Distributed Computing Assignment

Prepared By,

Student Number	**Name**
7097795	**Liyawurage** Kalpa Dilruksha Fernando
6709190	Sahar **Ramezani Jolfaei**

Università di Genova

Table of Contents

A1: Queue Simulation....... 3
Code Completion and Result Plotting....... 3
Results for d=1 correspond to theory....... 4
Reproducing theoretical plot fractional queue length....... 5
Plot for Simulated Results....... 5
Interpretation of Theoretical vs. Simulated Results....... 6
Theoretical Results....... 6
Simulated Results....... 6
Conclusion....... 6
Weibull Distribution....... 6
Extensions and Modifications....... 7
1. Introducing SJF Scheduling Policy....... 7
First In First Out - FIFO (Default Behaviour - Not the Modification)....... 7
Shortest Job First - SJF (Modification 1)....... 8
2. Introducing a deadline for tasks and the Earliest Deadline First (EDF) Scheduling (Modification 2)....... 10
Implementation....... 10
Analysis....... 11
Lambda (λ) vs Miss Rate....... 12
Slack Margin vs Miss Rate....... 12
Key Insights....... 12
A2: Erasure Coding....... 14
Overview....... 14
Implementation....... 14
Extension: Dynamic Erasure Coding....... 15
Key Challenges....... 15
File Hierarchy....... 15
File-by-File Explanation....... 16
storage.py....... 16
1. Key Classes....... 16
2. Dynamically Changing n_active....... 17
3. Main Entry Point....... 18
discrete_event_sim.py....... 18
The Bash Script....... 19
Overall Process from Start to End....... 19
Key Features of n_active:....... 20
Examining the Efficiency....... 20
1. Combined P2P Simulation Results for Small N....... 21
1.1 Failures Over Time....... 21

1.2 Recoveries Over Time.. 22
1.3 Backups Over Time.. 22
2. Combined Client-Server Simulation Results for Small N..22
1. Combined Client-Server Simulation Results for Large N..24
1.1 Failures Over Time... 25
1.2 Recoveries Over Time.. 25
1.3 Backups Over Time.. 25
2. Combined P2P Simulation Results for Large N...26
Conclusion...27

A1: Queue Simulation

Instructions for running the program have been added to Readme.md.

Code Completion and Result Plotting

Completed queue_simulation.py and discrete_event_sim.py.

The following graph was produced: running queue_experiments.sh and queue_plot.sh.

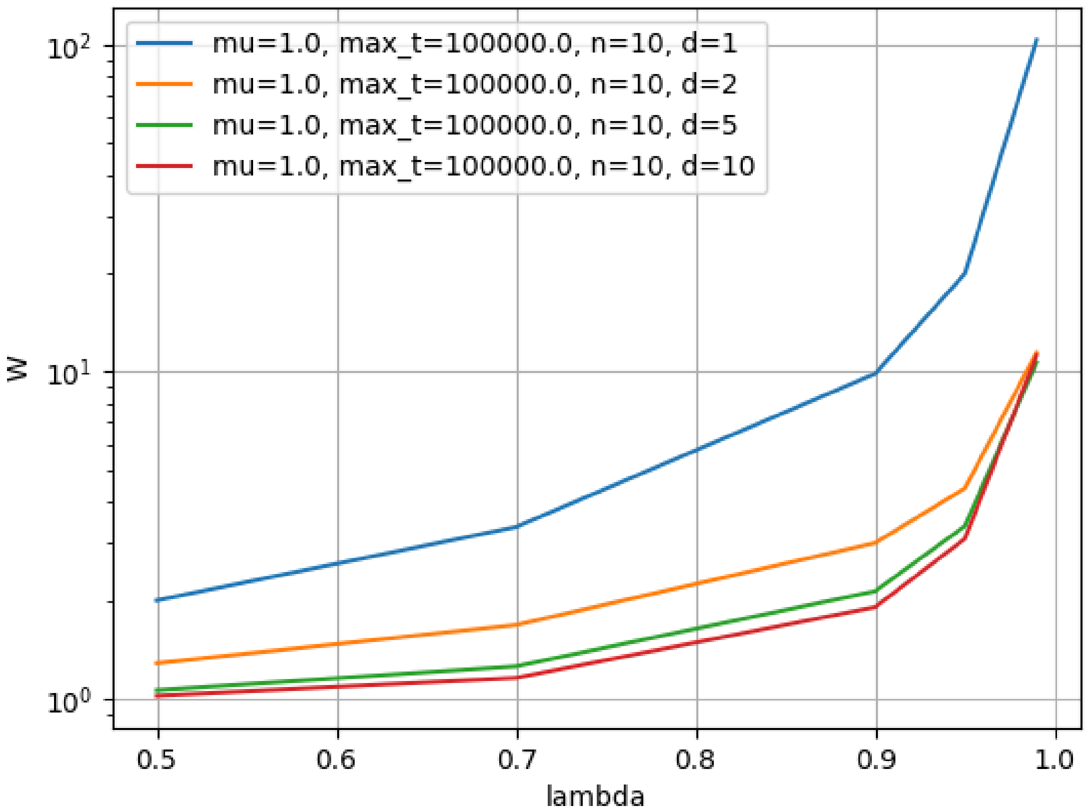


Figure 1 - Graph after completing the code

Results for d=1 correspond to theory.

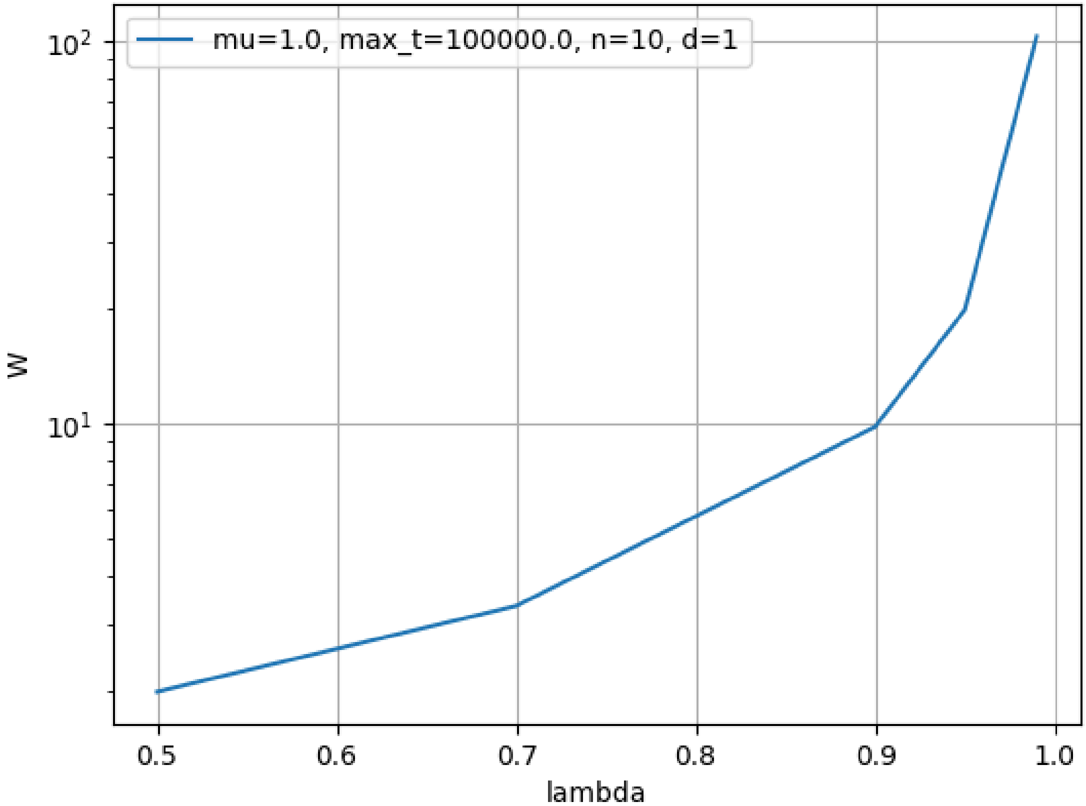


Figure 2 - for d = 1

```
(venv) (base) kalpafernando@MacBook-Pro dcasign % /bin/zsh /Users/kalpaferna
0.5 1
Average time spent in the system: 1.9989106820641112
Theoretical expectation for random server choice (d=1): 2.0
0.7 1
Average time spent in the system: 3.349658116208867
Theoretical expectation for random server choice (d=1): 3.3333333333333333
0.9 1
Average time spent in the system: 9.853737063927914
Theoretical expectation for random server choice (d=1): 10.000000000000002
0.95 1
Average time spent in the system: 19.932170176683282
Theoretical expectation for random server choice (d=1): 19.999999999999982
0.99 1
Average time spent in the system: 103.5894601254234
Theoretical expectation for random server choice (d=1): 99.99999999999991
```

Figure 3 - for d = 1 (approximately similar to theoretical values)

Reproducing theoretical plot fractional queue length.

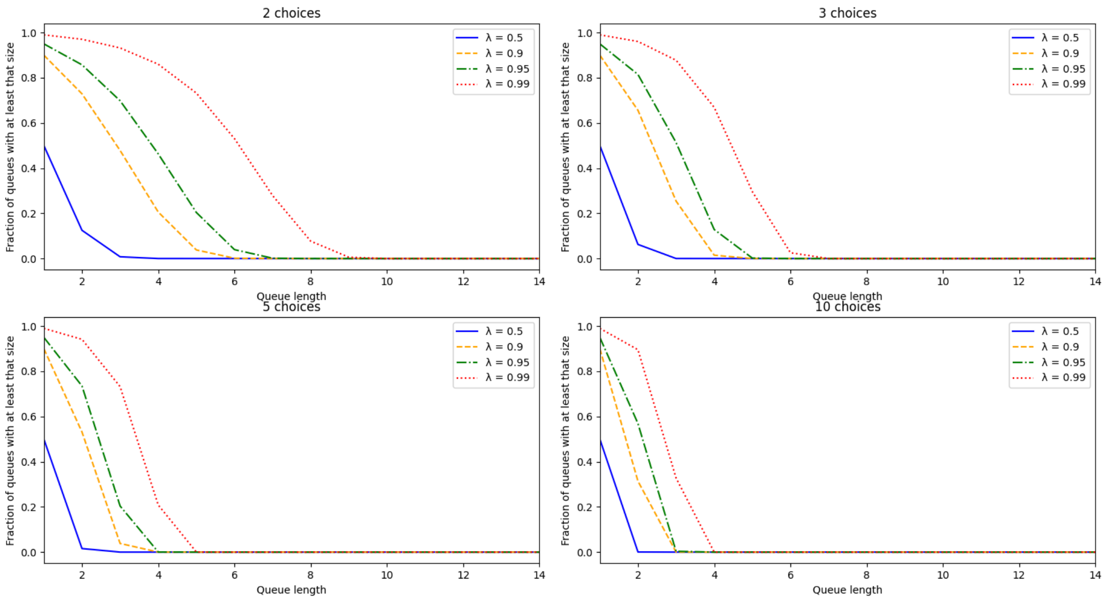


Figure 4 - Theoretical Queue

Plot for Simulated Results

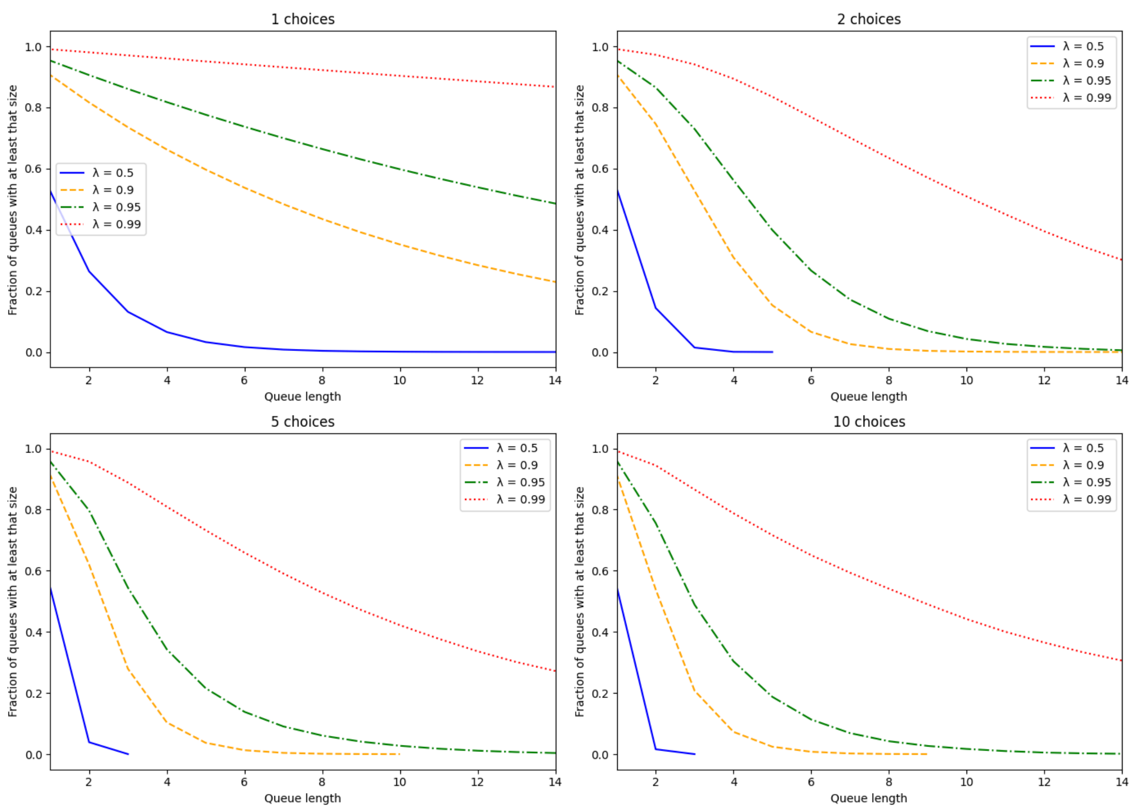


Figure 5 - Simulated Queue

Interpretation of Theoretical vs. Simulated Results

Theoretical Results

Theoretical results are based on the queueing model using the formula $\lambda^{\frac{d^i-1}{d-1}}$. Key observations include a sharp reduction in the fraction of long queues as the number of choices (d) increases. For low arrival rates (λ = 0.5), queues stabilize quickly with minimal long queues. Long queues were reduced for higher arrival rates (λ = 0.9, 0.95, 0.99) with higher d values (e.g., d = 5 or d = 10).

Simulated Results

Simulated results incorporate stochastic elements, reflecting real-world randomness. Similar trends to theoretical results are observed, with higher d reducing the fraction of long queues. However, reductions are less pronounced, particularly for smaller d values, and deviations from theory are evident at high λ (e.g., λ = 0.95, 0.99).

Conclusion

Both theoretical and simulation results confirm higher d's effectiveness in reducing large queues. Theoretical models provide an upper performance bound, while simulations highlight practical implications under real-world conditions. For low loads (λ = 0.5), small d values suffice, while high loads (λ = 0.95, 0.99) require more significant d (e.g., d = 10) to prevent bottlenecks.

Weibull Distribution

A Comparison of Expovariate and Weibull. W is very similarly close.

Expovariate	**Weibull Shape = 1**
0.5,1,100000.0,10,1,1.9989106820641112 0.5,1,100000.0,10,2,1.2845078825756266 0.5,1,100000.0,10,5,1.0625845143708383 0.5,1,100000.0,10,10,1.0225459218853559 0.7,1,100000.0,10,1,3.349658116208867 0.7,1,100000.0,10,2,1.681243913034345 0.7,1,100000.0,10,5,1.2565913872919137 0.7,1,100000.0,10,10,1.158013624072451 0.9,1,100000.0,10,1,9.853737063927914 0.9,1,100000.0,10,2,2.9944469930188156 0.9,1,100000.0,10,5,2.13119152025219 0.9,1,100000.0,10,10,1.9057525792797787 0.95,1,100000.0,10,1,19.932170176683282 0.95,1,100000.0,10,2,4.39967650261775 0.95,1,100000.0,10,5,3.3714449479630257 0.95,1,100000.0,10,10,3.0837908180711 0.99,1,100000.0,10,1,103.5894601254234 0.99,1,100000.0,10,2,11.466948044554742 0.99,1,100000.0,10,5,10.688830699488252 0.99,1,100000.0,10,10,11.220173515800353	0.5,1,100000.0,10,1,1.998920717827757 0.5,1,100000.0,10,2,1.2845056201798914 0.5,1,100000.0,10,5,1.062570218310457 0.5,1,100000.0,10,10,1.0225424717564837 0.7,1,100000.0,10,1,3.3496643258218533 0.7,1,100000.0,10,2,1.681250274527287 0.7,1,100000.0,10,5,1.2565917786134648 0.7,1,100000.0,10,10,1.1580119797881068 0.9,1,100000.0,10,1,9.853713621242688 0.9,1,100000.0,10,2,2.9944457853104396 0.9,1,100000.0,10,5,2.131187674999965 0.9,1,100000.0,10,10,1.9057519787350183 0.95,1,100000.0,10,1,19.932149949461394 0.95,1,100000.0,10,2,4.399633929094588 0.95,1,100000.0,10,5,3.3714540473157557 0.95,1,100000.0,10,10,3.0837901245266996 0.99,1,100000.0,10,1,103.58941593225732 0.99,1,100000.0,10,2,11.467008934922367 0.99,1,100000.0,10,5,10.688808398457859 0.99,1,100000.0,10,10,11.220148710349122

Extensions and Modifications

1. Introducing SJF Scheduling Policy

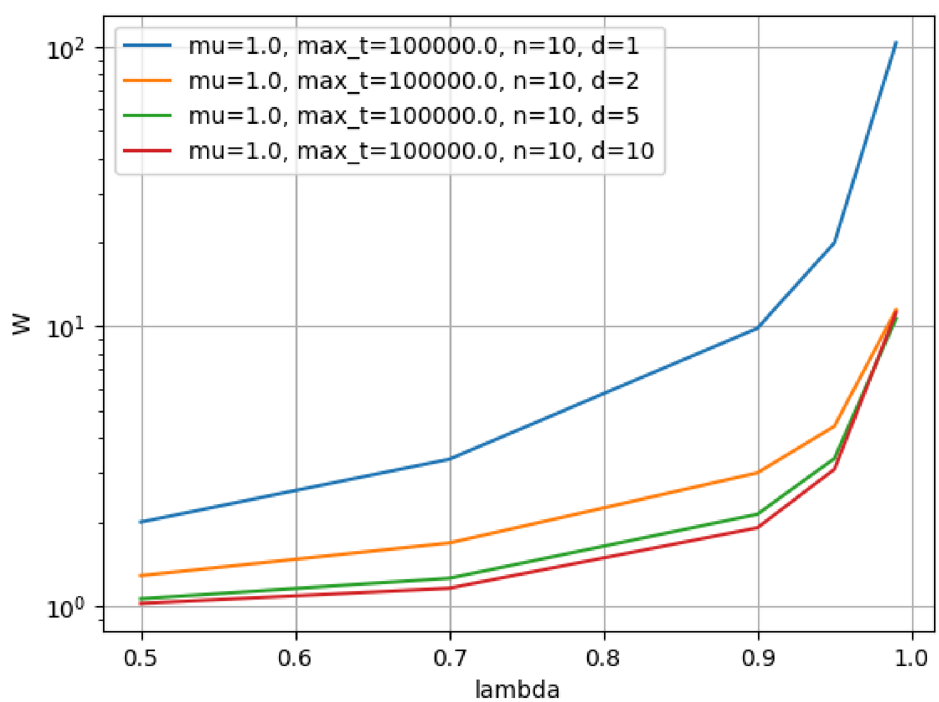
What is the motivation for this extension?

Introducing the Shortest Job First (SJF) scheduling policy aims to address the inefficiencies of the default First-In-First-Out (FIFO) approach. This approach processes tasks in order of arrival without prioritization, leading to long waiting times for smaller or urgent tasks, especially under high loads.

First In First Out - FIFO (Default Behaviour - Not the Modification)

- **Behavior**:
  - Tasks are processed in the order they arrive, without prioritization.
  - No consideration is given to task size.
  - It is predictable and straightforward but may result in long waiting times for smaller or urgent tasks when large tasks dominate the queue.

- **Real-World Examples**:
  - Supermarket checkout lines and basic task queues in systems.
  - Results:



Shortest Job First - SJF (Modification 1)

- **Implementation:**

In the Arrival(Event) class, process method, records the service time.

With the service time saved at arrival, the scheduler can later compare jobs in the queue using their predetermined service times.

```
def process(self, sim: Queues):
    """Process an arrival of a new job at the simulation."""
    sim.arrivals[self.id] = sim.t  # Log the arrival time


    sample_queues = sample(range(sim.n), sim.d)  # Choose d queues at random

    service_time = expovariate(sim.mu)

    sim.service_times[self.id] = service_time

    queue_index = min(sample_queues, key=sim.queue_len)
```

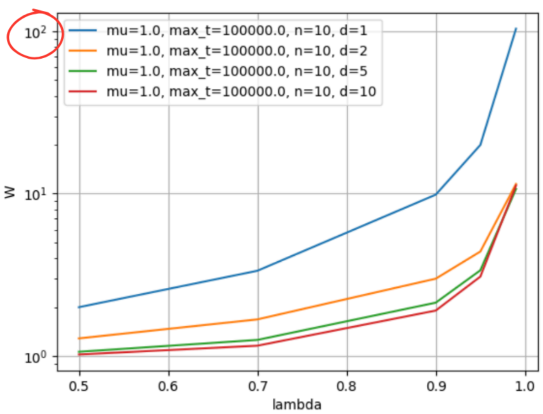
In **Queues(Simulation), schedule_completion method** uses the service_time to schedule completion.

```
def schedule_completion(self, job_id, queue_index):

    completion_delay = self.service_times[job_id]

    # Then schedule the completion event:
    self.schedule(completion_delay, Completion(job_id, queue_index))
```

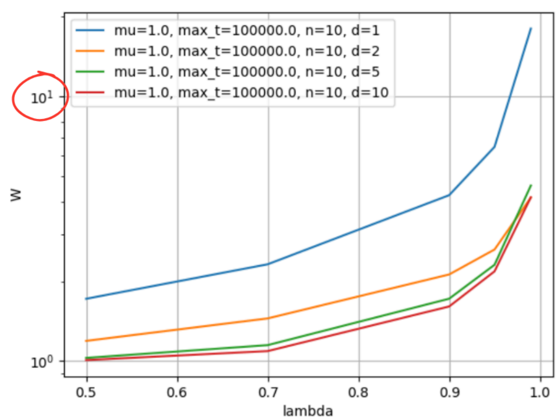
In the Completion(Event): class, Service the Shortest Job First

```
class Completion(Event):

    def process(self, sim: Queues):
        queue_index = self.queue_index
        assert sim.running[queue_index] == self.job_id  # the job must be the one running
        sim.completions[self.job_id] = sim.t

        queue = sim.queues[queue_index]

        if queue:  # If queue is not empty, choose next job based on scheduling type
            if sim.scheduling_type == SchedulingType.SJF:
                # Choose job with the shortest service time
                new_job_id = min(queue, key=lambda job: sim.service_times[job])
                queue.remove(new_job_id)  # Remove it from the queue
            else:  # FIFO fallback
                new_job_id = queue.popleft()
```

- **Behavior**:
  - Prioritizes tasks with the shortest service times.
  - Minimizes the overall waiting time by quickly completing smaller jobs.
- **Real-World Examples**:
  - CPU scheduling for batch processing and packet prioritization in networks.
- **Results**:



Before



After

  - **Significant Drop in Waiting Times vs. FIFO**
    With SJF implementation, waiting times are **much lower** than FIFO — **close to 2×** improvement.
  - **Potential Starvation of Long Tasks**
    By always prioritizing shorter jobs, SJF runs the risk that large jobs wait longer—*though in non-preemptive systems, "starvation" is less acute than in a preemptive scenario.* Still, under heavy load, short jobs dominate the queue, pushing bigger tasks further back.

Overall, **SJF** achieves **lower average waiting times** than **FIFO** across all λ, with the gap widening at high arrival rates.

2. Introducing a deadline for tasks and the Earliest Deadline First (EDF) Scheduling (Modification 2).

What is the motivation for this extension?

Simulate scenarios where tasks have deadlines that must be met. Critical systems, such as healthcare often prioritize meeting deadlines. This helps analyze how well the system handles task deadlines under different configurations.

Implementation

- Deadline Related parameters: to control deadline mode & slack margin

```
# New Modes
self.deadline_mode = deadline_mode

if scheduling_type == SchedulingType.EDF and not deadline_mode:
    logging.warning("Scheduling type EDF requires deadline mode. Switching to deadline mode.")
    self.deadline_mode = deadline_mode

self.schedule(expovariate(lambd), Arrival(0))

# Deadline mode
if self.deadline_mode:
    self.slack_margin = slack_margin
    self.deadline_misses = 0
    self.deadlines = {}

# Scheduling type
self.scheduling_type = scheduling_type
```

- **Arrival(Event),process: Modified the code to add deadlines.**

```
class Arrival(Event):
    """Event representing the arrival of a new job."""

    def __init__(self, job_id):
        self.id = job_id
        self.deadline = None

    def process(self, sim: Queues):
        """Process an arrival of a new job at the simulation."""
        sim.arrivals[self.id] = sim.t  # Log the arrival time

        if sim.deadline_mode:
            job_service_time = expovariate(sim.mu)
            sim.service_times[self.id] = job_service_time

            self.deadline = sim.t + job_service_time * sim.slack_margin
            sim.deadlines[self.id] = self.deadline

        sample_queues = sample(range(sim.n), sim.d)  # Choose d queues at random

        queue_index = min(sample_queues, key=sim.queue_len)
```

- Each task is assigned a deadline according to its job size.

```
self.deadline = sim.t + job_service_time * sim.slack_margin
sim.deadlines[self.id] = self.deadline
```

- Adding *expected service time x slack margin determines a task's deadline*.

- **Slack margin** is an added buffer (a multiplier) on a job's estimated service time used to set its deadline, providing extra leeway so that tasks can finish without missing their deadlines if the system runs slightly slower or faces unexpected delays.

- **sim.t > sim.deadline ?.**

  Track the number of tasks that miss their deadlines (deadline_misses).

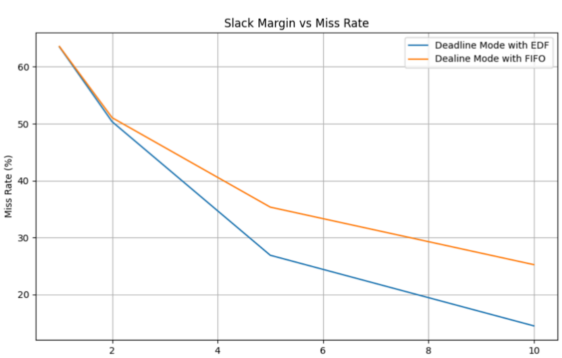
```
def process(self, sim: Queues):  Kalpa D. Fernando *
    queue_index = self.queue_index
    assert sim.running[queue_index] == self.job_id  # the job must be the one running
    sim.completions[self.job_id] = sim.t

    if sim.deadline_mode:
        if sim.t > sim.deadlines[self.job_id]:
            sim.deadline_misses += 1
```

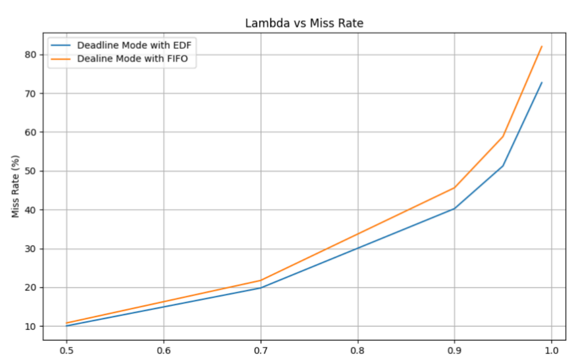
Analysis

we experimented with the following parameters;

```
SCHEDULING="EDF" and “FIFO”
for SLACK_MARGIN in 1 2 5 10; do
 for LAMBD in 0.5 0.7 0.9 0.95 0.99; do
  for D in 1 2 5 10; do
```

Below are the results;





We evaluated both FIFO and EDF scheduling under deadline mode, varying λ (arrival rate), the slack margin, and d. The main metrics are deadline miss rate and how the system reacts to changing workloads.

Lambda (λ) vs Miss Rate

As λ increases, both FIFO and EDF see more congestion and rising miss rates. However, EDF consistently reports fewer misses, because it prioritizes the most urgent (earliest-deadline) jobs. This advantage grows especially at moderate loads, where EDF's scheduling makes a noticeable difference.

Slack Margin vs Miss Rate

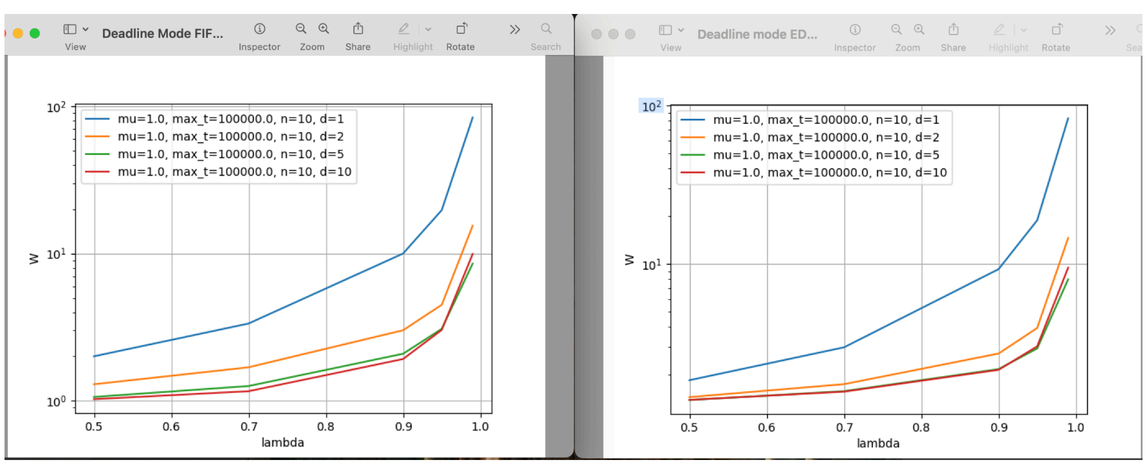
Increasing the slack margin from 1 to 10 significantly decreases the miss rates for both approaches. The graphs show that **EDF's** miss rate declines more sharply. Hence, giving each job a larger deadline buffer helps EDF more effectively meet its deadlines.

Key Insights

1. **System Efficiency with Deadline Mode**
    - **EDF** prioritizes urgent tasks, **yielding fewer misses than FIFO**.
    - Larger slack margins reduce miss rates across the board, and EDF leverages this margin more effectively.

2. **Reaction to Workload Changes**
    - As arrival rates increase, deadline misses inevitably climb, but EDF's focus on earlier deadlines curbs these misses compared to FIFO.
    - Higher d improves both methods by spreading the load, but EDF remains the winner for meeting deadlines.

Average Wait times: Almost the same!.



Conclusion

Based on the results, **EDF** consistently provides a lower miss rate than FIFO across all tested arrival rates and slack margins. Its core advantage is prioritizing jobs that must be finished as soon as possible, making it a choice for deadline-driven scenarios.

A2: Erasure Coding

Overview

In this assignment, we have a **distributed backup system** configured in two modes: **Peer-to-Peer (P2P)** and **Client-Server**. The tasks included completing the implementation, extending the functionality, and analyzing performance.

Implementation

Two main Python modules formed the backbone of the system:

1. **storage.py**: This module contains the core simulation logic.
2. **discrete_event_sim.py**: Shared between assignments, it is for event-driven simulation.

The **storage.py** module was initially incomplete, with several placeholders. The primary task was to complete these sections, ensuring the program could run without errors or unintended output.

This system simulates data backup and recovery in a network of nodes. Each node uses erasure coding parameters:

- **n**: total number of blocks into which the data is split (k + m).
- **k**: number of data blocks necessary to reconstruct the data.
  (Hence, there are *n - k* “redundant” blocks.)

However, there is a twist (added for the extension part of our project): a node can choose a smaller “effective” or “active” number of blocks to use at any given time, denoted by **n_active**. Instead of always using all n blocks, the node may initially use a subset (e.g., 5 out of 10 total blocks), then dynamically add more blocks as needed.

Why do this? It demonstrates a scenario where a node may start with a smaller redundancy level and **dynamically increase** that redundancy (by incrementing `n_active`) when the system detects it is in danger of data loss. This feature is an optional “extension” meant to adapt to changes in system reliability.

Extension: Dynamic Erasure Coding

What is the motivation for this extension?

Two approaches were considered for dynamically changing n and k based on the system's needs:

1. **Traditional Reed-Solomon Erasure Coding**:
    - In this model, values are fixed.
    - Adjusting redundancy dynamically involves starting with a large, and then backing up fewer blocks (to the maximum n blocks).
2. **Dynamic Erasure Coding Strategies**:
    - Altering dynamically requires re-encoding the data. This process necessitates having a full copy of the data available for re-encoding and must occur at a time when all data is accessible.
    - Alternatively, employing a non-deterministic erasure coding strategy allows more flexibility but does not guarantee a specific number of blocks to recover the data.

Key Challenges

1. **Dynamic Adjustment (in case of using the first approach)**:
    - Implemented logic to adjust the number of total blocks dynamically based on system needs.
    - Less flexibility in changing n and k compared to the second approach.
2. **Re-encoding for Changing (in case of using the second approach)**:
    - Re-encoding everything is **computing-intensive**.
    - **Enhanced flexibility** in redundancy management vs **computing-intensive** is a trade-off to be considered.

Because the second approach was computing-intensive, we chose to go with the first approach and started with a larger number of n and backed up only a fraction of it in a way that n >= n' >= k.

File Hierarchy

- `storage.py`
    - Main simulation driver and entry point.
- `discrete_event_sim.py`
    - Provides the discrete-event simulation framework.
- `Configuration Files`
    - `p2p.cfg`: Settings for peer-to-peer simulations (e.g., number of peers, speeds, n, k, etc.).
    - `client_server.cfg`: Settings for a mixed client-server simulation.
- `storage.sh` (Bash script)

    - Example script showing how to invoke `storage.py` with different scenarios and flags (e.g., `--n-active`). This way, it is easier for everyone to run the code and see the results.
    - Creates `logs/` and `plots/` directories, then saves logs and plots there.
- **`logs/`**
    - Directory where `.log` files from simulations are stored.
- **`plots/`**
    - Directory where `.png` plots (failures over time, recoveries, backups) are saved.

File-by-File Explanation

`storage.py`

This is the main module that sets up and runs the simulation.

1. Key Classes

1. **Backup(Simulation)**
    - Manages the overall scheduling of node events, data backups, restorations, and checks for redundancy.
    - The most important method for our focus:
      `check_redundancy()`:
        - Periodically called (via a `RedundancyCheckEvent`) to see if the system is stable and is not in a danger zone.
        - If it detects that a node has few blocks (i.e., is in danger of data loss), it can **increase** that node's `n_active` up to the maximum of `n`.
        - Once `n_active` is increased, new backups get scheduled.
2. **Node**
    - A `@dataclass` that encapsulates a node's parameters:
        - **n** (total blocks)
        - **k** (blocks needed for recovery)
        - **n_active** (actual number of blocks in use at a given point)
        - **Tolerance** -> It is saying that less than what number of redundant blocks puts us in a danger zone.
        - network speeds, storage limits, lifetime, etc.
    - By default, `n_active` is set to `n` if you do not specify anything in the config. If your config has `n_active = -1`, or if the user passes `--n-active -1`, the system sets `n_active` to `n` in `__post_init__()`.
    - **`set_n_active(new_n_active)`**:
        - Increases `n_active` to `new_n_active`.
        - Sets the newly "activated" blocks in `local_blocks[...] = True`.

        ■ Decreases `free_space` by exactly the size needed for the extra blocks.

3. **Events** (`NodeEvent`, `Online`, `Offline`, `Fail`, `BlockBackupComplete`, `BlockRestoreComplete`, etc.)
    ○ Each event class has a `process(self, sim: Backup)` method that modifies the system's state.
    ○ An important event for our focus:
        ■ **`RedundancyCheckEvent`**: calls `sim.check_redundancy()` and re-schedules itself in 1 week.

2. Dynamically Changing `n_active`

- **Initial Setting:**
  When a node is created, `n_active` is initialized in `__post_init__()`.
    ○ If `n_active` from the config or command line is `-1` or `None`, it defaults to `n`.
    ○ If you pass `--n-active 5`, for example, it sets `n_active = 5` in all nodes that read that config.
- **During the Simulation (`check_redundancy`):**
  Here's the core snippet:

```
def check_redundancy(self):
    """
    Checks redundancy.
    If we find that many nodes are failing or overall redundancy is below a threshold,
    we increase n_active on certain nodes.
    """

    # Suppose we want to ensure each node is using at least half of its n if the system is unstable:
    # (You'd define your own logic or threshold condition here.)
    for node in self.nodes:
        if node.failed:
            continue  # skip failed nodes

        blocks_we_have = sum(node.local_blocks) + sum(peer is not None for peer in node.backed_up_blocks)
        if blocks_we_have >= node.k and blocks_we_have - node.k < node.tolerance: # Did not fail and the num
            desired_n_active = min(node.n_active + (node.tolerance - blocks_we_have - node.k), node.n)
            additional = desired_n_active - node.n_active
            space_for_new_blocks = node.block_size * additional
            free_space = node.free_space - space_for_new_blocks
            if desired_n_active > node.n_active and free_space >= 0:
                self.log_info(f"Increasing n_active for {node} from {node.n_active} to {desired_n_active}")
                node.set_n_active(desired_n_active)
                # Trigger new backups
                if node.online:  # Ensure the node is online before scheduling uploads
                    node.schedule_next_upload(self)
```

- If the node has fewer blocks than `k + tolerance`, the system sees that as a "danger" and tries to add more redundancy.
- It bumps `n_active` to k + tolerance, up to the maximum `n`, calls `node.set_n_active(...)`, and triggers new backups (uploads).

`set_n_active(self, new_n_active)`:

```python
def set_n_active(self, new_n_active: int):
    """
    Increase n_active from the current value up to new_n_active (<= n).
    This means we now treat blocks in [old_n_active..new_n_active-1] as 'active' too.
    """
    if new_n_active <= self.n_active:
        return  # do nothing if it's not actually an increase

    additional = new_n_active - self.n_active
    self.n_active = new_n_active
    # Mark those new blocks as locally held
    for b in range(new_n_active - additional, new_n_active):
        self.local_blocks[b] = True

    # Adjust our free space usage
    space_for_new_blocks = self.block_size * additional
    self.free_space -= space_for_new_blocks
    assert self.free_space >= 0
```

- This method expands the range of blocks in `local_blocks` that are set to `True`. Effectively, we're telling the node:
  "Now you want to be responsible for storing (and eventually backing up) those newly active parity blocks as well."
- The node's `free_space` decreases accordingly since these extra blocks need local storage.

3. Main Entry Point

- The `main()` function:
    1. Parses command-line arguments (`--max-t`, `--verbose`, `--n-active` `--tolerance`).
    2. If `--n-active` is specified, it also schedules a `RedundancyCheckEvent` for 1 week.

`discrete_event_sim.py`

This file defines a **discrete-event simulation** framework.

Plotting Functions:

- After finishing the simulation, the code decides which logs to parse and auto-generates some plots (failures, recoveries, backups).

The Bash Script

```
# Run the storage simulation with the provided configuration
python3 storage.py p2p.cfg --max-t "100 years" --verbose > logs/simulation_p2p.log 2>&1
python3 storage.py client_server.cfg --max-t "100 years" --verbose > logs/simulation_client_server.log 2>&1

echo "Simulation result is saved in simulation_p2p.log and simulation_client_server.log file."

# Run the storage simulation with the provided configuration
python3 storage.py p2p.cfg --max-t "100 years" --verbose --n-active 5 --tolerance 1 > logs/extension_simulation_p2p.log 2>&1
python3 storage.py client_server.cfg --max-t "100 years" --verbose --n-active 5 --tolerance 1 > logs/extension_simulation_client_server.log 2>&1
```

1. The first two lines run the simulation without `n_active` (so effectively `n_active = n`).
2. The last two lines illustrate the "dynamic n" extension:
    - `--n-active 5` overrides the default. In the simulation, each node is told to only use 5 blocks out of maximum n.
    - The redundancy check can raise that `n_active` from 5 closer to maximum n if the node encounters data-loss danger during the simulation.

Overall Process from Start to End

1. **Node Creation**
2. **Simulation Start**:
    - Each node is initially offline.
    - The simulation schedules the node's first "Online" event at `backup time`.
3. **Online Event**:
    - When a node comes online, it tries to upload blocks to peers that do not have them (or restore missing blocks from peers that do).
4. **Failures & Recoveries**:
    - The node can go offline or fail. If it fails, it loses all local data. Later, a "Recover" event can bring it back online.
5. **Redundancy Checking**:
    - If `--n-active` was passed (i.e., the "extension scenario"), the simulation schedules a periodic `RedundancyCheckEvent()` (every 1 week).
    - That calls `Backup.check_redundancy()`, which checks each node's block count. If `blocks_we_have >= node.k and blocks_we_have - node.k < node.tolerance`, the simulation calls `node.set_n_active(...)` to increase `n_active`.
6. **Increasing `n_active`**:
    - The node "activates" new blocks locally; it shrinks `free_space` and then tries to schedule new backups to peers.
    - Over time, each node can gradually raise its redundancy from an initially smaller `n_active` up to `n`.
7. **Simulation End**:
    - The simulation runs until the specified `max_t` or until no more events remain.

- Results (failures, recoveries, data loss, etc.) get logged to `.log` files, and charts are produced in the `plots` folder.

Key Features of `n_active`:

- **`n_active`** is the "live" or "in-use" portion of the total number of blocks (`n`).
- By default, `n_active = n`. But you can explicitly set a lower value (e.g., 5) to reduce overhead redundancy.
- During runtime, **`check_redundancy()`** can detect that a node is in trouble and **increment** `n_active` automatically.
- The method **`set_n_active(new_n_active)`** is the place where the code physically toggles additional blocks to `True` in `local_blocks` and recalculates free space.
- The result is a system that starts with low overhead but can "turn on" more blocks if the situation worsens.

The simulation code demonstrates how to move from a static erasure-coding setup with fixed (n, k) to a more flexible scheme in which `n_active` can be adapted at runtime—helping to mitigate data loss under high-failure scenarios.

Examining the Efficiency

In the following, **n is relatively small** (n = 10) and we will inspect if, in small amounts of n, we have a difference in both methods.

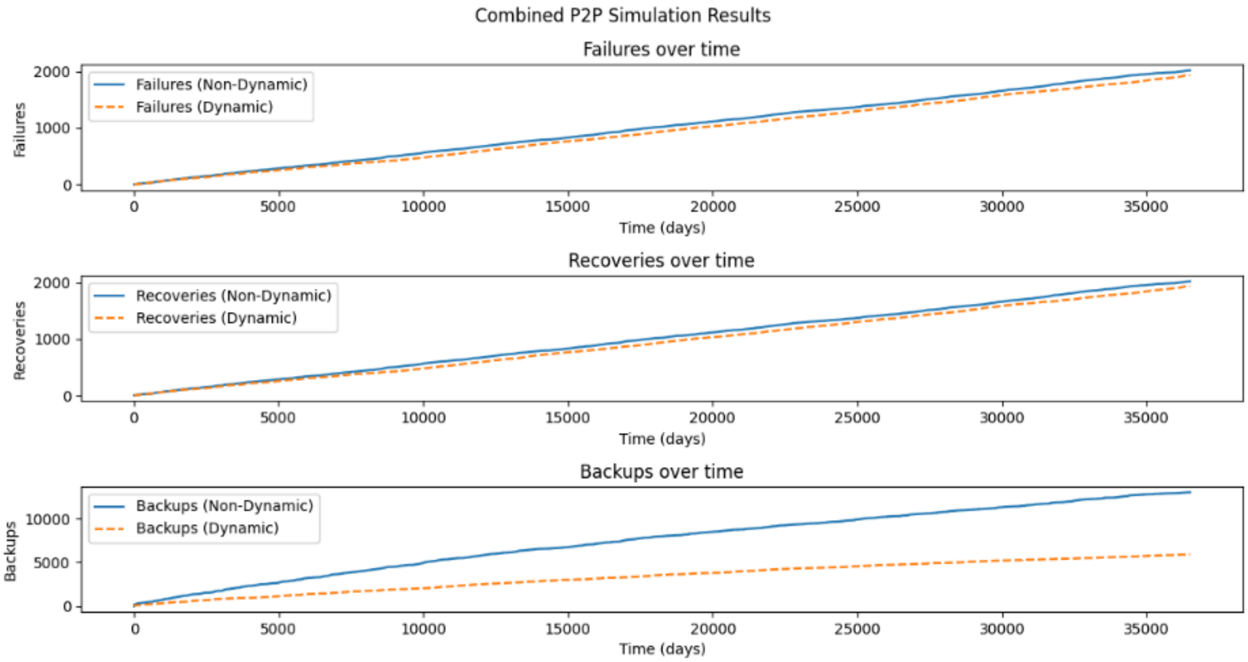
In both sets of plots:

- **Non-Dynamic (blue, solid)**: **No dynamic `n_active`** ("baseline" scenario, always using `n_active = n`)
- **Dynamic (orange, dashed)**: **Dynamic `n_active`** turned on (can increment `n_active` on demand)

We compare the following metrics over simulation time (in days):

- **Failures**
- **Recoveries**
- **Backups**

1. Combined P2P Simulation Results for Small N




```
[peer]
number = 20
n = 10
k = 5
n_active = -1
tolerance = 0
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
```

1.1 Failures Over Time

- **Interpretation**: Enabling dynamic `n_active` does not prevent nodes from *failing*. Failures arise from each node's configured **lifetime distribution** so that frequency is roughly unchanged whether or not the node is using dynamic redundancy.

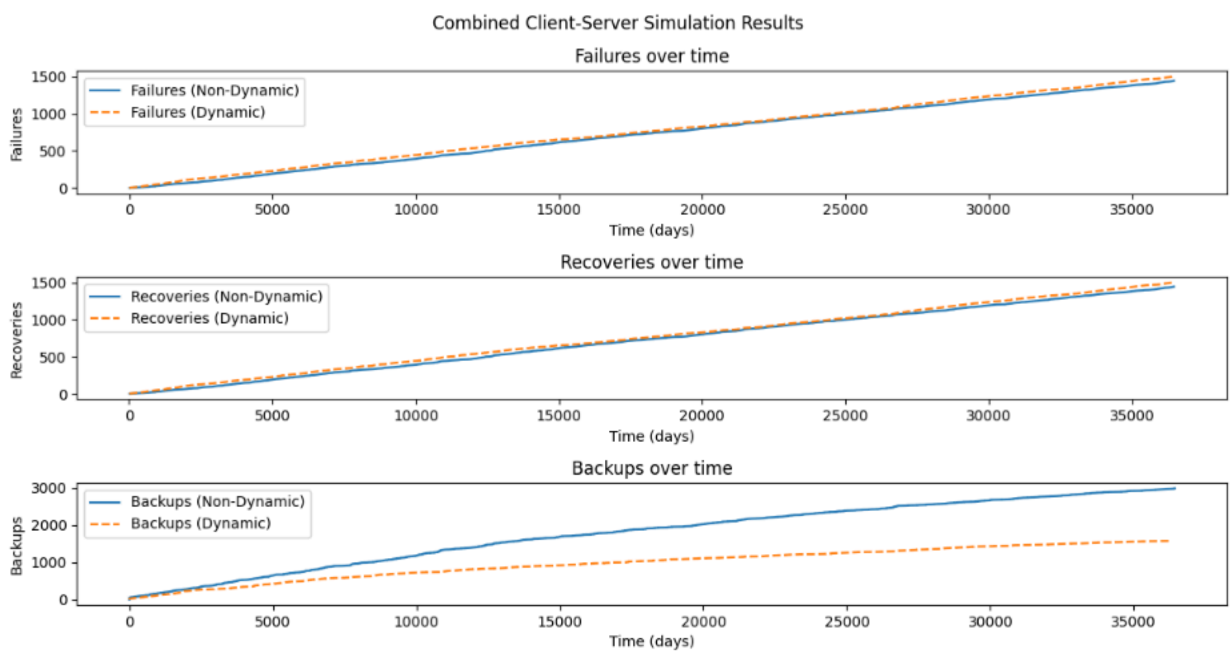
1.2 Recoveries Over Time

- **Interpretation**: Every time a node fails, it (eventually) recovers. Since the average downtime and recovery times are nearly the same in both scenarios, the system sees roughly the same number of recoveries.

1.3 Backups Over Time

- **Observation**: There is a noticeable difference here. Around day 35,000, Non-Dynamic has a slightly higher total count of completed backups than Dynamic.
- **Interpretation**: In the dynamic scenario, a node might start with fewer active blocks and only enable more if needed. This can reduce the volume of overall backups (since not all n blocks are always activated at once). Hence, slightly fewer total backups occur when we only use as many active blocks as necessary.

2. Combined Client-Server Simulation Results for Small N



```
[client]
number = 5
n = 10
k = 5
n_active = -1
tolerance = 0
data_size = 1 GiB
storage_size = 2 GiB      You, 2 min
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
n_active = -1
tolerance = 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
```

As explained previously, the overall backup overhead is reduced.

1. Combined Client-Server Simulation Results for Large N

In the following, **n is relatively large** (n = 100) and we will inspect if, in large amounts of n, we have a difference in both methods.


```
[client]
number = 5
n = 100
k = 5
n_active = -1
tolerance = 0
data_size = 1 GiB
storage_size = 20 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
n_active = -1
tolerance = 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
```

For this part, I chose n = 100. Also, we needed to increase storage_size accordingly.

1.1 Failures Over Time

- As explained previously, the results are the same.

1.2 Recoveries Over Time

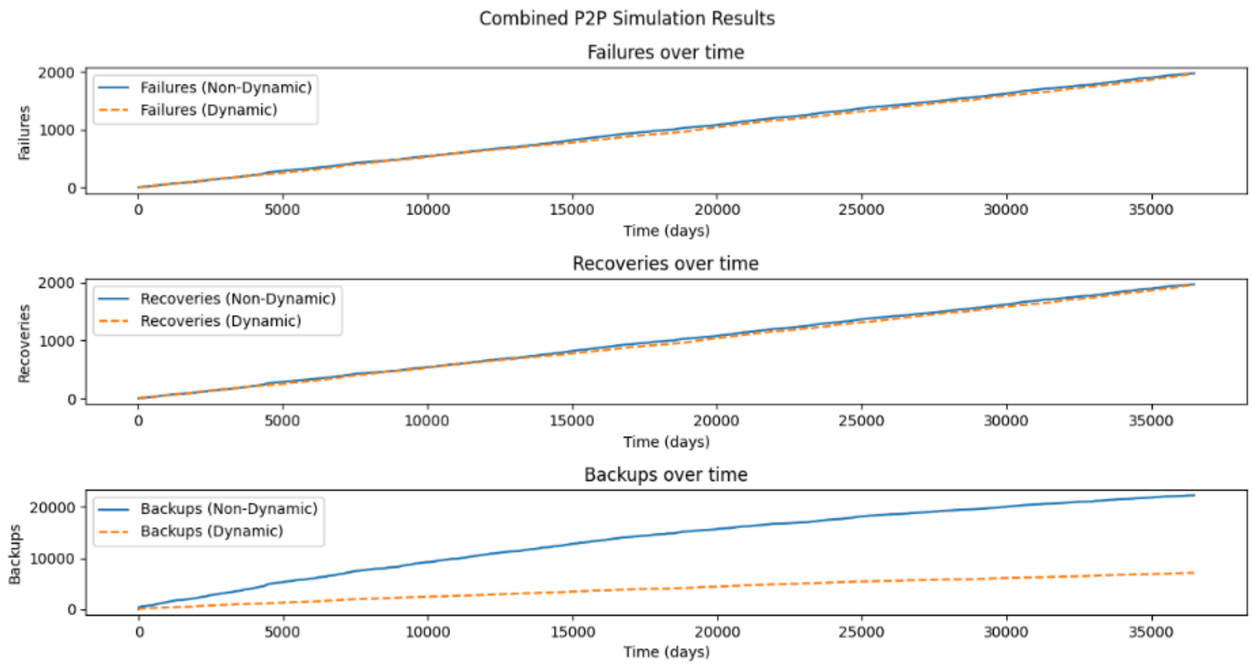
- As explained previously, the results are the same.

1.3 Backups Over Time

- **What the plot shows**
    - There is a **bigger** gap here.
- **Interpretation**
    - With **n=100**, this difference in “active blocks” leads to substantially **fewer total backups** in the dynamic scenario.

2. Combined P2P Simulation Results for Large N

For this part, I chose n = 100. Also, we need to increase storage_size accordingly.




```
[peer]
number = 20
n = 100
k = 5
n_active = -1
tolerance = 0
data_size = 1 GiB
storage_size = 100 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
```

As explained previously, the overall backup overhead is reduced.

Conclusion

1. **Failures & Recoveries**
    - Nearly identical patterns whether dynamic or not—these are driven primarily by node lifetimes and not by redundancy strategy.
2. **Backups**
    - **Biggest difference**: dynamic n **greatly** reduces the total backup count, saving bandwidth/storage overhead.
    - The gap becomes even more pronounced with large n (100) because “always storing 100 blocks” vs. “storing fewer blocks (and adding more only if necessary)” yields a big difference in how many backup transfers occur.

Bottom Line:

- With **n=100**, dynamic n **does** meaningfully reduce backup overhead in both Client-Server and P2P simulations.
- The final **failure** and **recovery** counts, however, remain similar.
- When some remote servers have failed and we had some data on them and now also we are in danger, if we are lucky and make it in time, we can make new redundant blocks because of the check_redundancy function.