Distributed Computing

Report of Assignments,
Computer Science – Software Security and Engineering

Contributors:
Amir Mohammad Azimi (5795736@studenti.unige.it)
Parsa Moslem: (5755015@studenti.unige.it)

UNIVERSITÀ DEGLI STUDI
DI GENOVA

Dibris

Table of Contents

PART 1- SCHEDULING SIMULATOR 3

Introduction 4
Implementation 4
Results 6
Conclusion 9

PART 2 – ERASURE CODING 10

Introduction 11
Code 11
Configs 12
What are we supposed to do? 12
Implementation 12
"storage.py" Module Completion 12
Extension 12
Analyze Part 13
Conclusion 21

PEER REVIEW 22

Part 1- Scheduling Simulator

Introduction

Our project embarked on a detailed exploration to model a scenario reflective of real-world server-job allocation dynamics, starting from the ground up with a singular server and queue setup. This initial step laid the foundation for progressively increasing the complexity of our simulation, first by introducing multiple servers and then by assigning each server its own queue. The venture into complexity didn't stop there; we adopted the supermarket model, a strategic approach where, upon the arrival of a job, we choose the least burdened server from a subset of d servers, aiming to optimize the flow and reduce congestion.

The journey took a notable turn with the transition from a traditional First-In-First-Out (FIFO) approach to adopting a Round Robin scheduling method. This method is distinctive in its approach to handling jobs by allocating a fixed time slice for execution, after which jobs exceeding this limit are paused and placed at the end of the queue. This ensures a more democratic distribution of processing time among all tasks.

The primary aim of incorporating the supermarket model alongside the Round Robin scheduling was to scrutinize their combined impact on the system's operational efficiency, particularly focusing on job waiting times and queue management. By simulating an environment where each server is matched with a queue, forming an M/M/n system, and employing the supermarket model's strategy to delegate jobs to the server with the shortest queue, we aimed to highlight the potential advantages of this setup. Additionally, the Round Robin policy, by reintroducing paused jobs at the queue's end, promised an intriguing dynamic to the overall system management.

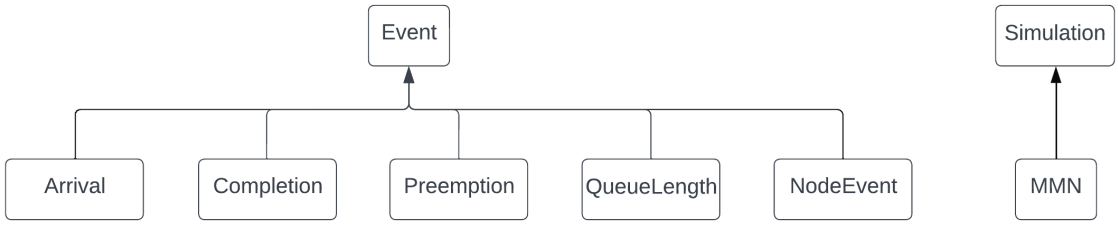
Contrary to initial expectations, the simulation unveiled that the Round Robin method did not lead to reductions in waiting times or queue lengths. This outcome prompted a deeper analysis into the interplay between job scheduling policies and system efficiency.

Implementation

This simulation is implemented using the Python programming language, and some of its famous libraries as random (for generating arrival times following an exponential distribution, sampling d servers among n, …) and matplotlib (for plotting the queue lengths on a graph).
The important variables in this simulation are as follows:

- lambd, showing the arrival rate of the jobs for the system[1][2].
- mu, showing the completion rate of the jobs for the system.
- n, showing the number of servers.
- d, showing the number of choices among n servers.
- timeInterval, the time interval in which the queue lengths will be recorded.
- queueLengths, an array to record the queue length in different time intervals.
- timeSlice, the maximum time that the job will not be preempted.

[1] Lambda is written as lambd because lambda corresponds to a keyword in python programming language.
[2] The arrival rate is multiplied by n, to keep a coherent load for the system.



The key classes in the simulator are Simulation and Event classes.

The Simulation class has 2 important properties, t (time of simulation) and events (list of events), and 2 important methods, schedule (to schedule an event by adding it to the list of events with specific delay) and run (to take the events one by one, proceed of time of simulation and process the event). The Event class has an important method, and process, which will be implemented by its subclasses.

The MMN class is extended from the Simulation class and contains the important variables explained before. In this class, there are two important properties, running (which stores the ID of the running job(s)) and queue (which is the queue(s) of the jobs). There are schedule arrival and schedule completion methods that schedule the events according to an exponential distribution. Finally, we have the queue_len method, which returns the length of a specific queue.

The QueueLength, Arrival, Completion, and Preemption classes are subclasses of the Event class, which each has a different implantation of process method.

In the QueueLength class, the process method, records the length of all queues, to be calculated and plotted on a graph later. Then it schedules another QueueLength event according to the specific time interval.

In the Arrival class, the process method considers the arrival of a new job and records its arrival time, assigns it to any empty server, or adds it to a queue. And schedules the arrival of the next job.

In the Completion class, the process method considers the completion of a job records its completion time, checks if there are any other jobs in the corresponding server, and schedules its completion, otherwise sets the server as free.

In the Preemption class, the process method considers that a job has been run for a specific time, adds it to the end of the queue of that server, and schedules the completion of the next job.

Key steps and highlights in implementing each model:

- **Transition from M/M/1 to M/M/n:**
  To transit from M/M/1 to M/M/n, we changed the running property of the MMN class to be an array instead of a variable, by doing this, the system simulates a situation where multiple jobs can be running, meaning that multiple servers are there. Also, the queue is now an array of queues, so each server has its queue of jobs.
- **Implementation of the Supermarket model:**
  To implement the supermarket model, in the Arrival class, each time the server with the shortest queue is chosen from a sample of d servers among n servers, and the job is assigned to the queue of that server.
- **Integration of round-robin scheduling:**

To implement the round-robin scheduling policy, in the schedule completion method, a comparison has been made to check if the completion time of the job is more than the time slice specified. If so, the job is scheduled as a Preemption event with remaining time, which means that it will simulate its running for specified time slice and then it will be added to the end of the queue and its completion would be scheduled according to the remaining time.

Results

The results of the MM1 simulation were quite like the theoretical expectations of the specific lambdas. Of course, there were slightly different results in the simulation, as the behavior of the jobs cannot be completely predicted.

MM1 Simulation Results:

Lambda	**Average time spent in the system**	
	Theoretical Expectation	**MM1**
0.5	2.0	2.00
0.9	10.0	9.95
0.95	20.0	19.20
0.99	100.0	103.93

After implementing the supermarket model, the average time spent was reduced, especially for higher lambdas and for the simulations that we had higher choices for the system. As visible in the tables below, even for 1 choice, for lambda 0.99, we have almost half of the waiting time regarding to MM1 simulation. But the higher the choices, the lower the average time spent in the system.

It is also notable that the system was run considering there are 1000 servers for a maximum time of 10000. The higher the number of servers and the more we have on maximum time, the better results we get from the simulation.

After implementing the Round Robin job policy, it has been evident that the average waiting time has not been decreasing. Although there is no decrease in average time spent, the Round Robin job policy still makes sure that the jobs get equivalent importance, and one large job is not taking time of the server forever. Because, by preempting long jobs, and giving time for shorter ones, it is now possible to run and complete the short jobs also. The optimal time slice for round robin also has been selected by experimenting with different time slices in the system.

MMN Simulation Results:

- **1 choice:**

Lambda	**Average time spent in the system**	
	Supermarket Model	**Round Robin**

0.5	1.9945625686522086	1.9993549601146734
0.9	9.96937740501342	9.974916920955215
0.95	18.643023096444484	19.17303208006652
0.99	52.647596371984626	51.70575129659926

- **2 choices:**

Lambda	**Average time spent in the system**	
	Supermarket Model	**Round Robin**
0.5	1.439833074733396	1.4392825592377703
0.9	2.677100577312354	2.675373988816492
0.95	3.3999674681072065	3.415638858474325
0.99	5.368234488321302	5.428413240582608

- **5 choices:**

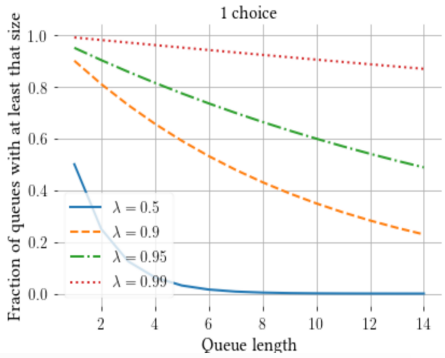
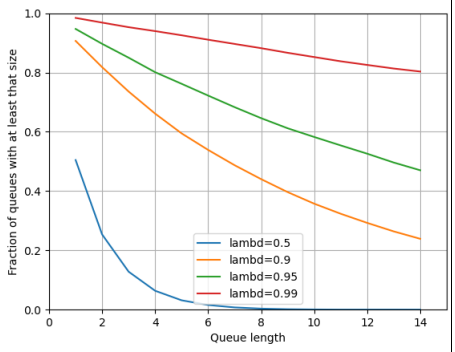
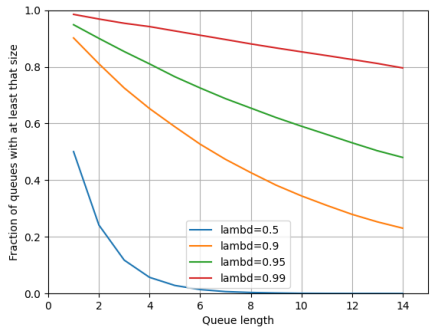
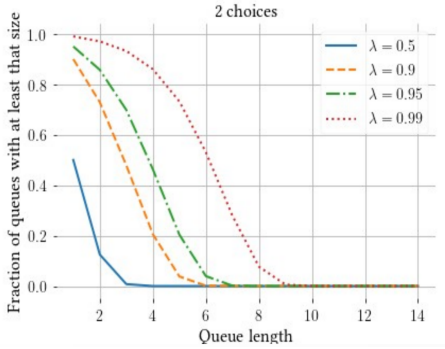
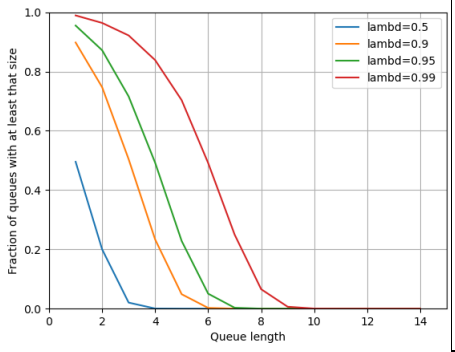
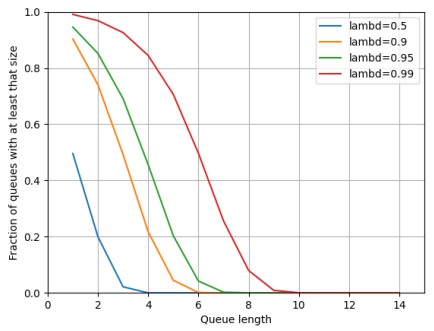
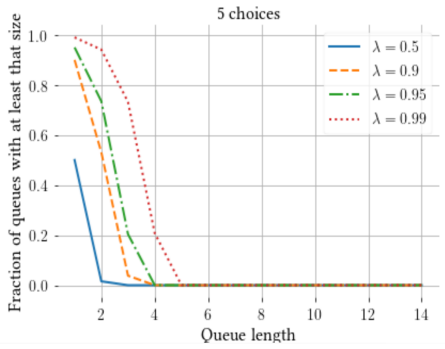
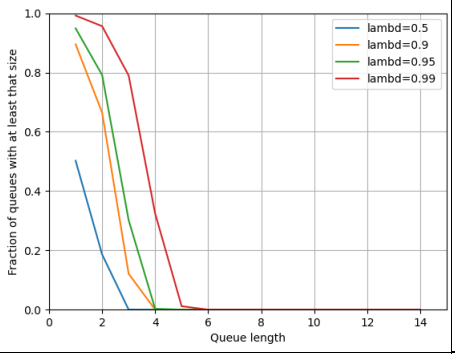
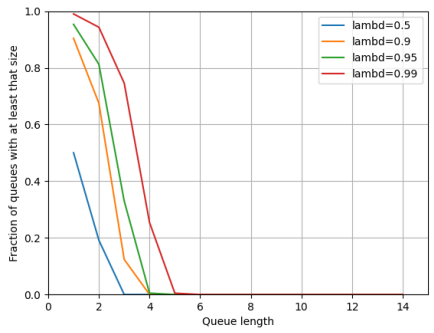
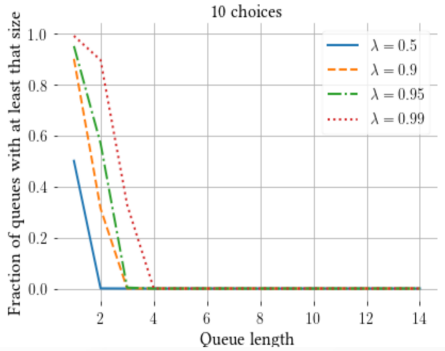
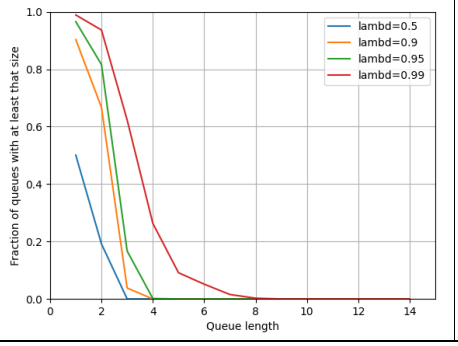
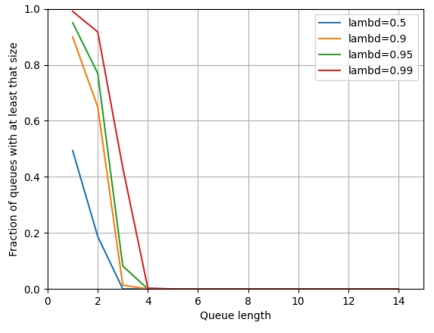
Lambda	**Average time spent in the system**	
	Supermarket Model	**Round Robin**
0.5	1.3816281913405073	1.3828117982519046
0.9	1.8666507354900406	1.8763483644375385
0.95	2.1609424491707485	2.1511977055727294
0.99	2.9935755672355846	3.0228024286491473

- **10 choices:**

Lambda	**Average time spent in the system**	
	Supermarket	**Round Robin**
0.5	1.3824576159753958	1.382673280202601
0.9	1.7360270471049222	1.734644933921397
0.95	1.8756790445063711	1.8801723602930345
0.99	2.4255954973237137	2.3894139926527944

Once again, after plotting the graphs of the queue lengths, it was evident that the results of the supermarket model were quite like the theoretical expectations of the systems. However, there may be some differences in the graphs, especially for a higher number of choices, and that is due to the limits of the simulations. As per theoretical expectations, it is considered that the number of servers, n, is infinitely large, which is not applicable in a simulation.

After implementing the round-robin job policy, its effects were evident also in the graphs of simulations with higher choices, as the queue lengths were slightly shorter than the queue lengths of the supermarket model.

C	Theoretical Expectation	Supermarket Model	Round Robin
1 choice			
2 Choices			
5 choices			
10 choices			

Conclusion

In wrapping up our simulation work, we discovered that MM1 systems performed in line with what we anticipated from theory, confirming the soundness of these models under standard scenarios. Yet, the real potential for system enhancement was revealed through the idea of increasing server counts, which directly contributes to faster job processing and reduced wait times, thereby boosting system efficiency. The integration of the supermarket model further elevates this optimization, strategically directing jobs to servers with shorter queues to alleviate congestion effectively. When combined with the Round Robin job policy, this approach ensures a smoother flow of tasks by preventing the system from being bogged down by longer jobs, thus maintaining shorter queues, and improving overall responsiveness.

The project vividly illustrated the benefits of distributed systems in managing workloads efficiently, especially when employing a thoughtfully chosen job scheduling policy like the supermarket model. This model, alongside potential future enhancements like the Shortest Job First (SJF) and Shortest Remaining Time First (SRTF) algorithms, promises to refine the system's handling of tasks by prioritizing shorter jobs, thereby streamlining operations further.

Our journey through this simulation not only reaffirmed the reliability of established theoretical frameworks but also opened avenues for tangible system improvements, marrying theory with practicality in the quest for optimal system performance. This exploration underlines the importance of adaptive system design, suggesting that even well-established models can benefit from strategic adjustments to meet the challenges of real-world application.

Part 2 – Erasure Coding

Introduction

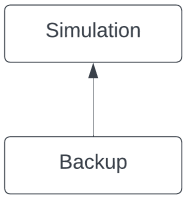
In this assignment, we have a distributed backup system that has two different config files:

1. P2P
2. Client-Server

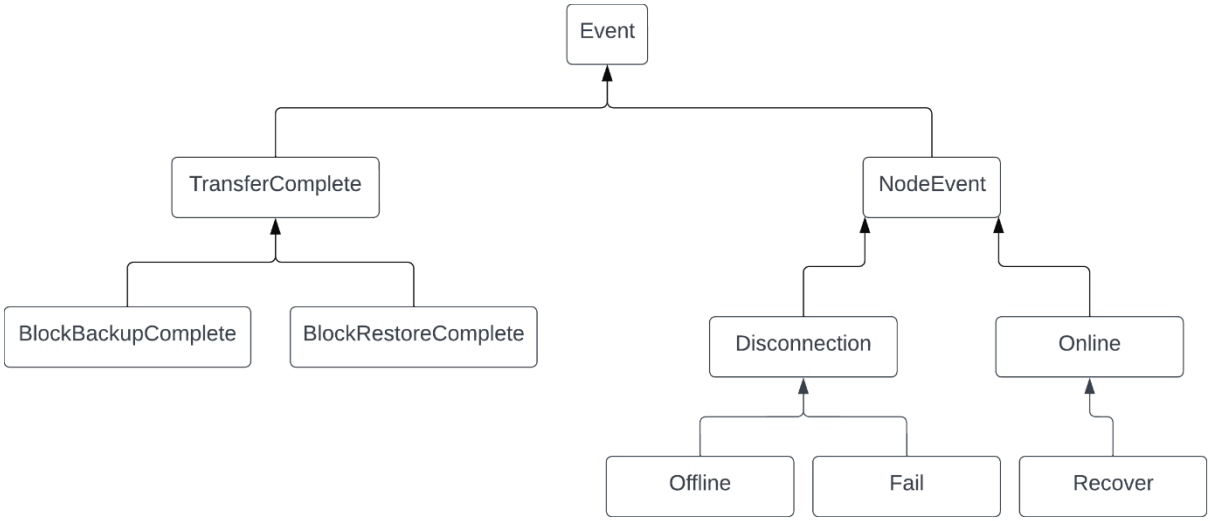
Code

Regardless to the two mentioned config files, this assignment has two main Python modules, Storage.py and discrete_event_sim.py, which the second one is common between both assignments.

To summarize the event succinctly, two diagrams are prepared.



This is the first diagram that consists of the Backup class, which is the core part of the storage.py module and is responsible for the overall flow of the simulation, including creating nodes, scheduling events, and handling interactions between nodes. Another created diagram is illustrating different possible events during the simulation:



NodeEvent part is responsible for the events of any node, while TransferComplete is responsible for the actions that could happen for blocks.

- **How are modules organized?**

  We have created a directory for the second assignment, called ‘storage’. This directory includes all the necessary modules and configurations. Below, is the list of files and their functionalities:

  - **client_server.cfg:** the client-server configuration file, which is given to us.
  - **p2p.cfg:** the P2P configuration file, which is given to us.
  - **storage.py:** the main module for the second assignment which consist of the main simulation logic.

- **p2p.py:** logic which run the simulation with the p2p.cfg is included in this file.
- **client_server.py:** logic which run the simulation with the client_server.cfg is included in this file.
- **test.py:** this module is where we can easily run our simulation, it is only sufficient to change the default value that you have considered (**p2p**, or **client_server**).

```
parser.add_argument("--config", default="client_server", help="You should choose between 'p2p' and 'client_server'")
```

Configs

P2P:
This config file enables system to work as a peer to peer one, and each node can download or upload its own data using another node, while each node has its own events.

Client-Server:
This config file enables system to work as if the available nodes are divided into two types: Client and Servers, which clients want to upload and download their data using the servers.

What are we supposed to do?

For the second assignment, we are supposed to do these steps:

- **Completing the storage.py module to work properly.**
- **Adding an extension to the module.**
- **Analyzing different aspects of the simulation that we have noticed.**

Implementation

“storage.py” Module Completion

There were some sections in the storage.py that were highlighted by ‘…’ and we were supposed to complete those parts. After completing the storage.py module, the program should run properly without either error or specific output (if logs are commented).

Extension

Extension: Modifying the code to implement selfish nodes

Definition: The selfish node is the one who does not care about system performance and redundancy, it just cares about its data that should be uploaded, even if the blocks are already backed up.

Description: Building upon the existing system, we propose enhancing the storage.py module to accommodate selfish node scenarios. This extension will enable our system to operate seamlessly with or without selfish nodes. Additionally, this modification will facilitate the exploration of the impact of varying selfish node percentages on the overall system performance and stability.

Question: The question that is raised up for us is that if the selfish node strategy influences the data loss (lost blocks) during the simulation or not? But there were two **challenges** for us:

1. How to implement selfish node strategy in the system?
2. How we can get the lost blocks for each node during the simulation

Lost blocks Definition: *To find lost blocks, we will calculate* ***not lost blocks*** *for each node, and if this number is lower than* ***k*** *for that node, then the node's data is not recoverable, and we will consider all blocks in that node to the lost ones. These functions work together to provide an overall picture of data availability and potential data loss within the simulation.*

Not lost blocks Conditions: *Not lost blocks are those that have at least one of these conditions.*

Condition	***Description***
Block is locally stored on the node	*The block is accessible.*
Block is backed up to another node	*The block is recoverable.*

Solutions for the mentioned challenges:

1. Modifying upload strategy:

   To implement the selfish mode strategy in the system, we have created a new attribute called 'selfish' in the Node class and set its default value to False (as the nodes are not selfish by default), then we made some of them selfish in the 'p2p.py' and 'client_server.py' modules. These nodes will prioritize uploading their own blocks. This logic is implemented in the 'find_block_to_backup' function.

2. How to get not lost blocks during the simulation?

   get_not_lost_blocks_count: Counts the number of blocks held by a node that are not considered lost, then returns the count variable. We will use this function in the second function.

   get_lost_blocks_count: Counts the total number of lost blocks across all nodes in the system. The 'get_not_lost_blocks_count' function is used within this function.

By using these two functions, we are now able to calculate the total lost blocks during the simulation.

Analyze Part

Client-server Config – Config number 1:

```
[DEFAULT]  # parameters that are the same by default, for all classes
average_lifetime = 1 year
arrival_time = 0

[client]
number = 1
n = 10
k = 8
data_size = 1 GiB
storage_size = 2 GiB
upload_speed = 500 KiB  # per second
download_speed = 2 MiB  # per second
average_uptime = 8 hours
average_downtime = 16 hours
average_recover_time = 3 days

[server]
```

```
number = 10
n = 0
k = 0
data_size = 0 GiB
storage_size = 1 TiB
upload_speed = 100 MiB
download_speed = 100 MiB
average_uptime = 30 days
average_downtime = 2 hours
average_recover_time = 1 day
```

Now, we run the simulation to see what will happen:

```
client-0: local blocks: 10, backed up blocks: 10, remote blocks: 0, lost blocks: 0, is data recoverable? True
server-0: remote blocks: 1
server-1: remote blocks: 1
server-2: remote blocks: 1
server-3: remote blocks: 1
server-4: remote blocks: 1
server-5: remote blocks: 1
server-6: remote blocks: 1
server-7: remote blocks: 1
server-8: remote blocks: 1
server-9: remote blocks: 1
```

By running the simulation, we can see that the client (client-0) has 10 local blocks which all of them are backed up using server-0 to server-9. Each server got 1 block from the client-0. What will happen if we increase the client numbers?

Client-server Config – Config number 2:

```
[DEFAULT]  # parameters that are the same by default, for all classes
average_lifetime = 1 year
arrival_time = 0

[client]
number = 2
n = 10
k = 8
data_size = 1 GiB
storage_size = 2 GiB
upload_speed = 500 KiB  # per second
download_speed = 2 MiB  # per second
average_uptime = 8 hours
average_downtime = 16 hours
average_recover_time = 3 days

[server]
number = 10
n = 0
k = 0
data_size = 0 GiB
storage_size = 1 TiB
upload_speed = 100 MiB
download_speed = 100 MiB
average_uptime = 30 days
average_downtime = 2 hours
average_recover_time = 1 day
```

After running the simulation with the given configuration, we got this output:

```
/usr/bin/python3 /Users/parsa/PycharmProjects/mmn_2nd/storage/test.py
client-0: local blocks: 10, backed up blocks: 10, remote blocks: 0
client-1: local blocks: 10, backed up blocks: 10, remote blocks: 0
server-0: remote blocks: 2
server-1: remote blocks: 2
server-2: remote blocks: 2
server-3: remote blocks: 2
server-4: remote blocks: 2
server-5: remote blocks: 2
server-6: remote blocks: 2
server-7: remote blocks: 2
server-8: remote blocks: 2
server-9: remote blocks: 2
```

Initially, each client stored its own blocks on different servers. As it can be seen, in this case both of clients' data are recoverable because they successfully uploaded their data on servers. Also, as the servers do not have any data by their owns, we did not show the local and backed up, or lost blocks for them.

We run the simulation for the second time:

```
/usr/bin/python3 /Users/parsa/PycharmProjects/mmn_2nd/storage/test.py
client-0: local blocks: 10, backed up blocks: 10, remote blocks: 0
client-1: local blocks: 10, backed up blocks: 10, remote blocks: 1
server-0: remote blocks: 2
server-1: remote blocks: 2
server-2: remote blocks: 2
server-3: remote blocks: 2
server-4: remote blocks: 2
server-5: remote blocks: 2
server-6: remote blocks: 2
server-7: remote blocks: 2
server-8: remote blocks: 2
server-9: remote blocks: 1
```

client-0 decided to upload one of its blocks on the other client node (client-1) instead of server-9, and this is acceptable why it is our system default behavior. (with respect to what it is said on *Aulaweb*)

Here is another situation which both clients decided to upload one of their blocks on each other storage, and there is no block which is uploaded on server-7:

```
/usr/bin/python3 /Users/parsa/PycharmProjects/mmn_2nd/storage/test.py
client-0: local blocks: 10, backed up blocks: 10, remote blocks: 1
client-1: local blocks: 10, backed up blocks: 10, remote blocks: 1
server-0: remote blocks: 2
server-1: remote blocks: 2
server-2: remote blocks: 2
server-3: remote blocks: 2
server-4: remote blocks: 2
server-5: remote blocks: 2
server-6: remote blocks: 2
server-7: remote blocks: 0
server-8: remote blocks: 2
server-9: remote blocks: 2
```

Let's now think about adding selfish node strategy again, what will be happen?

In this case, the simulation will be interrupted by a client node which is decided to be selfish.
Why this is the case?

The error that has occurred during the simulation was this:

```
/usr/bin/python3 /Users/parsa/PycharmProjects/mmn_2nd/storage/test.py
Traceback (most recent call last):
  File "/Users/parsa/PycharmProjects/mmn_2nd/storage/test.py", line 12, in <module>
    client_server.run()
  File "/Users/parsa/PycharmProjects/mmn_2nd/storage/client_server.py", line 42, in run
    sim.run(parse_timespan(args.max_t))
  File "/Users/parsa/PycharmProjects/mmn_2nd/discrete_event_sim.py", line 37, in run
    event.process(self)
  File "/Users/parsa/PycharmProjects/mmn_2nd/storage/storage.py", line 371, in process
    self.update_block_state()
  File "/Users/parsa/PycharmProjects/mmn_2nd/storage/storage.py", line 390, in update_block_state
    assert peer.free_space >= 0
AssertionError
```

As the selfish nodes like to upload their blocks into the servers (even if their blocks are already backed up, by our definition), this selfish node behavior will raise an error sooner or later, which is related to insufficient space on the server to store replicated blocks of the client at some point.

Peer review – revision:
We have decided to check if the 'peer.free_space' is more than 0 or not. After fixing the above error, we now run the simulation to see the results while one or two of clients are selfish:

```
/usr/bin/python3 /Users/parsa/PycharmProjects/mmn_2nd/storage/test.py
client-0: local blocks: 10, backed up blocks: 10, remote blocks: 1
client-1: local blocks: 10, backed up blocks: 10, remote blocks: 1
server-0: remote blocks: 2
server-1: remote blocks: 2
server-2: remote blocks: 2
server-3: remote blocks: 2
server-4: remote blocks: 2
server-5: remote blocks: 2
server-6: remote blocks: 2
server-7: remote blocks: 2
server-8: remote blocks: 2
server-9: remote blocks: 1

Process finished with exit code 0
```

Totally, 21 blocks are uploaded, while there were just 20 blocks. That 1 additional block is for the client-0 (which is a selfish node) and uploaded twice, during the simulation. What will happen if both clients are selfish? Here is the result:

```
/usr/bin/python3 /Users/parsa/PycharmProjects/mmn_2nd/storage/test.py
client-0: local blocks: 10, backed up blocks: 10, remote blocks: 1
client-1: local blocks: 10, backed up blocks: 10, remote blocks: 1
server-0: remote blocks: 2
server-1: remote blocks: 2
server-2: remote blocks: 2
server-3: remote blocks: 2
server-4: remote blocks: 2
server-5: remote blocks: 2
server-6: remote blocks: 2
server-7: remote blocks: 2
server-8: remote blocks: 2
server-9: remote blocks: 2

Process finished with exit code 0
```

Totally, there are 22 blocks that are uploaded, so there are two additional blocks, each of them is uploaded twice, as both clients are selfish.

P2P Config – Config number 1:

```
[peer]
number = 10
n = 10
k = 8
data_size = 1 GiB
storage_size = 10 GiB
upload_speed = 2 MiB  # per second
download_speed = 10 MiB  # per second
average_uptime = 8 hours
average_downtime = 16 hours
average_recover_time = 3 days
average_lifetime = 1 year
arrival_time = 0
```

For this section, we analyze and compare how many blocks will be lost for different lifetimes' simulation either there is no selfish node, with the time that we have 1 or 9 selfish nodes in our system.

As it is obvious in the below plot, if there were no selfish nodes, lost blocks count will be decreased during the simulation time, and whenever we have some selfish nodes, the range of lost blocks will be higher than the normal simulation. In this *Figure 1*, no selfish nodes state is compared with having a selfish node in the simulation:

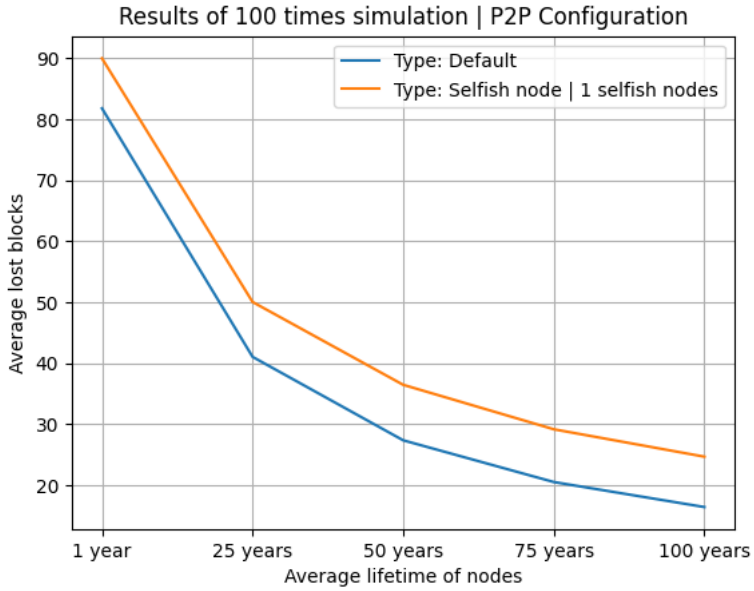


Figure 1

As illustrated in *Figure 2*, the introduction of selfish nodes significantly amplifies the average number of lost blocks. The comparison between 9 selfish nodes and no selfish nodes confirms this trend.

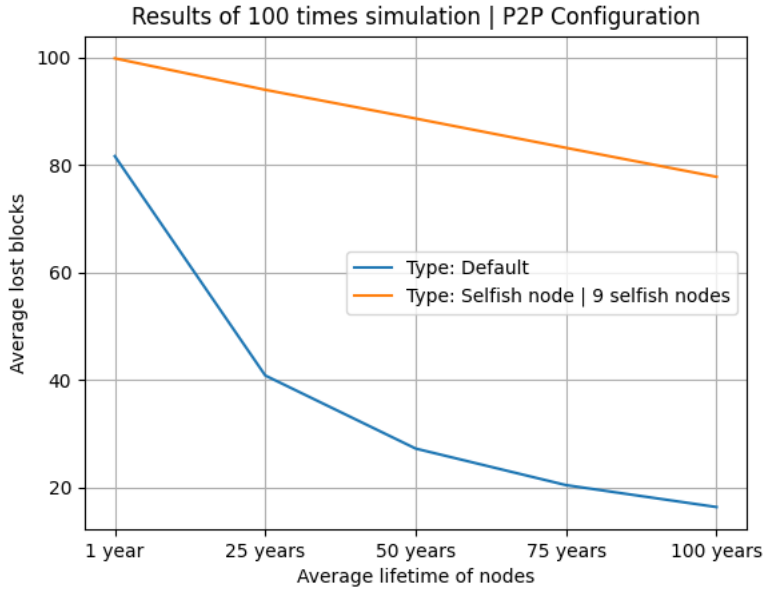


Figure 2

In our default configuration, we set n = 10, indicating that each node stores its data across 10 blocks. The replication factor k = 8 further specifies that at least 8 blocks must be available to reconstruct the node's data. This raises the question of whether k = 8 is the optimal choice for our specific configuration and simulation setup. To investigate this further, we conducted experiments to assess the impact of varying k on the average number of lost blocks in the system. The *Figure 3* is result of our investigation for the 100 years simulation:

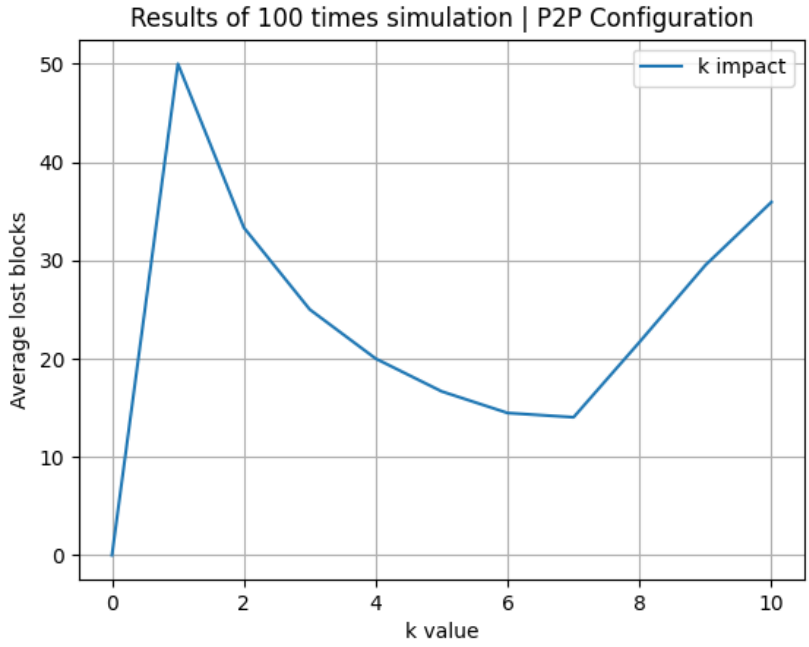


Figure 3

The best choice for the k, is the one which has the least average lost blocks. Look at this table which shows the k and its average lost blocks:

K	Average lost blocks (≈)
0	0
1	50
2	34
3	25
4	20
5	17
6	15
7	**14**
8	21
9	30
10	35

By finishing our investigation on k optimal number, **we have decided to change the k number to 7 in our config.**

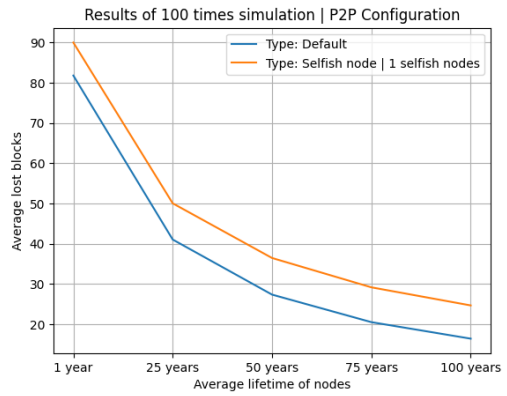
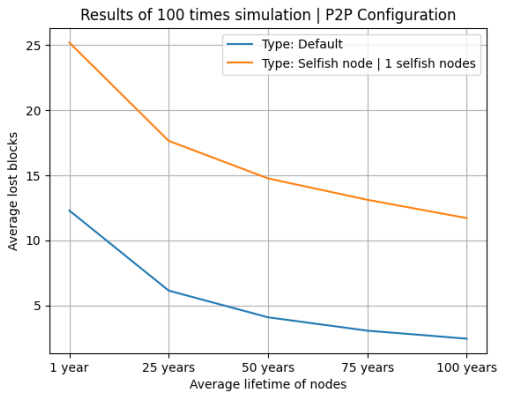
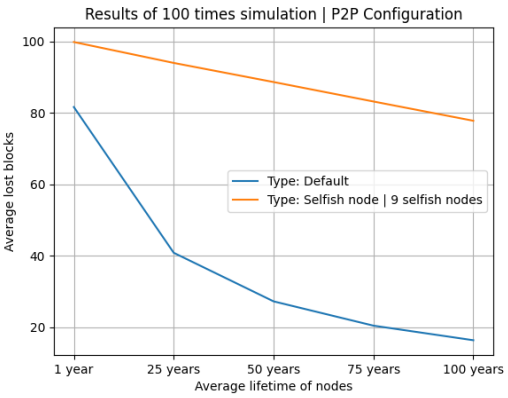
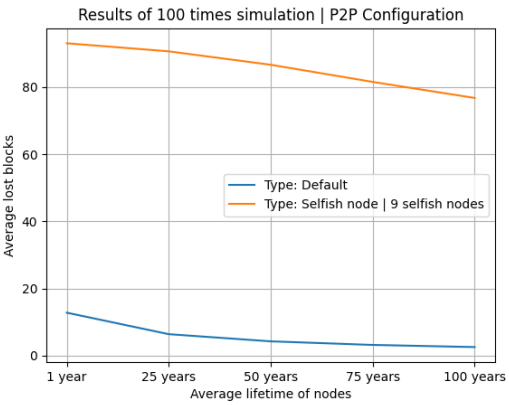
P2P – Config number 2:

```
[peer]
number = 10
n = 10
k = 7
data_size = 1 GiB
```

```
storage_size = 10 GiB
upload_speed = 2 MiB  # per second
download_speed = 10 MiB  # per second
average_uptime = 8 hours
average_downtime = 16 hours
average_recover_time = 3 days
average_lifetime = 1 year
arrival_time = 0
```

We run the simulation to check what will happen after changing the k to its optimal value. The *Table 1* is showing the simulation result compared with the previous configuration for both 1 and 9 selfish nodes simulations:

Table 1 – Comparison between two configurations' results

K = 8	K = 7
	
	

As we expected, the average number of lost blocks significantly reduced compared to the previous configuration.

Conclusion

Client-Server Configuration:

1. Selfish nodes in this configuration results in uploading blocks more than one time.

P2P Configuration:

By analyzing so far, we have understood two important facts:

1. The presence and increase of selfish nodes in the system significantly increases the average number of lost blocks during the simulation.
2. Optimizing the k number for the system significantly reduces the average number of lost blocks during simulation.

Peer Review

It is noteworthy to say that most of the comments that were written on the peer review for our project, were not actionable. They were just some questions which were raised for our reviewers. Nevertheless, we have decided to answer them and if it was possible, apply them in our project.

1. On page 17, Figure 1 It is demonstrated that the count of lost blocks is higher when there is a selfish node than in normal simulation. This makes sense. However, It is also shown that the number of lost blocks decreased as the years went by (It is almost a steep decrease). but this looks illogical, however, if it is counter-intuitive could you explain why

   - The system is working correctly, the reason behind the decrease is that longer-lived nodes have more time to back up their data and recover from failures, leading to fewer lost blocks.

2. On page 16, the report for the implementation of the selfish node strategy is incomplete. It says an error occurred (by also showing a screenshot of terminal output) during the simulation after it was interrupted by a selfish client node. It would have been better if a screenshot of the snippets of code relevant to this occurrence was included. The reader should not find out the cause by going through and reading the whole source code. Furthermore, it doesn't explicitly indicate the number of simulations that have to happen before the simulation fails.

```
2 usages   Parsa Moslem
def find_block_to_back_up(self):
    """Returns the block id of a block that needs backing up, or None if there are none."""
    # servers doesn't have blocks to be uploaded, also they are not selfish.
    if "client" in self.name or "peer" in self.name:
        # if it's selfish node, it doesn't care if it already has uploaded his blocks or not,
        # it prefers to upload his blocks again.
        if self.selfish:
            # for selfish nodes, we back up the blocks that we have locally,
            # even if they are already backed up because the selfish node will not care about the redundancy
            for block_id, held_locally in enumerate(self.local_blocks):
                if held_locally:
                    return block_id
        else:
            # find a block that we have locally but not remotely, default behavior
            # check `enumerate` and `zip`at https://docs.python.org/3/library/functions.html
            for block_id, (held_locally, peer) in enumerate(zip(self.local_blocks, self.backed_up_blocks)):
                if held_locally and not peer:
                    return block_id
    return None
```

   - As it can be seen in the implementation of selfish node behavior, in the selfish node section, we do not check if a specific block in a client is currently uploaded into a host or not. This may result in uploading a block several times into a host and as a result,

the host will be out of space sooner or later, but we managed to solve this problem too. (It is explained in question 4 of this file, peer review section.)

3. On page 18, the table shows that when k=7 and k=8, the resulting lost blocks are 14 and 21 respectively. On page 19, table 1 illustrates that the number of lost blocks in k=8 and k=7 is significantly different when the selfish node is just one. However, the line of graphs is nearly identical for both k=7 and k=8 when the number of selfish nodes is 9. It is not expressed why this happened. This result raises questions on how effective our choice of k as 7 is when the selfish node is different from one This squarely refutes the second point in your conclusion (page 20) which says "Optimizing the k number for the system significantly reduces the average number of lost blocks during simulation."

- The presence of selfish nodes, whether none, one, or multiple, does not affect the process of finding the optimal value of k in our simulation. Regardless of the number of selfish nodes, optimizing k will still contribute to minimizing lost blocks by improving the overall performance, as it is shown in our plots.

4. Page 20 the conclusion for the Client Server Configuration why you defined selfish node strategy as unacceptable is VAGUE. Why is selfish node strategy acceptable in P2P configuration while not in client-server configuration (which you have pointed out fails)? It is not clearly stated why

- Best point of the review! As we were thinking about this question so far, and we saw this question again in the peer review, we have found out something is definitely wrong. We revised the project, and specifically, the selfish node logic implementation. The error occurred here when “peer.free_space” was going below zero:

```
def update_block_state(self):
    owner, peer = self.uploader, self.downloader
    peer.free_space -= owner.block_size
    assert peer.free_space >= 0
    owner.backed_up_blocks[self.block_id] = peer
    peer.remote_blocks_held[owner] = self.block_id
```

We have decided to implement an if condition to prevent this problem in client_server.cfg configuration, so we have changed this function:

```
def update_block_state(self):
    owner, peer = self.uploader, self.downloader
    peer_free_space_after_update = peer.free_space - owner.block_size
    if peer_free_space_after_update <= 0:
        return
    peer.free_space -= owner.block_size
    assert peer.free_space >= 0
    owner.backed_up_blocks[self.block_id] = peer
    peer.remote_blocks_held[owner] = self.block_id
```

It will update block state only if peer.free_space won’t go below zero. So, the implementation is now works for the both p2p.cfg and client_server.cfg configuration files, when we have some selfish nodes in our simulation.

5. On page 18 it is not reasoned out why the number of lost blocks keeps steadily decreasing till k=7 but suddenly takes a u-turn and keeps on increasing after k=8
   - By analyzing the current configuration, we have found out that the optimal k equals to 7. If we change the configuration values, k=7 might not be the best option. So, to find the optimal k, we must make a plot (as we did) to see where we have the least lost blocks.

6. Unlike in the second assignment where you have explicitly listed the different files in the directory that run various logic and their intended functions, In the first part of your report (scheduling) you have left out this important description that can guide anyone fluent in the programming language you have used (Python) to reproduce the results
   - Since there is only one model of the simulation in the first assignment, there is no need to have different files to run. Unlike the second assignment, where you can run the simulation for p2p or client-server configurations, in the first assignments, there is only the supermarket model with the best possible job policy to run.

7. No explicit instructions to run the programs for Assignment 1
   - The assignment 1 can run with default values. It doesn't need any specific instructions to run the file. Although, it can also run with different values, which anyone who reads the code, can understand it easily.

8. In attempting to execute the test.py script for Assignment 2, We encountered errors, primarily due to missing imports of the module that contains the Simulation and Event classes, we had to manually transfer the module to the correct place and import it on the Python files to run the program.

9. NoSectionError was raised while trying to run the test.py file with p2p config, because of missing or improper format in the configuration file. We had to include it in the default parameter to run it properly.
   - Any of these errors are not shown for us, we did not encounter these errors.

10. Running the test.py script Using a p2p configuration and with 100 simulations count did not generate any results or plots, even after an extended waiting period.
    - As this is a simulation that will 100 times run, it takes time to show the plots (after 1 hour for each plot). So, the reviewers might not wait for the sufficient time to see the results.

11. When executing test.py with p2p configuration, no logs are displayed, making it difficult to ascertain whether it is operational or not. I suggest including logs for every simulation run.

- We have decided to show the results using the plots which are shown in this report file, and by executing the simulation, so we thought that would be unnecessary to show the logs, while they are commented in the code and can be uncommented.