacts
1 commit to the branch*
2 Switch back to the master*
16. git merge <branch_name> - Merging the contents.
17. git branch -d <branch_name> - Removing a branch
18. git push - Syncing with the remote repository
19. git stash - Park your changes in directory.
20. git stash apply - Applying the changes back (git stash options)
21. git rebase - Reapply commits on top of another base tip

* Steps that are repeated for completing the exercise

Students should be encouraged to do all the subsequent experiments in a GIT repository.

Network Programming (Java/C)

1. Implement Bidirectional Client-Server communication using TCP.
2. Implement Bidirectional Client-Server communication using UDP.
3. Implement Echo Server using TCP
4. Implement Chat Server using UDP.

CONTENTS

Cycle - 1
Linux Commands & Shell Scripting

Introduction: The Linux terminal is an extremely powerful tool that goes well beyond the GUI. The basic Linux commands are universal, and they should work on any past and future distribution of Linux. They will even work with non-Linux operating systems based on Unix, such as FreeBSD or the Mac OS X terminal. There is not a single Linux distribution that doesn't have a terminal of some kind. On the contrary, there are a few distributions that don't have a GUI by default, and everything is done on the command line.

This is basic anatomy of most Linux commands:

[sudo] command [optional switch] [file or directory path]

Using sudo will run any command with administrative rights. Most Linux commands that have to deal with system files and installation/uninstallation of programs demand sudo.

1.1 Linux Commands

1. The linux commands are case-sensitive and it is important to remember that the command is "sudo", neither "Sudo", "SUDO", nor "sUdO".

2. Most Linux commands are lowercase, but there are capitalized switches, like "chown -R".

3. The file and directory names are also case-sensitive. "File1" and "file1" are different files.

4. Pressing the Up keyboard key will cycle through the last Linux commands we successfully used, in order. We can also use the history command to see all the Linux commands we have ever used on the terminal.

5. The Tab button on the keyboard will automatically fill in the names of files and directories.

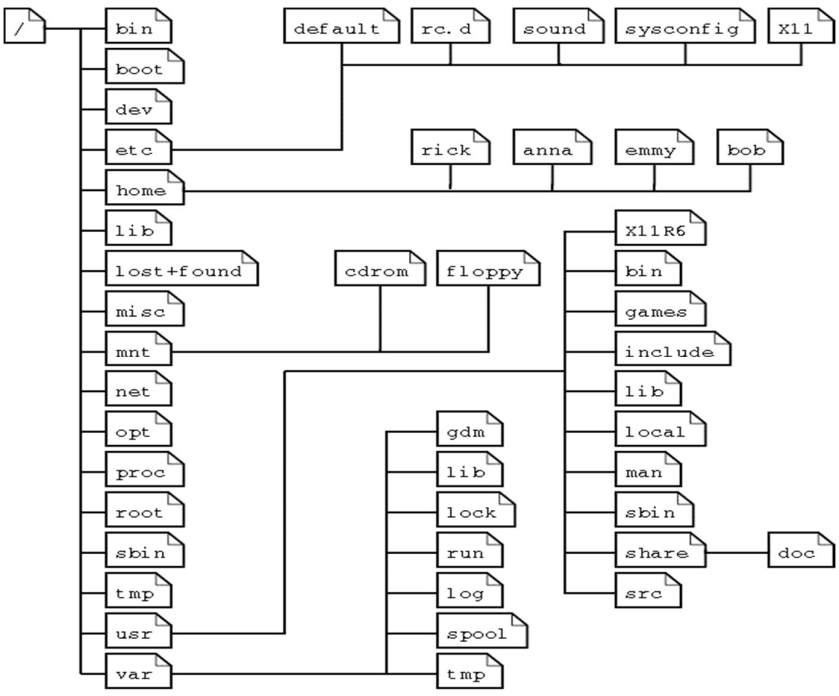
6. While using Linux commands, the characters "?" and "*" are wildcards. "?" will replace any single character. So, if we have two files names test1file and test2file, we can delete them both with "rm test?file". But this won't delete test12file.

7. We can learn more about any of the Linux commands with [command] --help and man [command]. [command] --help will show the usage of the command, and the available options and switches.

Basic Commands

cd Changes a shell's current working directory cp
Copies files
find Searches files matching certain given criteria less
Displays texts (such as manual pages) by page ln
Creates ("hard" or symbolic) links
locate Finds files by name in a file name
database ls Lists file information or directory
contents mkdir Creates new directories
more Displays text data by page
mv Moves files to different directories or renames them
pwd Displays the name of the current working directory
rm Removes files or directories
rmdir Removes (empty) directories
slocate Searches file by name in a file name database
updatedb Creates the file name database for locate
xargs Constructs command lines from its standard input

1.2 Linux File Hierarchy



Directory	Content
/bin	Common programs, shared by the system, the system administrator and the users.
/boot	The startup files and the kernel, vmlinuz. In some recent distributions also grub data. Grub is the GRand Unified Boot loader and is an attempt to get rid of the many different boot-loaders we know today.
/dev	Contains references to all the CPU peripheral hardware, which are represented as files with special properties.
/etc	Most important system configuration files are in /etc, this directory contains data similar to those in the Control Panel in Windows
/home	Home directories of the common users.
/initrd	(on some distributions) Information for booting. Do not remove!
/lib	Library files, includes files for all kinds of programs needed by the system and the users.
/lost+found	Every partition has a lost+found in its upper directory. Files that were saved during failures are here.
/misc	For miscellaneous purposes.
/mnt	Standard mount point for external file systems, e.g. a CD-ROM or a digital camera.
/net	Standard mount point for entire remote file systems
/opt	Typically contains extra and third party software.
/proc	A virtual file system containing information about system resources. More information about the meaning of the files in proc is obtained by entering the command **man *proc*** in a terminal window. The fileproc.txt discusses the virtual file system in detail.
/root	The administrative user's home directory. Mind the difference between /, the root directory and /root, the home directory of the *root* user.
/sbin	Programs for use by the system and the system administrator.
/tmp	Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work!
/usr	Programs, libraries, documentation etc. for all user-related programs.
/var	Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

/root	The administrative user's home directory. Mind the difference between /, the root directory and /root, the home directory of the *root* user.
/sbin	Programs for use by the system and the system administrator.
/tmp	Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work!
/usr	Programs, libraries, documentation etc. for all user-related programs.
/var	Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

1.3 File & Directory Commands

- The tilde (~) symbol stands for your home directory. If you are user, then the tilde (~) stands for /home/user
- **pwd**: The **pwd** command will allow you to know in which directory you're located (**pwd** stands for "print working directory"). Example:**"pwd"** in the Desktop directory will show "~/Desktop". Note that the GNOME Terminal also displays this information in the title bar of its window. A useful gnemonic is "present working directory."
- **ls**: The **ls** command will show you ('list') the files in your current directory. Used with certain options, you can see sizes of files, when files were made, and permissions of files. Example: **"ls ~"** will show you the files that are in your home directory.
- **cd**: The **cd** command will allow you to change directories. When you open a terminal you will be in your home directory. To move around the file system you will use **cd**. Examples:
    - To navigate into the root directory, use **"cd /"**
    - To navigate to your home directory, use **"cd"** or **"cd ~"**
    - To navigate up one directory level, use **"cd .."**
    - To navigate to the previous directory (or back), use **"cd -"**

To navigate through multiple levels of directory at once, specify the full directory path that you want to go to. For example, use, **"cd /var/www"** to go directly to the /www subdirectory of /var/. As another example, **"cd ~/Desktop"** will move you to the Desktop subdirectory inside your home directory.

- **cp**: The **cp** command will make a copy of a file for you. Example: **"cp file foo"** will make an exact copy of "file" and name it "foo", but the file "file" will still be there. If you are copying a directory, you must use **"cp -r directory foo"** (copy recursively). (To understand what "recursively" means, think of it this way: to copy the directory and all its files and subdirectories and all their files and subdirectories of the subdirectories and all their files, and on and on, "recursively")
- **mv**: The **mv** command will move a file to a different location or will rename a file. Examples are as follows: **"mv file foo"** will rename the file "file" to "foo". **"mv foo ~/Desktop"** will move the file "foo" to your Desktop directory, but it will not rename it. You must specify a new file name to rename a file.
    - To save on typing, you can substitute '~' in place of the home directory.
    - Note that if you are using **mv** with **sudo** you can use the ~ shortcut, because the terminal expands the ~ to your home directory. However, when you open a root shell with **sudo -i** or **sudo -s**, ~ will refer to the root account's home directory, not your own.
- **rm**: Use this command to remove or delete a file in your directory.
- **rmdir**: The **rmdir** command will delete an empty directory. To delete a directory and all of its contents recursively, use **rm -r** instead.
- **mkdir**: The **mkdir** command will allow you to create directories. Example: **"mkdir music"** will create a directory called "music".
- **man**: The **man** command is used to show you the manual of other commands. Try **"man man"** to get the man page for **man** itself. See the "**Man** & Getting Help" section down the page for more information.
- **sudo**: The **sudo** command is used to perform file operations on files that the **Root**

1.4 File Permissions

In Linux and Unix, everything is a file. Directories are files, files are files and devices are files. Devices are usually referred to as a node; however, they are still files. All of the files on a system have permissions that allow or prevent others from viewing, modifying or executing. If the file is of type Directory then it restricts different actions than files and device nodes. The super user "root" has the ability to access any file on the system. Each file has access restrictions with permissions, user restrictions with owner/group association. Permissions are referred to as bits.

To change or edit files that are owned by root, sudo must be used

If the owner read & execute bit are on, then the permissions are:

-r-x-

There are three types of access restrictions:

Permission	**Action**	**chmod option**
Read	(view)	r or 4
Write	(edit)	w or 2
execute	(execute)	x or 1

There are also three types of user restrictions:

User	**ls output**
owner	-rwx------
group	----rwx---
other	-------rwx

Note: The restriction type scope is not inheritable: the file owner will be unaffected by restrictions set for his group or everybody else.

Folder/Directory Permissions

Directories have directory permissions. The directory permissions restrict different actions than with files or device nodes.

Permission	Action	Chmod option
Read	(view contents, i.e. ls command)	r or 4
Write	(create or remove files from dir)	w or 2
execute	(cd into directory)	x or 1

- read restricts or allows viewing the directories contents, i.e. ls command
- write restricts or allows creating new files or deleting files in the directory. (Caution: write access for a directory allows deleting of files in the directory even if the user does not have write permissions for the file!)
- execute restricts or allows changing into the directory, i.e. cd command

Folders (directories) must have 'execute' permissions set (x or 1), or folders (directories) will NOT FUNCTION as folders (directories) and WILL DISAPPEAR from view in the file browser (Nautilus).

Permissions in Action

mca123@mca/home/user$ ls -l /etc/hosts
-rw-r--r-- 1 root root 288 2005-11-13 19:24
/etc/hosts mca123@mca/home/user$

Using the example above we have the file "/etc/hosts" which is owned by the user root and belongs to the root group.

What are the permissions from the above /etc/hosts ls output?

-rw-r--r--owner = Read & Write (rw-)
group = Read (r--)
other = Read (r--)

1.5 Changing Permissions

The command to use when modifying permissions is chmod. There are two ways to modify permissions, with numbers or with letters. Using letters is easier to understand for most people. When modifying permissions be careful not to create security problems. Some files are configured to have very restrictive permissions to prevent unauthorized access. For example, the /etc/shadow file (file that stores all local user passwords) does not have permissions for regular users to read or otherwise access.

mca123@mca/home/user# ls -l /etc/shadow
-rw-r----- 1 root shadow 869 2005-11-08 13:16 /etc/shadow mca123@mca/home/user#

Permissions:
owner = Read & Write (rw-)
group = Read (r--)
other = None (---)

Ownership:
owner = root
group = shadow

chmod with Letters

Usage: chmod {options} filename

Options	Definition
U	owner
G	group
O	other
A	all (same as ugo)

X	execute
W	write
R	read
+	add permission
-	remove permission
=	set permission

Here are a few examples of chmod usage with letters (try these out on your system).

First create some empty files:

mca123@mca/home/user$ touch file1 file2 file3 file4 mca123@mca/home/user$ **ls -l**
total 0
-rw-r--r-- 1 user user 0 Nov 19 20:13 file1
-rw-r--r-- 1 user user 0 Nov 19 20:13 file2
-rw-r--r-- 1 user user 0 Nov 19 20:13 file3
-rw-r--r-- 1 user user 0 Nov 19 20:13 file4

Add owner execute bit:

mca123@mca/home/user$ chmod u+x file1 mca123@mca/home/user$ **ls -l file1**
-rwxr--r-- 1 user user 0 Nov 19 20:13 file1

Add other write & execute bit:

mca123@mca/home/user$ chmod o+wx file2 mca123@mca/home/user$ **ls -l file2**
-rw-r--rwx 1 user user 0 Nov 19 20:13 file2

Remove group read bit:

mca123@mca/home/user$ chmod g-r file3 mca123@mca/home/user$ **ls -l file3**
-rw----r-- 1 user user 0 Nov 19 20:13 file3

Add read, write and execute to everyone:
mca123@mca/home/user$ chmod ugo+rwx file4 mca123@mca/home/user$ **ls -l file4**
-rwxrwxrwx 1 user user 0 Nov 19 20:13 file4 mca123@mca/home/user$

chmod with Numbers

Usage: chmod {options} filename

Options	Definition
#--	owner
-#-	group
--#	other
1	execute
2	write
4	read

Owner, Group and Other is represented by three numbers. To get the value for the options determine the type of access needed for the file then add.

For example if you want a file that has -rw-rw-rwx permissions you will use the following:

Owner	Group	Other
read & write	read & write	read, write & execute
4+2=6	4+2=6	4+2+1=7

mca123@mca/home/user$ chmod 667 filename

Another example if you want a file that has --w-r-x--x permissions you will use the following:

Owner	Group	Other
write	Read & execute	execute
2	4+1=5	1

mca123@mca/home/user$ chmod 251 filename
Here are a few examples of chmod usage with numbers (try these out on your system). First create some empty files:

mca123@mca/home/user$ touch file1 file2 file3 file4 mca123@mca/home/user$ **ls -l**
total 0
-rw-r--r-- 1 user user 0 Nov 19 20:13 file1
-rw-r--r-- 1 user user 0 Nov 19 20:13 file2
-rw-r--r-- 1 user user 0 Nov 19 20:13 file3
-rw-r--r-- 1 user user 0 Nov 19 20:13 file4

Add owner execute bit:

mca123@mca/home/user$ chmod 744 file1 mca123@mca/home/user$ **ls -l file1**
-rwxr--r-- 1 user user 0 Nov 19 20:13 file1

Add other write & execute bit:

mca123@mca/home/user$ chmod 647 file2 mca123@mca/home/user$ **ls -l file2**
-rw-r--rwx 1 user user 0 Nov 19 20:13 file2

Remove group read bit:

mca123@mca/home/user$ chmod 604 file3 mca123@mca/home/user$ **ls -l file3**
-rw----r-- 1 user user 0 Nov 19 20:13 file3

Add read, write and execute to everyone:

mca123@mca/home/user$ chmod 777 file4 mca123@mca/home/user$ **ls -l file4**
-rwxrwxrwx 1 user user 0 Nov 19 20:13 file4 mca123@mca/home/user$

chmod with sudo

Changing permissions on files that you do not have ownership of: (Note that changing permissions the wrong way on the wrong files can quickly mess up your system a great deal! Please be careful when using sudo!)

mca123@mca/home/user$ ls -l /usr/local/bin/somefile

-rw-r--r-- 1 root root 550 2005-11-13 19:45 /usr/local/bin/somefile mca123@mca/home/user$

mca123@mca/home/user$ sudo chmod o+x /usr/local/bin/somefile mca123@mca/home/user$ **ls -l /usr/local/bin/somefile**

-rw-r--r-x 1 root root 550 2005-11-13 19:45 /usr/local/bin/somefile

mca123@mca/home/user$

Recursive Permission Changes

To change the permissions of multiple files and directories with one command. Please note the warning in the chmod with sudo section and the Warning with Recursive chmod section.

Recursive chmod with -R and sudo

To change all the permissions of each file and folder under a specified directory at once, use sudo chmod with -R

mca123@mca/home/user$ sudo chmod 777 -R /path/to/someDirectory mca123@mca/home/user$ **ls -l**

total 3

-rwxrwxrwx 1 user user 0 Nov 19 20:13 file1

drwxrwxrwx 2 user user 4096 Nov 19 20:13 folder

-rwxrwxrwx 1 user user 0 Nov 19 20:13 file2

Recursive chmod using find, pipemill, and sudo

To assign reasonably secure permissions to files and folders/directories, it's common to give files a permission of 644, and directories a 755 permission, since chmod -R assigns to both. Use sudo, the find command, and a pipemill to chmod as in the following examples.

To change permission of only files under a specified directory.

mca123@mca/home/user$ sudo find /path/to/someDirectory -type f -print0 | xargs -0 sudo chmod

6

4

4

mca123@mca/home/user$ ls -l total 3

-rw-r--r-- 1 user user 0 Nov 19 20:13 file1 drwxrwxrwx 2 user user 4096 Nov 19 20:13 folder

-rw-r--r-- 1 user user 0 Nov 19 20:13 file2

To change permission of only directories under a specified directory (including that directory):

mca123@mca/home/user$ sudo find /path/to/someDirectory -type d -print0 | xargs -0 sudo chmod

7

5

5

mca123@mca/home/user$ ls -l total 3

-rw-r--r-- 1 user user 0 Nov 19 20:13 file1

drwxr-xr-x 2 user user 4096 Nov 19 20:13 folder

-rw-r--r-- 1 user user 0 Nov 19 20:13 file2

Warning with Recursive chmod

WARNING: Although it's been said, it's worth mentioning in context of a gotcha typo. Please note, Recursively deleting or chown-ing files are extremely dangerous. You will not be the first, nor the last, person to add one too many spaces into the command. This example will hose your system:

mca123@mca/home/user$ sudo chmod -R / home/john/Desktop/tempfiles

Note the space between the first / and home. You have been warned.

Changing the File Owner and Group

A file's owner can be changed using the chown command. For example, to change the foobar file's owner to tux:

mca123@mca/home/user$ sudo chown tux foobar

To change the foobar file's group to penguins, you could use either chgrp or chown with special syntax:

mca123@mca/home/user$ sudo chgrp penguins foobar
mca123@mca/home/user$ sudo chown :penguins foobar

Finally, to change the foobar file's owner to tux and the group to penguins with a single command, the syntax would be:

mca123@mca/home/user$ sudo chown tux:penguins foobar

1.6 User Management

User management is a critical part of maintaining a secure system. Ineffective user and privilege management often lead many systems into being compromised. Therefore, it is important that you understand how you can protect your server through simple and effective user account management techniques.

Where is root?

Ubuntu developers made a conscientious decision to disable the administrative root account by default in all Ubuntu installations. This does not mean that the root account has been deleted or that it may not be accessed. It merely has been given a password which matches no possible encrypted value, therefore may not log in directly by itself.

Instead, users are encouraged to make use of a tool by the name of *sudo* to carry out system administrative duties. *Sudo* allows an authorized user to temporarily elevate their privileges using their own password instead of having to know the password belonging to the root account. This simple yet effective methodology provides accountability for all user actions, and gives the administrator granular control over which actions a user can perform with said privileges.

If for some reason you wish to enable the root account, simply give it a password:

sudo passwd

Sudo will prompt you for your password, and then ask you to supply a new password for root as shown below:

[sudo] password for username: **(enter your own password)**
Enter new UNIX password: **(enter a new password for root)**

Retype new UNIX password: **(repeat new password for root)**
passwd: **password updated successfully**

To disable the root account, use the following passwd syntax:

sudo passwd -l root

You should read more on *Sudo*by checking out it's man page:

man sudo

By default, the initial user created by the Ubuntu installer is a member of the group "admin" which is added to the file /etc/sudoers as an authorized sudo user. If you wish to give any other account full root access through *sudo*, simply add them to the admin group.

Adding and Deleting Users

The process for managing local users and groups is straight forward and differs very little from most other GNU/Linux operating systems. Ubuntu and other Debian based distributions, encourage the use of the "adduser" package for account management.

- To add a user account, use the following syntax, and follow the prompts to give the account a password and identifiable characteristics such as a full name, phone number, etc.

  sudo adduser username

- To delete a user account and its primary group, use the following syntax:

  sudo deluser username

Deleting an account does not remove their respective home folder. It is up to you whether or not you wish to delete the folder manually or keep it according to your desired retention policies.

Remember, any user added later on with the same UID/GID as the previous owner will now have access to this folder if you have not taken the necessary precautions.

You may want to change these UID/GID values to something more appropriate, such as the root account, and perhaps even relocate the folder to avoid future conflicts:

```
sudo chown -R root:root /home/username/
sudo mkdir /home/archived_users/
sudo mv /home/username /home/archived_users/
```

- To temporarily lock or unlock a user account, use the following syntax, respectively:

  sudo passwd -l username sudo passwd -u username

- To add or delete a personalized group, use the following syntax, respectively:

  sudo addgroup groupname sudo delgroup groupname

- To add a user to a group, use the following syntax:

  sudo adduser username groupname

1.7 GREP

Grep stands for "global search for regular expressions and print". The power of grep lies in using regular expressions mostly. Grep is a command-line tool that allows you to find a string in a file or stream. It can be used with a regular expression to be more flexible at finding strings.

For more information enter: **man grep** in a terminal.

How To Use grep

In the simplest case grep can be invoked as follows:

grep [options] pattern [filename]

The above command searches the file for STRING and lists the lines that contain a match. This is OK but it does not show the true power of grep. The above command only looks at one file. A cool example of using grep with multiple files would be to find all lines in all files in a given directory that contain the name of a person. This can be easily accomplished as follows:

grep 'Abdul Kalam' *

The above command searches all files in the current directory for the name and lists all lines that contain a match.

Notice the use of quotes in the above command. Quotes are not usually essential, but in this example they are essential because the name contains a space. Double quotes could also have been used in this example.

grep provides various options.

Option	Description
-b	Display the block number at the beginning of each line.
-c	Display the number of matched lines.
-h	Display the matched lines, but do not display the filenames.
-i	Ignore case sensitivity.
-l	Display the filenames, but do not display the matched lines.
-n	Display the matched lines and their line numbers.
-s	Silent mode.

-v Display all lines that do NOT match.

-w Match whole word.

Regular Expressions

grep can search for complicated patterns to find what you need.

Here is a list of some of the special characters used to create a regular expression:

^ Denotes the beginning of a line

$ Denotes the end of a line

. Matches any single character

***** The preceding item in the regular expression will be matched zero or more times

[ ] A bracket expression is a list of characters enclosed by [and]. It matches any single character in that list; if the first character of the list is the caret ^ then it matches any character not in the list

\ < Denotes the beginning of a word

\ > Denotes the end of a word

Here is an example of a regular expression search:

grep "\<[A-Za-z].*\>" file

This regular expression matches any "word" that begins with a letter (upper or lower case). For example, "words" that begin with a digit would not match. The grep command lists the lines that contain a match.

Example:

1. Write a shell script to count occurences of a word in a file using grep.

echo “Enter a word”

read word

echo —Enter the filename‖ read file

nol=grep -c $word $file

echo — $nol times $word present in the $file‖

Shell Programming

Aim: To learn how to write shell scripts

Introduction: The most generic sense of the term shell means any program that users employ to type commands. A shell hides the details of the underlying operating system and manages the technical details of the operating system kernel interface, which is the lowest-level, or "inner-most" component of most operating systems. In Unix-like operating systems, users typically have many choices of command-line interpreters for interactive sessions. When a user logs in to the system interactively, a shell program is automatically executed for the duration of the session. The Unix shell is both an interactive command language as well as a scripting programming language, and is used by the operating system as the facility to control (shell script) the execution of the system.

Bash

Bash is the shell, or command language interpreter, for the GNU operating system. The name is an acronym for the 'Bourne-Again SHell'. While the GNU operating system provides other shells, including a version of csh, Bash is the default shell. Like other GNU software, Bash is quite portable. It currently runs on nearly every version of Unix and a few other operating systems - independently-supported ports exist for MS-DOS, OS/2, and Windows platforms.

Shell Scripting

Shell Script is series of command written in plain text file designed to be run by the Unix shell, a command-line interpreter. The various dialects of shell scripts are considered to be scripting languages. Typical operations performed by shell scripts include file manipulation, program execution, and printing text.

NOTE: The commands given in the scripting section are to be put into the text editor and not in the terminal unless instructed otherwise.

Writing the first script, the "Hello World" script.

You can create a bash script by opening your favorite text editor to edit your script and then saving it (typically the .sh file extension is used for your reference, but is not necessary. In our examples, we will be using the .sh extension).

#!/bin/bash
echo "Hello, World"

The first line of the script just defines which interpreter to use. **NOTE:** There is no leading whitespace before #!/bin/bash.
To run a bash script you first have to have the correct file permissions.
We do this with **chmod** command in terminal as follows:

chmod a+x hello_world.sh #Gives everyone execute permissions
OR
chmod 700 hello_world.sh #Gives read, write, execute permissions to the Owner

This will give the file the appropriate permissions so that it can be executed.
To run the script from where it is stored, type the follwing in Terminal:

./hello_world.sh

The output on the screen is Hello, World.

2.1 Shell script to show various parameters related to the currently logged in user

1. logged user and his login name	***x=$(logname)*** ***echo "Currently logged user and his logname : $x"***
2. Your current shell	***echo $SHELL***
3. Your home directory	***echo $HOME***
4. Your operating system type	***x=$(arch)*** ***echo "Your operating system type : $x"***
5. Your current path setting	***echo $PATH***
6. Your current working directory	***echo $pwd***
7. Show Currently logged number of users	***echo $users***

2.2. Shell script to show various system configuration like

1. About your OS and version, release number, kernel version:	***x=$(lsb_release -a)***
2. Show all available shells:	***cat /etc/shells***
3. Show mouse settings:	***echo $(xinput --list --short)***
4. Show computer CPU information like processor type, speed etc.:	***echo $(lscpu)***
5. Show memory information:	***echo $(free -m)***
6. Show hard disk information like size of hard-disk, cache memory, model etc.:	***echo $(sudo dmidecode -t memory)***
7. File system (Mounted):	***echo $(sudo fdisk -l)***

Special shell commands

$0	The filename of the current shell
$1 - $9	Command line argument ID
$#	Number of arguments supplied to a script
$?	Exit status of the last command
$$	The process ID of the current shell
$!	Process id of the last background command

2.3. Shell scripting using control structures and looping

Shell scripts can be used to make decisions and perform different actions depending on conditions.

The shell provides several commands that we can use to control the flow of execution in our program. These include:

- if
- exit
- for
- while
- until
- case
- break
- continue

if

The if command is fairly simple; it makes a decision based on a condition. The if command has three forms:

1	***if condition ; then*** ***commands*** ***fi***	If the condition is true, then commands are performed. If the condition is false, nothing is done.

2	***if condition ; then***
	commands
	else
	commands
	fi
		If the condition is true, then the first set of commands is performed.
		
		If the condition is false, the second set of commands is performed.

3	***if condition ; then*** ***commands*** ***elif condition ; then*** ***commands*** ***fi***	If the condition is true, then the first set of commands is performed. If the condition is false, and if the second condition is true, then the second set of commands is performed.

exit

In order to be good script writers, we must set the exit status when our scripts finish. To do this, use the exit command. The exit command causes the script to terminate immediately and set the exit status to whatever value is given as an argument. For example:

exit 0 - exits your script and sets the exit status to 0 (success)
exit 1 - exits your script and sets the exit status to 1 (failure).

Example: Testing for root

Sometimes scripts might need to run with superuser privileges. If a regular user runs our script,it produces error messages.
Here we try to stop the script it if a regular user attempts to run it.

The ***id*** command can tell us who the current user is. When executed with the "-u" option, it prints the numeric user id of the current user.

```
if [ $(id -u) = "0" ]; then
   echo "superuser"
fi
```

In this example, if the output of the comman***d id -***u is equal to the string "0", then print the string "superuser."

Case

A case is multiple-choice branch. Unlike the simple branch, where you take one of two possible paths, a case supports several possible outcomes based on the evaluation of a condition.

The case command has the following form:

```
case word in
   patterns ) statements ;;
esac
```

Example: Test to identify the number entered

```
#!/bin/bash

echo -n "Enter a number between 1 and 3 inclusive > "
read character
case $character in
   1 ) echo "You entered one."
      ;;
   2 ) echo "You entered two."
      ;;
   3 ) echo "You entered three."
      ;;
   * ) echo "You did not enter a number"
      echo "between 1 and 3."
esac
```

case selectively executes statements if word matches a pattern. You can have any number of patterns and statements. Patterns can be literal text or wildcards. You can have multiple patterns separated by the "|" character.

Notice the special pattern "*". This pattern will match anything, so it is used to catch cases that did not match previous patterns. Inclusion of this pattern at the end is wise, as it can be used to detect invalid input.

Example: Test to identify the key pressed

```
#!/bin/bash
echo -n "Type a digit or a letter > "
read character
case $character in
        # Check for letters
   [a-z] | [A-Z] ) echo "You typed the letter $character" ;;

        # Check for digits
   [0-9] )     echo "You typed the digit $character" ;;

        # Check for anything else
   * )         echo "You did not type a letter or a digit"
esac
```

Loops

The final type of program flow control is called looping. Looping is repeatedly executing a section of your program based on a condition. The shell provides three commands for looping: while, until and for. We are going to cover while and until in this lesson and for in a future lesson.

1. While

The while command causes a block of code to be executed over and over, as long as a condition is true. Here is a simple example of a program that counts from zero to nine:

```
#!/bin/bash
number=0
while [ $number -lt 10 ]; do
   echo "Number = $number"
   number=$((number + 1))
done
```

On line 3, we create a variable called *number* and initialize its value to 0. Next, we start the while loop. As you can see, we have specified a condition that tests the value of number. In our example, we test to see if number has a value less than 10.

Notice the word do on line 4 and the word done on line 7. These enclose the block of code that will be repeated as long as the condition is met.

In most cases, the block of code that repeats must do something that will eventually change the outcome of the condition, otherwise you will have what is called an endless loop; that is, a loop that never ends.

In the example, the repeating block of code outputs the value of number (the echo command on line 5) and increments number by one on line 6. Each time the block of code is completed, the condition is tested again. After the tenth iteration of the loop, number has been incremented ten times and the condition is no longer true. At that point, the program flow resumes with the statement following the word done. Since done is the last line of our example, the program ends.

2. Until

The until command works exactly the same way, except the block of code is repeated as long as the condition is false. In the example below, notice how the condition has been changed from the while example to achieve the same result:

```
#!/bin/bash

number=0
until [ $number -ge 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

3. For

Like while and until, for is used to construct loops. for works like this:

```
for variable in words; do
    commands
done
```

In essence, for assigns a word from the list of words to the specified variable, executes the commands, and repeats this over and over until all the words have been used up. Here is an example:

```
#!/bin/bash

for i in word1 word2 word3; do
    echo $i
done
```

In this example, the variable i is assigned the string "word1", then the statement echo $i is executed, then the variable i is assigned the string "word2", and the statement echo $i is executed, and so on, until all the words in the list of words have been assigned.

EXERCISES: Lab Cycle - 1

1) Write a shell script to print multiplication table for a given number.

```
# program to print multiplication table

read -p "enter  the number to find multiplication table: " n
echo "number : $n "
read -p " enter limit :" lm
i=1
while [ "$i" -le "$lm" ]
 do
  prod=$((i*n));
  echo "$i * $n = $prod"
  i=$((i+1))
 done
```

2) Write a shell program to find sum of odd /even/all numbers from a list of n numbers.

```
# sum of and even numbers from the list of numbers

read -p "enter the limit :" n
echo "enter the numbers"
i=1
sumev=0
sumodd=0
while [ "$i" -le "$n" ]
 do
  read -p "enter the number $i : " num
  i=$((i+1))
  if [ "$((num%2))" -eq "0"  ] ; then
            echo "even"
            sumev=$((sumev+num))
  else
            echo "odd"
            sumodd=$((sumodd+num))
  fi
done
echo "sum of even = $sumev "
echo "sum of odd  = $sumodd"
```

3) Write a shell program to determine a given year is leap year or not.

```
# leap year

read -p "enter the year: " yr
if [ "$((yr%4))" -eq "0" ]; then
 if [ " $((yr%100))" -eq "0" ]; then
```

```
  if [ " $((yr%400))" -eq "0" ]; then
     echo "leap year"
  else
     echo "not a leap year"
  fi
 else
  echo "leap year"
 fi
else
  echo "not a leap year"
fi
```

4) Write a shell script that accepts two integers as its arguments and computes the value of first number raised to the power of the second number.

```
#finding the power of number

n=$1
p=$2
pow=$((n**p))
echo "power of $n to $p is $pow"
```

5) Write a shell script that computes the gross salary of a employee according to the following rules:

i) If basic salary is < 1500 then HRA =10% of the basic and DA =90% of the basic.
ii) If basic salary is >=1500 then HRA =Rs. 500 and DA=98% of the basic
The basic salary is entered interactively through the key board.

```
# gross salry

read -p "enter the basic salary: " sal
if [ "$sal" -lt "1500" ]
 then
  hra=$((sal+sal*10/100+sal*90/100))

elif [ "$sal" -ge "1500"  ]
 then
  hra=$((sal+500+sal*98/100))
fi
#gsal=$((sal+hra+da))
echo "basic salary: $sal"
echo "gross salary : $hra"
```

6) Write a shell script to perform the following string operations,
a) To extract a substring from a given string.
b) To find the length of a given string.

```
# substring
```

```
read -p "enter the string : " str
read -p "enter the length of substring: " len
read -p "enter the starting position of the substring: " start
echo "the string is : $str"
subs=${str:$start:$len}
echo "the substring is:  $subs"
echo "length of the string is ${#str}"
```

7) Write a shell script which receives two file names as arguments. It should check whether the two file contents are same or not. If they are same then second file should be deleted.

```
#compare two files

file1=$1
file2=$2
if cmp -s "$file1" "$file2"
 then
        echo "the contents of the file are same and the file $file2 is removed"
        rm $file2
else
        echo "the file contents are different"
fi
```

8) Write a shell script that accepts one or more filenames as arguments and convert their lower case letters to upper case (and vice-versa).

```
# case conversion

for  arg in "$@" #looping through arguments
do
        file=$arg
        tr '[:lower:][:upper:]' '[:upper:][:lower:]' < $file > tmp # reading from file $file
and coverting using tr(translator) and writing to temporary file called tmp
        mv tmp $file #moving from tmp file to orginal file
done
```

9) Develop an interactive shell script that ask for a word and a file name and then tells how many times that word occurred in the file.

```
#occurrence of word

read -p "enter the file name: " file
count=0
if [ -f $file ]
then
        read -p "enter the word to be checked  and counted: " wrd
        for i in $(cat $file)
        do
          if [ "$i" == "$wrd" ]
```

```
            then
                count=$((count+1))
            fi
        done
        echo "the word count is: $count"
else
        echo "file not exist"
fi
```

10) Write an interactive file-handling shell program. Let it offer the user the choice of copying, removing, renaming, or linking tiles and do the necessary operations according to user's choice

```
#file operations

f=0
while [ $f -ne 5 ]
do

        echo "1.copy"
        echo "2.remove"
        echo "3.rename file"
        echo "4.link file"
        echo "5.exit"
        echo "enter your choice"
        read n
        case $n in
                1)      echo "enter the file to be copied"
                        read file1
                        echo "enter the destination file"
                        read newfile
                        if [ -f $file1 -a $newfile ]
                        then
                                cp $file1 $newfile
                                echo "coping done"
                        fi
                        ;;
                2)      echo "enter the file to be deleted"
                        read file1
                        if [ -f $file1 ]
                        then
                                rm $file1
                                echo "file deleted"
                        fi
                        ;;
                3)      echo "ente file name to renamed"
                        read file1
                        echo "enter the new name"
                        read newfile
                        if [ -f $file1 ]
```

```
                then
                        mv $file1 $newfile
                fi
                echo "file renamed"
                ;;
        4)      echo "enter the filename to be linked "
                read file1
                echo "enter the hard link name "
                read newfile
                if [ -f $newfile ]
                then
                        echo "you cant link to an existing file"
                else
                        ln $file1 $newfile
                        echo "file linked"
                fi

        esac
        f=$n
done
```

11) Write a shell script that takes a login name as command -line argument. Display some message when that person logs in.

```
#welcome user with a message

name=$LOGNAME
if [ "$#" -eq 0 ] # checking arguments supplied or not
then
        echo "no arguments supplied"
else
        if [ "$1" == "$name" ]
        then
                echo "welcome"
        else
                echo "you are not the current user"
        fi
fi
```

12) Write a shell script that determines the period for which a specified user is working on the system.

```
#login time period


echo Enter the username
read user
us=$LOGNAME
```

```
if [ "$us" = "$user" ]
then
t=`who | tr -s " " | head -1 | cut -d " " -f4`
hr=`echo $t | cut -d ":" -f1`
min=`echo $t | cut -d ":" -f2`
h=`date "+%H"`
m=`date "+%M"`
if [ $m -lt $min ]
then
h=`expr $h - 1`
m=`expr $m + 60`
fi
h=`expr $h - $hr`
m=`expr $m - $min`
echo username is $user
echo working time is $h hrs $m minutes
else
echo unknown user
fi
```

Cycle -2
Version Control System setup and usage using GIT.

What is GIT

Git is a distributed revision control and source code management system with an emphasis on speed. Git was initially designed and developed by Linus Torvalds for Linux kernel development. Git is a free software distributed under the terms of the GNU General Public License version

Installing GIT

The official website of Git has detailed information about installing on Linux, Mac, or Windows. In this case, we would be using Ubuntu 13.04 for demonstration purposes, where you install Git using apt-get.

sudo apt-get install git

Initial Configuration

Let's create a directory inside which we will be working. Alternately, you could use Git to manage one of your existing projects, in which case you would not create the demo directory as below.

mkdir my_git_project cd my_git_project
The first step is to initialize Git in a directory. This is done using the command init, which

creates a .git directory that contains all the Git-related information for your project.

git init

Next, we need to configure our name and email. You can do it as follows, replacing the values with your own name and email.

git config --global user.name 'donny"
git config --global user.email donny@gmail.com"
git config --global color.ui 'auto'

It is important to note that if you do not set your name and email, certain default values will be used. In our case, the username 'donny' and email 'donny@gmail' would be the default values.

Also, we set the UI color to auto so that the output of Git commands in the terminal are color coded. The reason we prefix--global to the command is to avoid typing these config

commands the next time we start a Git project on our system.
Staging Files for Commit

The next step is to create some files in the directory. You could use a text editor like Vim. Note that if you are going to add Git to an already existing directory, you do not need to perform this step.

Check the Status of Your Repository

Now that we have some files in our repository, let us see how Git treats them. To check the current status of your repository, we use the git status command.

git status

Adding Files for Git to Track

At this point, we do not have any files for Git to track. We need to add files specifically to

Git order to tell Git to track them. We add files using add. git add my_file
Checking the status of the repository again shows us that one file has been added.

To add multiple files, we use the following (note that we have added another file for demonstration purposes.)

git add myfile2 myfile3

You could use git add recursively, but be careful with that command. There are certain files (like compiled files) that are usually kept out of the Git repository. If you use add recursively, it would add all such files, if they are present in your repository.

Removing Files

Let's say you have added files to Git that you do not want it to track. In such a situation, you tell Git to stop tracking them. Yet, running a simple git rm will not only remove it from Git, but will also remove it from your local file system as well! To tell Git to stop tracking a file, but still keep it on your local system, run the following command:

git rm --cached [file_name]

Committing Changes

Once you have staged your files, you can commit them into Git. Imagine a commit as a snapshot in time where you can return back to access your repository at that stage. You associate a commit message with every commit, which you can provide with the -m prefix.

git commit -m "My first commit"

Provide a useful commit message because it helps you in identifying what you changed in that commit. Avoid overly general messages like “Fixed bugs”. If you have an issue tracker, you could provide messages like “Fixed bug #234”. It’s good practice to prefix your branch name or feature name to your commit message. For instance, “Asset management – Added feature to generate PDFs of assets” is a meaningful message.

Git identifies commits by attaching a long hexadecimal number to every commit. Usually, you do not need to copy the whole string, and the first 5-6 characters are enough to identify your commit. In the screenshot, notice that 8dd76fc identifies our first commit.

Further Commits

Let’s now change a few files after our first commit. After changing them, we notice through git status that Git notices the change in the files that it is tracking.
You can check the changes to the tracked files from the last commit by running git diff. If you want to have a look at the changes to a particular file, you can run git diff <file>.

You need to add these files again to stage the changes in tracked files for the next commit. You can add all the tracked files by running:

git add -u

You could avoid this command by prefixing -a to git commit, which adds all changes to tracked files for a commit. This process, however, is very dangerous as it can be damaging. For instance, let’s say you opened a file and changed it by mistake. If you selectively stage them, you would notice changes in each file. But if you add -a to your commit, all files would be committed and you would fail to notice possible errors.

Once you have staged your files, you can proceed to a commit. Im mentioned that a message can be associated with every commit, which we entered by using -m. However, it is possible for you to provide multi-line messages by using the command git commit, which opens up an interactive format for you to write!

git commit

Managing of Your Project

To check the history of your project, you can run the following command.

git log

This shows you the entire history of the project — which is a list of all the commits and their information. The information about a commit contains the commit hash, author, time and commit message. There are many variations of git log, which you could explore once you understand the concept of a branch in Git. To view the details of a particular commit and the files that were changed, run the following command:

git show <hash>
Where <hash> is the hex number associated with the commit. As this tutorial is for beginners, we will not cover how to get back to the state of a particular commit in time or how to manage branches.

Putting Your Code in the Cloud

Once you have learned how to manage your code on your system, the next step is to put it in the cloud. Since Git doesn't have a central server like Subversion, you need to add each source to collaborate with others. That is where the concept of remotes comes in. A remote refers to a remote version of your repository.

If you wish to put your code in the cloud, you could create a project on GitHub, GitLab, or BitBucket and push your existing code to the repository. In this case, the remote repository in the cloud would act as a remote to your repository. Conveniently, a remote to which you have write access is called the origin.

After you create a remote repository, you have the ability to add a remote origin and then push the code to the origin.

git remote add origin https://github.com/donnygo/my_git_project.git
git push -u origin master

EXERCISES: Lab Cycle – 2

1) Preparing Directory and Files (Create a Directory myMaster, create 3 files inside the directory: myProfile. myFav, myHometown and fill the tiles with 5 sentences each, create two subdirectories projA and projB. create 2 files inside teach sub-directory: projDetail and projStatus and fill in the details and development status of your two projects).

```
$ mkdir myMaster
$ cd myMaster
$ touch myProfile myFav myHometown
$ mkdir projA projB
$ cd projA
$ touch projDetail projStatus
$ cd ..
$ cd projB
$ touch projDetail projStatus
```

2) Familiarisation of GIT environment (Installation and configuration of GIT in Linux) git. git config

```
$ sudo apt-get install git
$ git config --global user.name 'donny"
$ git config --global user.email donny@gmail.com"
$ git config --global color.ui 'auto'
```

3) Navigate inside the directory myMaster and initialise git repository. git init, git status

```
$ git init
➔   $ Initialized empty Git repository in /mca/myMaster/.git/
$ git status
➔   $ On branch master
➔   $  Initial commit
➔   $  Untracked files:
```

4) Add existing mes to the staging area and check status. git add <filename1>, <filename2> , git status

```
$ git add myProfile
$ git status
➔   $ On branch master
➔   $  Initial commit
➔   $  Changes to be committed:
➔   $        new file: myProfile
➔   $  Untracked files:

$ git add *
$ git status
```

5) Saving the files in staging area to the repository and check the history git commit , git log, git show

```
$ git commit –m "First Commit"
➔    [master (root-commit) 21e4d3f] first commit
➔    5 files changed, 0 insertions(+), 0 deletions(-)
```

➔ create mode 100644 myProfile
➔ create mode 100644 myFav

$ git log
➔ commit 21e163aee8756392367e5473fe77635625d0c2
➔ Author: Donny <donny@ubuntu.com>
➔ Date: Tue Jan 09 11:12:23 2018 +0530
➔ First Commit

$ git show
➔ commit 21e163aee8756392367e5473fe77635625d0c2
➔ Author: Donny <donny@ubuntu.com>
➔ Date: Tue Jan 09 11:12:23 2018 +0530
➔ First Commit
➔ diff –git myFav myProfile
➔ new file mode 100644
➔ index 0000000..e69de75

6) Getting old version of a file being tracked git checkout <filename>, git checkout <commit id> <filename>

$ git checkout 7e4fa3 myFav
➔ Note: checking out '7e4fa3'

7) Branching to other directories and working on different branches git branch <name>, git checkout <branchname>

$ git branch projA

8) Merging and deleting branches as per need and resetting a merge git merge <branchname>, git branch -d <branchname>, git reset

$ git merge projA

$ git reset

9) Revert to older versions git revert <commit id>

$ git revert <commit id>

10) Getting online. github, putting your repo into the cloud git remote push, pull

$ git remote add origin http://github.com/donnygo/myGitProject
$ git push –u origin master

11) Understanding more on git git stash, git rebase

$ git rebase

$ git stash

Cycle -3
Socket Programming

Client/Server Communication

At a basic level, network-based systems consist of a server , client , and a media for communication. A computer running a program that makes a request for services is called client machine. A computer running a program that offers requested services from one or more clients is called server machine.
They involve networking services provided by the transport layer.
The transport layer comprises two types of protocols,
TCP (Transport Control Protocol)
UDP(User Datagram Protocol).
The most widely used programming interfaces for these protocols are sockets.

- TCP is a connection-oriented protocol that provides a reliable flow of data between two computers.
  Example applications that use such services are HTTP, FTP, and Telnet.

- UDP is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival and sequencing.
  Example applications that use such services include DNS,Ping.

The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer. Port is represented by a positive (16-bit) integer value. Some ports have been reserved to support common and well-known services:

- FTP 21 TCP
- TELNET 23 TCP
- SMTP 25 TCP
- HTTP 80 TCP,UDP
- HTTPS 443 TCP,UDP

User-level processes and services generally use port number value >= 1024.

Socket

- The combination of an IP address and a port number.
- We say "create a socket connection between machine A and machine B". This means, roughly, create input and output streams for sending data between programs running simultaneously on each machine.

Two types

• Stream socket : reliable two-way connected communication streams
• Datagram socket

Socket functional calls

• socket (): Create a socket
• bind(): bind a socket to a local IP address and port #
• listen(): passively waiting for connections
• connect(): initiating connection to another socket
• accept(): accept a new connection
• Write(): write data to a socket
• Read(): read data from a socket
• sendto(): send a datagram to another UDP socket
• recvfrom(): read a datagram from a UDP socket
• close(): close a socket (tear down the connection)

Java classes for direct socket programming (3 additional classes are needed)

java-net-InetAddress
java-net-Socket
java-net-ServerSocket

EXERCISES: Lab Cycle – 3

1) Implement Bidirectional Client-Server communication using TCP.

```
import java.io.*;
import java.net.*;

public class ServerThread implements Runnable
{
ServerSocket  serverSocket = null;
Socket socket = null;
BufferedReader readFromClient = null; BufferedReader readFromServer  = null; PrintWriter
writeToClient = null;
String message = null;

public ServerThread()
{
try {
 serverSocket = new ServerSocket(4444); System.out.println("Waiting for connection...");
 socket = serverSocket.accept(); System.out.println("Connection accepted...");
 readFromClient = new BufferedReader(new InputStreamReader( socket.getInputStream()));
 writeToClient = new PrintWriter( socket.getOutputStream(), true);
 readFromServer  = new BufferedReader(new InputStreamReader( System.in));
 new Thread(this).start();
 while(true)
 {
  message = readFromServer.readLine(); writeToClient.println(message); writeToClient.flush();
  if(message.equalsIgnoreCase("exit"))
  { System.exit(O);}
 }
 }
 catch(IOException exp) { exp.printStackTrace();}
}

public void run()
{
 try {
 while(true) {
  String msg = readFromClient.readLine();
  if(!msg.equalsIgnoreCase("exit"))
  {
       System.out.println(msg);
  }
 else
 {
   System.exit(O);
 }
 }
 }
 catch(Exception exp) {
    exp.printStackTrace();
   }
 }
}
```

Server.java

```
public class Server
{     public static void main(String[] args)
  {      new ServerThread();     }
}
```

ClientThread.java

```
import java.io.*;
import java.net.*;

public class ClientThread implements Runnable
{
Socket socket = null;
BufferedReader readFromClient = null;
BufferedReader readFromServer  = null;
PrintWriter writeToServer = null;
String message = null;
public ClientThread(Socket socket)
{
 try
 {
  this.socket = socket;
  readFromServer  = new BufferedReader( new InputStreamReader( socket.getInputStream()));
  writeToServer = new PrintWriter( socket.getOutputStream(), true);
  readFromClient = new BufferedReader( new InputStreamReader(System.in));
  new Thread(this).start();
  while(true)
  {
     message = readFromClient.readLine();
     writeToServer.println(message);
     writeToServer.flush();
     if(message.equalsIgnoreCase("exit"))
     {
       System.exit(O);
     }
  }
 }
 catch(IOException exp) { exp.printStackTrace();}
}

public void run()
{
 try
 {
   while(true) {
   String msg = readFromServer.readLine();
   if(!msg.equalsIgnoreCase("exit"))
   {
```

```
        System.out.println(msg);
    }
    else { System.exit(O);}
  }
 }
 catch(Exception exp)
 {exp.printStackTrace();}
 }
}
```

Client.java

```
------------
import java.io.*;
import java.net.*;
public class Client
{
  Socket socket = null;
  public Client()
  {
    try
    {
          socket = new Socket("localhost",4444);
          new ClientThread(socket);
    }
    catch(UnknownHostException exp)
    {
          exp.printStackTrace();
    }
    catch(IOException exp)
    {
          exp.printStackTrace();
        }
  }
 public static void main(String[] args)
 {
   new Client();
 }
}
```

SAMPLE OUTPUT

Server	Client
Waiting for connection... Connection accepted... Hello Server. Hi Client. I request for the file ABCDE You will be given access to the file ABCDE now. exit	 Hello Server. Hi Client. I request for the file ABCDE You will be given access to the file ABCDE now.

2) Implement Bidirectional Client-Server communication using UDP.

ServerUDP.java

```
import java.io.*;
import java.net.*;

class ServerUDP
{
  public static void main(String args[]) throws Exception
    {
      DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];
        while(true)
          {
            DatagramPacket receivePacket = new DatagramPacket(receiveData,
                                                              receiveData.length);
            serverSocket.receive(receivePacket);
            String sentence = new String( receivePacket.getData());
            System.out.println("RECEIVED: " + sentence);
            InetAddress IPAddress = receivePacket.getAddress();
            int port = receivePacket.getPort();
            String capitalizedSentence = sentence.toUpperCase();
            sendData = capitalizedSentence.getBytes();
            DatagramPacket sendPacket =
            new DatagramPacket(sendData, sendData.length, IPAddress, port);
            serverSocket.send(sendPacket);
          }
    }
}
```

ClientUDP.java

```
import java.io.*;
import java.net.*;

class ClientUDP
{
  public static void main(String args[]) throws Exception
  {
    BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
    DatagramSocket clientSocket = new DatagramSocket();
    InetAddress IPAddress = InetAddress.getByName("localhost");
    byte[] sendData = new byte[1024];
    byte[] receiveData = new byte[1024];
    String sentence = inFromUser.readLine();
    sendData = sentence.getBytes();
    DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
                                                   IPAddress, 9876);
    clientSocket.send(sendPacket);
    DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
    clientSocket.receive(receivePacket);
    String modifiedSentence = new String(receivePacket.getData());
```

```
    System.out.println("FROM SERVER:" + modifiedSentence);
    clientSocket.close();
  }
}
```

3) Implement Echo Server using TCP

EchoServer.java

```
------------------------
import java.io.*;
import java.net.*;

public class echoserver{
 public static void main(String args[])throws Exception
 { ServerSocket m=null; Socket c=null;
   DataInputStream usr_inp=null;
   DataInputStream din = new DataInputStream(System.in);
   DataOutputStream dout=null;
   try {
          m= new ServerSocket(1234); c=m.accept();
          usr_inp= new DataInputStream(c.getInputStream());
          dout= new DataOutputStream(c.getOutputStream());
     }
   catch(UnknownHostException e) { }
   catch(IOException g) { }
   if(c!=null || usr_inp!=null)
   {
          String unip;
          while(true)
          {
            System.out.println("Message From Client..");
            String m1=usr_inp.readLine();System.out.println(m1);
            dout.writeBytes(""+m1);dout.writeBytes("\n");
          }
   }
   dout.close(); usr_inp.close();c.close();
 }
}
```

SAMPLE OUTPUT

EchoClient	EchoServer
hello The Echoed Message hello Enter Your Message: server? The Echoed Message server? Enter Your Message: Hello The Echoed Message Hello Enter Your Message:	Message From Client.. hello Message From Client.. server? Message From Client.. Hello Message From Client.. STOP Message From Client..

4) Implement Chat Server using UDP

UDPChatServer.java

```
import java.io.*;
import java.net.*;

class UDPChatServer
{
        public static DatagramSocket serversocket;
        public static DatagramPacket dp;
        public static BufferedReader dis;
        public static InetAddress ia;
        public static byte buf[] = new byte[1024];
        public static int cport = 789,sport=790;
        public static void main(String[] a) throws IOException
        {
                serversocket = new DatagramSocket(sport);
                dp = new DatagramPacket(buf,buf.length);
                dis = new BufferedReader (new InputStreamReader(System.in));
                ia = InetAddress.getLocalHost();
                System.out.println("Server is Running...");
                while(true)
          {
                serversocket.receive(dp);
                String str = new String(dp.getData(), 0,dp.getLength());
                if(str.equals("STOP"))
                 {
                   System.out.println("Terminated...");
                   break;
                 }
           System.out.println("Client: " + str);
           System.out.print("Server: ");
           String str1 = new String(dis.readLine());
           buf = str1.getBytes();
           serversocket.send(new DatagramPacket(buf,str1.length(), ia, cport));
          }
        }
}
```

UDPChatClient.java

```
import java.io.*;
import java.net.*;

class UDPChatClient
{
        public static DatagramSocket clientsocket;
        public static DatagramPacket dp;
        public static BufferedReader dis;
        public static InetAddress ia;
        public static byte buf[] = new byte[1024];
        public static int cport = 789, sport = 790;
```

```
        public static void main(String[] a) throws IOException
        {
                clientsocket = new DatagramSocket(cport);
                dp = new DatagramPacket(buf, buf.length);
                dis = new BufferedReader(new InputStreamReader(System.in));
                ia = InetAddress.getLocalHost();
                System.out.println("Client is Running... Type 'STOP'to Quit");
          while(true)
          {
                System.out.print("Client: ");
                String str = new String(dis.readLine());
                if(str.equals("STOP"))
                {
                  System.out.println("Terminated...");
                  clientsocket.send(new DatagramPacket(buf,str.length(), ia,sport));
                 break;
           }

          buf = str.getBytes();
          clientsocket.send(new DatagramPacket(buf,str.length(), ia, sport));
          clientsocket.receive(dp);
          String str2 = new String(dp.getData(), 0, dp.getLength());
          System.out.println("Server: " + str2);
         }
        }
}
```

SAMPLE OUTPUT

Client	Server
Client is Running... Type 'STOP' to Quit **Client: Hello Server, Client here** Server: Hi Client, What can I do for you? **Client: How many files do you have?** Server: OH. I have so many files **Client: STOP** Terminated...	Server is Running... Client: Hello Server, Client here **Server: Hi Client, What can I do for you?** Client: How many files do you have? **Server: OH. I have so many files** Terminated…