ARIZONA STATE UNIVERSITY

CSE 434, SLN 60419 and 87949 — Computer Networks — Fall 2025

Instructors: Dr. Violet R. Syrotiuk and Dr. Bharatesh Chakravarthi

Socket Programming Project

Available Sunday, 09/14/2025; Milestone due Sunday, 09/28/2025; Full project due Sunday, 10/19/2025

The purpose of this project is to implement your own application program in which processes communicate using sockets to build a *distributed storage system* (DSS) to store and retrieve files.

- You may write your code in `C/C++` or in `Python`; no other programming language is permitted. Each of these languages has a socket programming library that you **must** use for communication.
- The application programs that form your solution to this project **must** be multi-threaded.
- This project **must** either be completed individually or by a group of size at most two. Whatever your choice, you **must** join a `Socket Group` under the `People` tab on Canvas before Sunday, 09/21/2025. This group can be the same as or different from the group used in Lab 1.
- Each group **must** restrict its use of port numbers to prevent the possibility of application programs from interfering with each other. As described in §3.3, port numbers are dependent on you group number.
- You **must** use GitHub, a version control system, as you develop your solution to this project. Your code repository **must** be *private* to prevent anyone from plagiarizing your work. It is expected that you will commit changes to your repository on a regular basis. You will be required to provide a full commit history to demonstrate both your effort invested in this project and the provenance of your code.

Following a brief overview of DSSs and the application architecture for this project in §1, the requirements of the `manager`, `user`, and `disk` programs are provided in §2. Some implementation details, such as the use of multi-threading, are described in §3. The requirements for the milestone and full project submissions are described in §4.

1 A Distributed Storage System using Block-Interleaved Distributed Parity

Disk arrays for storage systems were first proposed in the 1980s. Such network-based systems have evolved into cloud-based storage services such as Amazon's Simple Storage Service (S3) [1].

Today's cloud-based storage services are still based on ideas from RAID, *i.e.*, disk arrays in which multiple, independent physical disks are organized into a large, high-performance *logical* disk [2]. Disk arrays stripe data across multiple drives in order to access them in parallel. Striping not only enables higher data transfer rates, it also improves load balancing across the drives. The granularity of striping is either bit-level or block-level.

Large disk arrays, however, are vulnerable to disk failures [2]. As a result, some form of redundancy is used to tolerate disk failures [5]. Parity is one such method. Rather than a storing the results of a parity computation on the same drive, the parity can be distributed across the disk array.

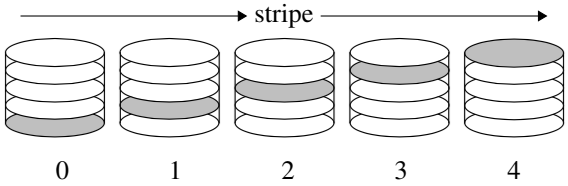

Figure 1: Block-interleaved distributed parity in a disk array of $n = 5$ drives.

Figure 1 illustrates *block-interleaved distributed parity* (BIDP) in an array of $n = 5$ drives. Each drive has multiple "platters" indicating blocks. The parity block (shaded) is computed from the contents of other blocks in the same stripe. In each stripe, the parity block is on a different drive and, here, is distributed uniformly over all of the drives.

Figure 2 depicts the high-level architecture of the DSS application to be implemented in this socket project. For simplification, the DSS will be implemented by a data structure in main memory of n processes rather than on n physical disk drives. In the figure, three processes collectively implement a disk array of $n = 3$ drives using BIDP.

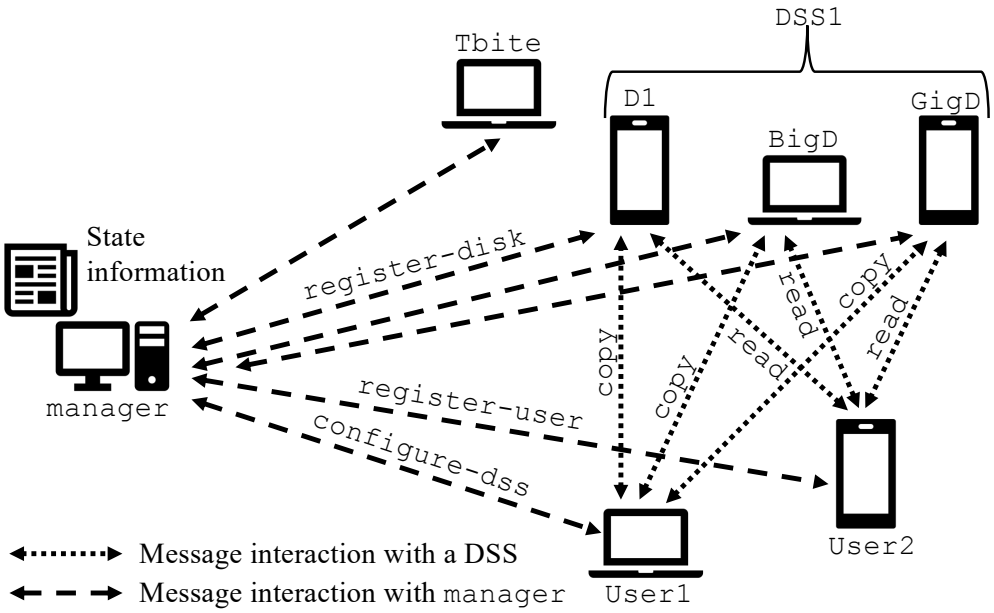

Figure 2: High level architecture of the DSS application.

The `manager` process maintains state information about the DSS, such as parameters of the disk array and users of the DSS. In this example, suppose three disk processes (`D1`, `BigD`, and `GigD`) are configured to operate as a disk array of size $n = 3$ named `DSS1` with striping-unit of 1 KB. One user process (`User1`) is shown copying a file to the DSS while another (`User2`) is reading a file from it. These file operations can performed in parallel by separate threads to improve performance, checking parity on a per stripe basis.

2 The Distributed Storage System Application

This socket programming project involves the design and implementation of three programs:

1. The first program implements an "always-on" single-threaded `manager` process to support management of the processes implementing $k \geq 1$ DSS, and its users. Your `manager` must read one command line parameter, an integer giving the port number (from your port number set) where the `manager` listens for messages.
2. The second program implements a multi-threaded `disk` process. It may participate as one disk in a disk array, *i.e.*, a DSS. A `disk` should read five command line parameters:
   (a) the name of the disk,
   (b) the IPv4 address of the `manager` in dotted decimal,
   (c) the port number of the `manager`,
   (d) a management port number (from your port number set), and
   (e) a command port number (from your port number set).
3. The third program implements a `user` process. After a DSS is configured, a user can use it to store and retrieve files, among other operations, some of which must be multi-threaded. A `user` reads five command line parameters, the same as the `disk` program except that the name is a user name rather than a disk name.

The messages to be supported by the manager are described in §2.1. Messages to be supported by peers, *i.e.*, disks and users, are described in §2.2.

2.1 The DSS Manager Protocol

Recall that a *protocol* defines the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event [3].

First, the manager process is started on a host. It then runs in an infinite loop listening on the given message port. As it processes messages, the manager updates state of the DSS application.

Peer processes (disks and users) can then be started. They read commands from stdin – no fancy UI is expected! A peer reads in a command and constructs a message to be sent to the manager or to another peer over a UDP socket in the expected format, and waits for a response. Management ports (m-port) are used for communication between the manager and peers. Command ports (c-port) are used for communication between peers.

As you process commands, you should produce a well-labelled trace the messages transmitted and received between processes so that it is easy to follow what is happening in your DSS application program.

In the following, <> bracket parameters to a command, while the other strings are literals. The manager must support messages corresponding to the following commands issued at a peer:

1. register-user ⟨user-name⟩ ⟨IPv4-addr⟩ ⟨m-port⟩ ⟨c-port⟩, sent by a user process to register itself with the manager. The user-name is an alphabetic string of length at most 15 characters, IPv4-addr is the IPv4 address in dotted decimal of the host running this user process, and m-port and c-port are port numbers the manager and peers use, respectively, to communicate with this user.

   On receipt of a register-user, the manager checks that the parameters of the command satisfy certain conditions. Specifically, each user-name may only be registered once. The IPv4-addr of a user need not be unique, *i.e.*, one or more peer processes may run on the same end host. However, the ports used for communication by each process must be unique. If the conditions of the parameters are satisfied, the manager adds the parameters to the app's state, and responds to the user with a return code of SUCCESS.

   In the case of a duplicate registration, or some other problem, the manager responds to the user's m-port with a return code of FAILURE.

2. register-disk ⟨disk-name⟩ ⟨IPv4-addr⟩ ⟨m-port⟩ ⟨c-port⟩, sent by a disk process to register itself with the manager. The parameters are the same as for the register-user command, except that the name is the name of the disk rather than a user's name.

   On receipt of a register-disk, the manager takes essentially the same actions as for the register-user command. One difference is that if the registration is successful, the manager also tracks the state of disk-name. Initially, disk-name has state Free, *i.e.*, it is a available to act as one disk of a disk array.

3. configure-dss ⟨dss-name⟩ ⟨n⟩ ⟨striping-unit⟩, sent by a user process to configure a DSS. The dss-name is an alphabetic string of length at most 15 characters and $n \geq 3$ is the number of disks in the disk array. The striping-unit is the block size in bytes; it must be a power of two.

   While each disk may participate in only one DSS at a time, the application should be able to support more than one DSS in the system.

   The configure-dss command results in a return code of FAILURE if:

   - $n \not\geq 3$,
   - there are fewer than n disks with a state of Free,
   - the striping-unit is not a power of two between 128 bytes and 1 MB, or
   - a DSS with name dss-name has already been configured.

Otherwise, the `manager` selects at random `n` disks with a state of `Free` from those registered updating each one's state to `InDSS` for the given `dss-name` and `striping-unit`. The **order** of the disks in the disk array is part of the configuration, to ensure the correct storage and retrieval of files. Then the `manager` confirms configuration of a DSS by responding with a return code of `SUCCESS`.

4. `ls`, sent by a user process to list the files stored on all DSSs. This command results in a return code of `FAILURE` if no DSSs are configured. Otherwise, the `manager` sets the return code to `SUCCESS` and constructs a response to the query that includes, for each DSS:

   - the parameters of the DSS itself, including its `dss-name`, `n` along with the names of the `n` disks making up the disk array given in order, and the `striping-unit`.
   - the parameters of each file stored on the DSS including the `file-name`, `file-size`, and `owner`.

   On receipt of a successful query, the user process summarizes the output.

   Suppose, as Figure 2 depicts, `DSS1` is configured with $n = 3$, and a striping-unit of size 1 KB. Further suppose that `DSS1` contains three files created by two different users. Then an example summary includes all details:

```
DSS1:  Disk array with n=3 (D1, BigD, GigD) with striping-unit 1 KB.
   day-of-affirmation.txt    6286 B    User2
   in-flanders.txt           587 B     User1
   tale-of-two-cities.txt    760,569 B User1
```

5. `copy` ⟨`file-name`⟩ ⟨`file-size`⟩ ⟨`owner`⟩, sent by a user process to initiate the storage of a copy of a file on a DSS (in a flat file system). If no DSSs are configured, then the `manager` sends a return code of `FAILURE`.

   Otherwise, this command operates in two phases, to effectively copy the file to a DSS in a critical section. The `manager` selects a DSS among those configured for the user to store its file. It informs the user of the parameters of the selected DSS as the response in the first phase. This includes the `dss-name`, the size n of the disk array, its `striping-unit`, and `n` triples providing the disk name, IPv4 address, and command port number of each disk process making up the disk array in configuration order:

$$\begin{array}{ccc} \texttt{disk}_0 & \texttt{IPv4-addr}_0 & \texttt{c-port}_0 \\ \texttt{disk}_1 & \texttt{IPv4-addr}_1 & \texttt{c-port}_1 \\ \vdots & \vdots & \vdots \\ \texttt{disk}_{n-1} & \texttt{IPv4-addr}_{n-1} & \texttt{c-port}_{n-1} \end{array}$$

   On receipt of the parameters, the user proceeds to copy the file to the named DSS; see 2.2.1 for details.

   The `manager` waits for a `copy-complete` message from the user, indicating the completion of the file copy to the DSS; receipt of any other commands by the manager should result in `FAILURE` as a response. Only then does the `manager` update the directory information of the DSS, updating its state to include the file, its size, and owner (*i.e.*, the `user-name` that executed the `copy`) on the selected DSS. Finally, the `manager` sends a return code of `SUCCESS` indicating completion of second phase of the `copy` command.

6. `read` ⟨`dss-name`⟩ ⟨`file-name`⟩ ⟨`user-name`⟩, sent by a user process to read a file from a DSS. If no DSSs are configured, the file does not exist on the named DSS, or the user is not the `owner` of the file, then the `manager` sends a return code of `FAILURE`.

   Otherwise, the file exists on the named DSS. The read operation does not need to take place within a critical section as it does not involve the modification of any blocks of the DSS. The `manager` sets the return code to `SUCCESS` and responds with the `file-size` and the parameters of the `dss-name` to the user. These parameters include the `dss-name`, the size n of the disk array, its `striping-unit`, and `n` triples providing the disk name, IPv4 address, and command port number of each disk process making up the disk array in configuration order as in the `copy` command.

On receipt of the parameters, the user proceeds to read the file from the named DSS; see 2.2.2 for details.

The `manager` must maintain state information of in-progress `read` operations to simplify any `disk-failure` commands. On receipt of `read-complete` from the named user, the `manager` deletes the read state and responds with a return code of `SUCCESS`.

7. `disk-failure` ⟨`dss-name`⟩, sent by a user process to trigger a disk failure in a DSS. If the named DSS is not configured or it has `read` operations in progress, then the `manager` sends a return code of `FAILURE`.

   Otherwise, the DSS `dss-name` exists and is not being read. Similar to the `copy` command, a `disk-failure` operates in two phases to prevent operations on the DSS until the failure is resolved. The `manager` sends the user the parameters of the named DSS as the response in the first phase as in the `copy` and `read` commands.

   On receipt of the parameters, the user proceeds to simulate a failure of a disk in the DSS; see 2.2.3 for details.

   The `manager` then waits for a `recovery-complete` message from the user, indicating that the DSS has recovered from the failure; receipt of any other commands by the `manager` should result in `FAILURE` as a response. Then the `manager` sends a return code of `SUCCESS` indicating completion of the second phase of the `disk-failure` command.

8. `deregister-user` ⟨`user-name`⟩, issued by a user process to deregister itself with the `manager`. If the user does not exist then the `manager` sends a return code of `FAILURE`. Otherwise it removes state associated with the `user-name` and sends a return code of `SUCCESS`. Once deregistered, the user process terminates.

9. `deregister-disk` ⟨`disk-name`⟩, issued by a disk process to deregister itself with the `manager`. If the disk does not exist or `disk-name` has state `InDSS` then the `manager` sends a return code of `FAILURE`. Otherwise, it removes state associated with the `disk-name` and sends a return code of `SUCCESS`. Once deregistered, the disk process terminates.

10. `decommission-dss` ⟨`dss-name`⟩, issued by a user process to initiate the decommissioning of a DSS. If the DSS does not exist then the `manager` sends a return code of `FAILURE`.

    Otherwise, this command needs to be effectively executed in a critical section, *i.e.*, the `manager` should respond with `FAILURE` for any incoming commands until this command is complete. Repurpose your implementation of the `disk-failure` and `recovery-complete` commands to instruct each disk in the DSS `dss-name` to delete its contents. Then, the state of each disk in the DSS should change from `InDSS` to `Free`, *i.e.*, each disk could be reconfigured in another DSS. Finally, the `manager` sends a return code of `SUCCESS` indicating completion of the `decommission-dss` command.

    You are to define the details of the `decommission-dss` command. Document your design decisions.

2.2 The DSS Peer-to-Peer (P2P) Protocol

2.2.1 Copying a File to a DSS, `copy`

In response to the first phase of a `copy`, a user process obtains the parameters of the DSS selected by the `manager` that will be used to store the file on the disks making up the disk array.

The user process now proceeds to copy the file `file-name` on the DSS. This involves repeatedly reading blocks of the file to form a stripe, and writing the stripe **in parallel** to the disk array of size n. The redundancy-method is *block-interleaved distributed parity* (BIDP) hence each stripe contains $n-1$ data blocks and one parity block. The parity block is computed as the XOR of the $n-1$ data blocks in the stripe. All blocks are of size `striping-unit`. If there is insufficient data in the file to fill all blocks of a stripe, the blocks should be `NULL` (ASCII `0x00`) padded.

In successive stripes, the parity block is written to a different disk, *i.e.*, the parity is *distributed* across the disk array. As in Figure 1, in stripe zero, the parity block is stored on disk $n-1$. In stripe 1, the parity block is stored on disk

$n-2$. In general, in stripe $i \geq 0$ the parity block is stored on disk $n - ((i \bmod n) + 1)$.

Suppose the DSS `dss-name` has n drives defined by the n triples. Further suppose that a file `file-name` to be stored on `dss-name` has f bytes, and that the `striping-unit` is b bytes. Then each stripe has $n-1$ blocks of size b, or $(n-1) \times b$ bytes, of data. Therefore, a total of $\lceil \frac{f}{(n-1)\times b} \rceil$ stripes are needed to store the file. For example, if $n = 3$ and $b = 128$ B, then each stripe has two data blocks and one parity block of size b. If the file `in-flanders.txt` has 587 B, then a total of $\lceil \frac{587}{2\times 128} \rceil = 3$ stripes are needed to store the file (with the last stripe `NULL` padded).

For each set of $n-1$ data blocks read from the file, a parity block is computed. The n blocks are then mapped to the stripe, interleaving the parity block with the data blocks as determined by the stripe number. Once the stripe is formed, n threads are spawned, with thread $0 \leq i < n$ writing block i of the stripe to `c-port`$_i$ of the `disk-name`$_i$ running on the host with `IPv4-addr`$_i$. The threads terminate after the stripe has been written.

Iterate until the entire file has been copied to the disk array, where the number of iterations equals $\lceil \frac{f}{(n-1)\times b} \rceil$, the number of stripes. The user process then sends a `copy-complete` message to the `manager`, and waits for a response of `SUCCESS`, indicating the second phase of the `copy` command is complete.

You are to define the details of the `copy` command protocol between a user process and disk processes making up a DSS. For example, the disks in the DSS need to store both the parameters of the DSS itself, and parameters of the files stored on it for subsequent `read` operations, and most likely the stripe number and block type (*i.e.*, a data or parity block). Document your design decisions.

2.2.2 Reading a File from a DSS, `read`

A response to a successful `read`, includes both the size of the file, and all the parameters of the DSS on which the `file-name` is stored. As in §2.2.1, let f be the file size in bytes and b the block size, *i.e.*, `striping-unit`, in bytes. Then, a total of $\lceil \frac{f}{(n-1)\times b} \rceil$ stripes must be read from the DSS and verified correct.

For each stripe $0 \leq i < \lceil \frac{f}{(n-1)\times b} \rceil$:

1. Spawn n threads, to read the blocks of stripe i of the file from the DSS **in parallel**.
2. Introduce a **single bit error** into any block of the stripe with probability $p \geq 0$. That is, for a given p, generate a random integer k between 0 and 100. If $k < p$ introduce the bit error (flip a single bit at random in the block). Experiment with the range $0 \leq p \leq 100\%$.
3. Verify the parity, *i.e.*, recompute the parity block and compare it to the parity block returned from the read.
   (a) If the parity is verified, *i.e.*, the parity blocks are equal, write the data blocks of the stripe to an output file, concatenating successive stripes.
   (b) Otherwise, reread the stripe from the DSS until the parity is verified. (While there is no need to introduce an error in rereading the stripe, it is possible a spurious error occurs in the read operation regardless.)
4. Terminate the threads after a stripe is read successfully.

After successfully reading all stripes, the output file produced by the `read` command should equal the file copied to the DSS with the `copy` command. Verify this fact by calling the `diff` function on the two files (see the `system` function from C++, or the `subprocess` module in python) from the user process.

Finally, send a `read-complete` message to the `manager` so that it can update the read state of the DSS, waiting for a response of `SUCCESS`.

You are to define the details of the `read` command protocol between a user process and disk processes making up a DSS. This includes how you obtain the value of p. It can be done on a per-stripe basis, or a per-file basis added to the `read` command, as you like. Document your design decisions.

2.2.3 Simulating a Disk Failure in the DSS, `disk-failure`

In response to the first phase of a `disk-failure`, a user process obtains the parameters of the named DSS.

At random, select one of the drives $0 \leq i < n$ making up DSS `dss-name` to fail. Send a `fail` message to `c-port`$_i$ of the `disk-name`$_i$ running on the host with `IPv4-addr`$_i$. On receipt of such a `fail` message, the disk process implements a disk failure by deleting all file contents of `disk-name`$_i$, *i.e.*, the data structure storing the blocks (data and parity) associated with the ith disk in the disk array `dss-name`. Once the disk process has completed the disk failure, it returns a `fail-complete` message to the user.

On receipt of the `fail-complete` message, the user must now recover the failed disk. This is accomplished as follows:

For each drive in the DSS except failed disk, *i.e.*, $D = \{0, 1, \ldots, n\} \setminus \{i\}$, $|D| = n - 1$:

1. For each file f stored on the DSS:
   (a) For each stripe j of the file:
      i. Read a block of stripe j from each disk in D **in parallel;** it may be a data block or a parity block.
      ii. Compute the XOR of the $n - 1$ blocks just read, and store the resulting block as stripe j on drive i for f. You need to determine if the computed block is a data block or a parity block.

Once the failed drive is reconstructed, the user process then sends a `recovery-complete` to the `manager`, waiting for a response of `SUCCESS`.

You are to define the details of the `disk-failure` command protocol between a user process, the disk selected to fail, and the process of recovering its contents. Document your design decisions.

3 Implementation Suggestions

3.1 Threading

Different threads should handle the socket associated with the `m-port` and `c-port` of each process. Threading is also required to implement the `copy`, `read`, and `disk-failure` commands. You may make use of additional threading beyond this; if you do, be sure to document it as a design decision.

Be aware that by default the function `recvfrom()` in C++ is blocking. You can change it to be non-blocking by setting the `flags` argument of `recvfrom()` or by using the function `fcntl()`.

3.2 Defining Message Exchanges and Message Format

Sections §2.1 and §2.2 have described the order of some message exchanges, as well as the actions taken on the transmission and/or receipt of a message. As part of this project, you need to complete the design details for many commands, including the design of data structures to store stripes of a file on a DSS.

This also includes defining the message format. This could be achieved by defining a structure with the fields required by the command. Or a message can be a string with concatenated fields separated by a delimiter. Either choice is fine so long as you are able to extract the fields of a message and interpret them.

You may want to define additional return codes to further differentiate `SUCCESS` and `FAILURE` states.

3.3 Port Numbers

Port numbers are divided into three ranges [4]:

- *Well-known ports:* 0 through 1023. These port numbers are controlled and assigned by the Internet Assigned Numbers Authority (IANA).
- *Registered ports:* 1024 through 49151. The upper limit of 49151 for these ports is new; RFC 1700 lists the upper range as 65535. These port numbers are not controlled by the IANA.
- *Dynamic* or *private ports:* 49152 through 65535. The IANA dictates nothing about these ports. These are the ephemeral ports.

Each group $1 \leq G \leq 200$ in CSE 434 is assigned a set of 500 unique port numbers to use in the following range. If $G \bmod 2 = 0$, *i.e.*, your group number is even, then use the range: $\left[\left(\frac{G}{2} \times 1000\right) + 1000, \left(\frac{G}{2} \times 1000\right) + 1499\right]$. If $G \bmod 2 = 1$, *i.e.*, your group number is odd, then use the range: $\left[\left(\left\lceil \frac{G}{2} \right\rceil \times 1000\right) + 500, \left(\left\lceil \frac{G}{2} \right\rceil \times 1000\right) + 999\right]$. That is, group 1 has range $[1500, 1999]$, group 2 has range $[2000, 2499]$, group 3 has range $[2500, 2999]$, and so on.

Do not use port numbers outside your assigned range, as otherwise you may send messages to another group's processes by accident and it is unlikely it will be able to interpret it correctly, causing spurious crashes.

4 Submission Requirements for the Milestone and Full Project Deadlines

Submit electronically before 11:59pm on the deadline date a zip file named `GroupX.zip` where `X` is your group number. You must use the `zip` archiving program.

1. The milestone is due on Sunday, 09/28/2025. See §4.1 for requirements.
2. The full project is due on Sunday, 10/18/2025. See §4.2 for requirements.

It is your responsibility to submit your project well before the time deadline!!! Late projects are not accepted. Do not expect the clock on your machine to be synchronized with the one on Canvas!

An unlimited number of submissions are allowed. Submit early, submit often! The last submission will be graded.

4.1 Submission Requirements for the Milestone

For the milestone deadline, you are to implement the following commands to the `manager`: `register-user`, `register-disk`, `configure-dss`, `deregister-user`, and `deregister-disk`.

For the milestone, your `zip` file must contain:

1. **Design document in PDF format (50%).** Describe the design of your DSS application program.
   (a) Include a description of your message format for each command implemented for the milestone.
   (b) Include a time-space diagram for each command implemented to illustrate the order of messages exchanged between communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event, *i.e.*, the protocol followed to implement the command. You **must** use a software tool, *e.g.*, PowerPoint, to draw these diagrams; photos of hand drawn diagrams are not accepted.
   (c) Describe your choice of data structures used, and design decisions made. For example, what state information are you storing at the `manager`?
   (d) Include the following screenshots taken in your GitHub repository for this socket project:
      i. Take screenshot(s) of the output of the command run in the terminal of the branch you are working on: `git log --pretty=format:"%h - %an, %ad (Commit) - %cd (Author)"`
      ii. Take a screenshot of the entire commit history of your GitHub repository.
      iii. Take a screenshot of the output of the command: `git reflog`

iv. Take a screenshot of the output of the command: `git fsck`

(e) Provide a *a link to your video demo* and ensure that the link is accessible to our graders. In addition, give a list of timestamps in your video at which each step 3(a)-3(f) is demonstrated.

2. **Code and documentation (25%).** Submit your well-documented source code implementing the milestone of your DSS application program.

3. **Video demo (25%).** Upload a video of length at most 7 minutes to YouTube with no splicing or edits, with audio accompaniment. This video must be uploaded and timestamped *before* the milestone submission deadline.

   The video demo of your DSS application for the milestone must include:

   (a) Compile your `manager`, `user` and `disk` programs (if applicable).
   (b) Your milestone demo needs to use at least two distinct end-hosts.
   (c) Start your `manager` program. Then start three `disk` processes and two `user` processes that each register with the `manager`. Be sure to assign port numbers from your port number space (see §3.3).
   (d) Have one `user` issue a `configure-dss` command to build a DSS of size `n` $= 3$ with other parameters of your choice.
   (e) Have the other `user` issue a `configure-dss` command; this should fail due to insufficient disks.
   (f) Now deregister each `user` and `disk`, and terminate your `manager`.

   For the end-hosts in your demo, you can use the PCs on the racks in BYENG 217 on a single-segment Ethernet as in Lab 1, VMs on a LAN you configure in CloudLab, or any other end-hosts available to you.

   Your video will require at least six (6) terminal windows: one for the `manager`, and one for each `disk` and `user`. Ensure that the font size in each window is large enough to read easily!

   Be sure that you produce a well-labelled trace of the messages transmitted to and received from processes so that it is easy to follow what is happening in your DSS application program.

4.2 Submission Requirements for the Full Project

For the full project deadline, you are to implement the all commands to the `manager` listed in §2.1. This also involves implementation of all commands issued between peers that are associated with these commands as described in §2.2.

For the full project submission, your `zip` file must contain:

1. **Design document in PDF format (30%).** Extend the design document for the milestone of your DSS application to include details for the remaining commands implemented for the full project, as well as any new or updated design decisions, including your use of threads. Provide all items listed in step 1 of §4.1, except that the list of timestamps for the video must be for the steps 3(a)-3(h) listed below.

2. **Code and documentation (20%).** Submit well-documented source code implementing your DSS application program. In particular, comment your use of threading!

3. **Video demo (50%).** Upload a video of target length at most 15 minutes to YouTube with no splicing or edits, with audio accompaniment. This video must be uploaded and timestamped *before* the full project deadline.

   The video demo of your DSS application for the full project must include:

   (a) Compile your `manager` and `peer` programs (if applicable).
   (b) Your full project demo must use at least four distinct end-hosts.
   (c) Start your `manager` program. Then start six `disk` processes and two `user` processes that each register with the `manager`.
   (d) Select one user and have it issue the following commands:
      i. `configure-dss` a DSS of size `n` $= 3$ with other parameters of your choice.

ii. `copy` three text files onto the DSS: `day-of-affirmation.txt`, `in-flanders.txt`, and `tale-of-two-cities.txt`.
iii. List the files, `ls`.
iv. `read` the file `day-of-affirmation.txt` from the DSS.

(e) Have the other user issue the following commands:
i. `configure-dss` a second DSS of size `n` $= 3$ with other parameters of your choice.
ii. `copy` the text file `wizard-of-oz.txt` onto the second DSS.
iii. List the files, `ls`.
iv. Try to `read` `in-flanders.txt` from the first DSS; this should fail due to ownership.

(f) Select one user to cause a `disk-failure` on a DSS.

(g) Then have an owner `read` a file from the DSS that suffered the disk failure.

(h) Finally, `decommission-dss` each DSS, and then deregister each `user` and `disk`, to gracefully terminate of your application. Of course, the `manager` process needs to be terminated explicitly.

As for the milestone, use one terminal window for the `manager` and each `disk` and `user` process. Ensure that the font size in each window you open is large enough to read easily!

As before, the output of your commands must be a well-labelled trace the messages transmitted and received between processes so that it is easy to follow what is happening in your DSS application program.

References

[1] Amazon Web Services. Amazon Simple Storage Service API reference. `https://docs.aws.amazon.com/AmazonS3/latest/API/RESTAPI.html`, 2025.

[2] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 2004.

[3] James F. Kurose and Keith W. Ross. *Computer Networking, A Top-Down Approach*. Pearson, 8th edition, 2021.

[4] J. Reynolds and J. Postel. Request for comments: 1700, assigned numbers. `https://datatracker.ietf.org/doc/html/rfc1700`, 1994.

[5] Zhirong Shen, Yuhui Cai, Keyun Cheng, Patrick P. C. Lee, Xiaolu Li, Yuchong Hu, and Jiwu Shu. A survey of the past, present, and future of erasure coding for storage systems. *ACM Trans. Storage*, 21(1), January 2025.