

P2P Report

2nd DC Assignment
















M.Sc. in Computer Science

Enrico Pezzano

4825087



Index

Introduction 	2
Peer-to-Peer Backup simulation definition   	2
Erasure coding 	2
Analysis and definition of various parameters 	3
Peer-to-Peer  Client-Server	4
Graphs visualization  	4
Evil nodes extension 	6
But what about the evil corruption? 	12
Conclusions  <small>END</small>	21
Peer Review changes 	22
Appendix: or how I learned to stop worrying and run the code  	23

Introduction

In the following document, I will evaluate the results obtained with respect to the choices made during the second assignment of the Distributed Computing course. In particular, with help of some starter python code, I developed a simulation to analyze the reliability and resilience of a distributed storage system (peer-to-peer or client-server). Additionally, I will discuss about my evil extension and the peer-review changes.

Peer-to-Peer Backup simulation definition

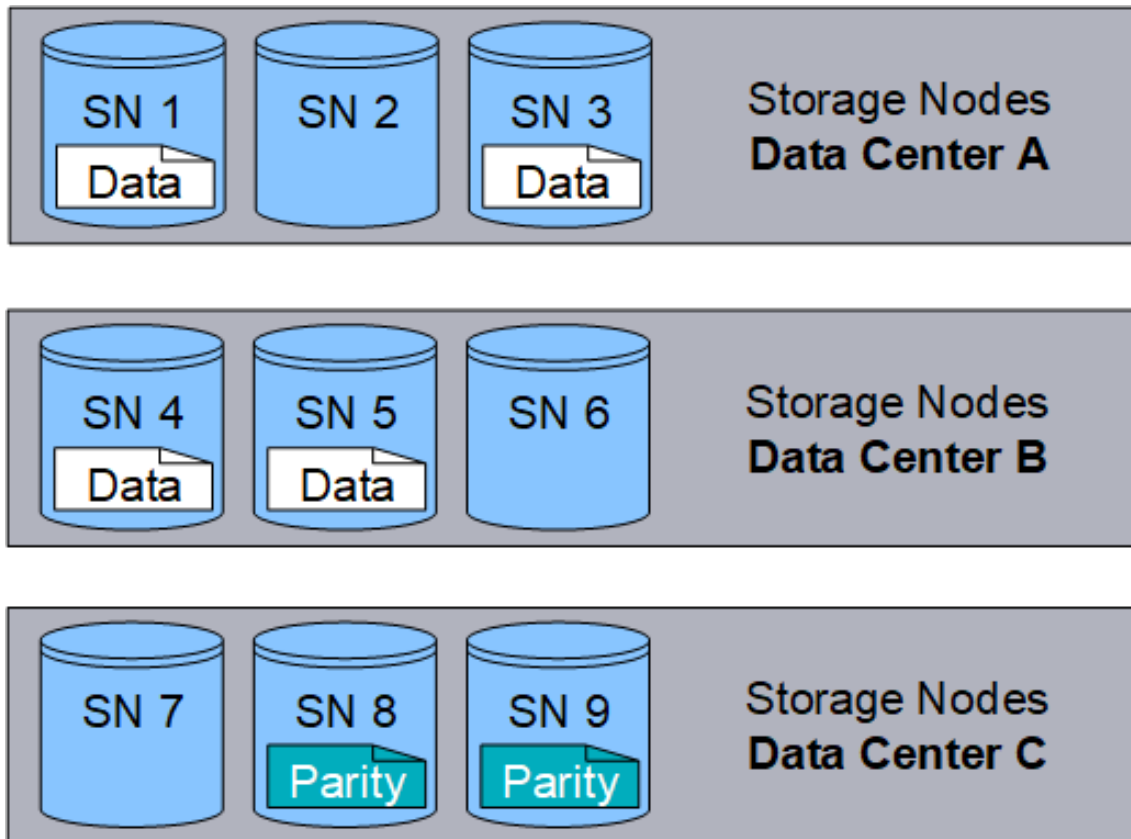
Peer-to-peer backup simulations are essential for modeling distributed data storage systems, as they simulate node activities such as storing, transferring, and recovering data. By evaluating behaviors like resilience and efficiency under various conditions—such as network disruptions, node failures, or malicious actions—we gain insights into system performance. Moreover, adjusting parameters like data sizes, network speeds, and failure rates allows the simulation to highlight strengths and weaknesses in the architecture. This enables real-time optimization to improve the system's robustness, in order to better understand nodes communication in distributed systems, i.e. the purpose of this assignment.

Erasure coding

For the following analysis, let's remember that Erasure Coding is a technique used to protect data during storage by introducing redundancy.

1. Data is split into N blocks (each of size $1/K$ with $K = N - M$) and stored across multiple machines, ensuring recovery even in case of device failure
2. At least K blocks are required to reconstruct the full data set

3. Here, in the simulation, K equals $N - M$, meaning that as long as at least K blocks remain intact, full data recovery is possible.



Analysis and definition of various parameters

Now we can see through the definition of all the parameters:

- **n**: number of blocks in which the data is encoded
- **k**: number of blocks required to recover the data of the node

completely

- **data_size**: bytes of data to back up
- **storage_size**: bytes allocated for storing remote data
- **upload_speed** & **download_speed**: data transfer in bytes per second
- **average_uptime** & **downtime**: mean of time spent online and offline
- **average_lifetime**: time before a crash or a data loss occur

- **arrival_time**: when the node comes online
- **average_recover_time**: mean recovery time after data loss
- * **config**: parameter that stores the path of the configuration file
- * **max-t**: amount of years which limits the simulation time
- * **seed**: number with which compute the randomness of the simulation
- * **verbose**: flag that, if present, makes the script print all the simulation's log to stdout
- * **corruption_delay**: Average delay before an evil node corrupts data, used to schedule data corruption events in the simulation
- * **e** : enables my **e**xtension for this assignment (number of **e**vil nodes).

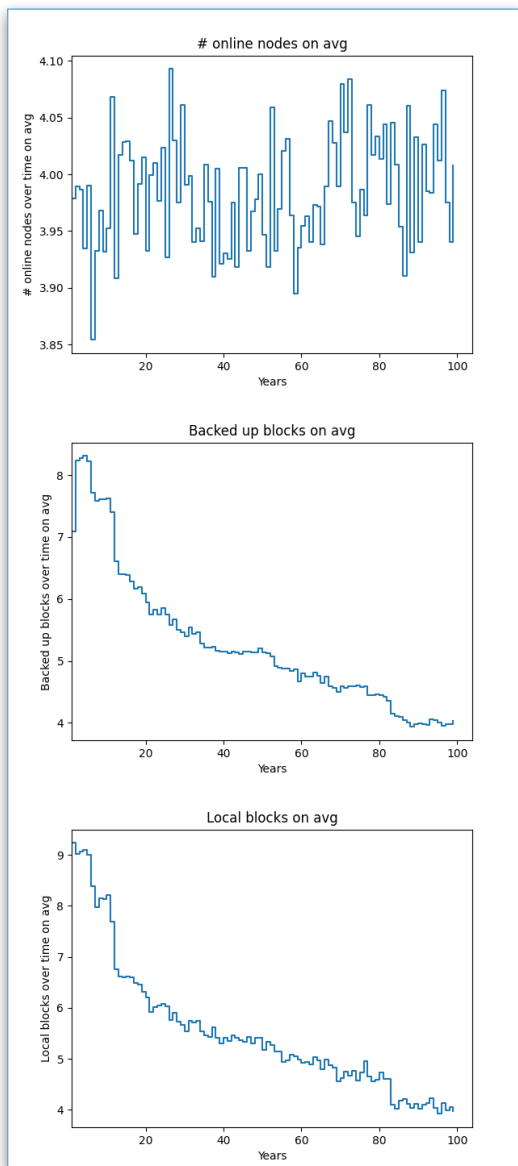
Peer-to-Peer Client-Server

The two configuration in the provided files are of two types: the first is a client-server one, where the server acts as the main hub, handling all service requests from clients. Thanks to the this setup being naturally centralized, it ensures reliable management, but the server may become a single point of failure. Alternatively, the second type of setup is a peer-to-peer system, where nodes communicate directly without a central server, leading to more variable behavior. The best configuration doesn't exist, it depends on our system's goals: peer-to-peer may offer better decentralization and fault tolerance, but it introduces challenges in maintaining consistent data availability. More generally, an optimal system balances redundancy, fault tolerance, and stability. As we will the client-server configuration will show greater resilience to evil nodes compared to the peer-to-peer configuration.

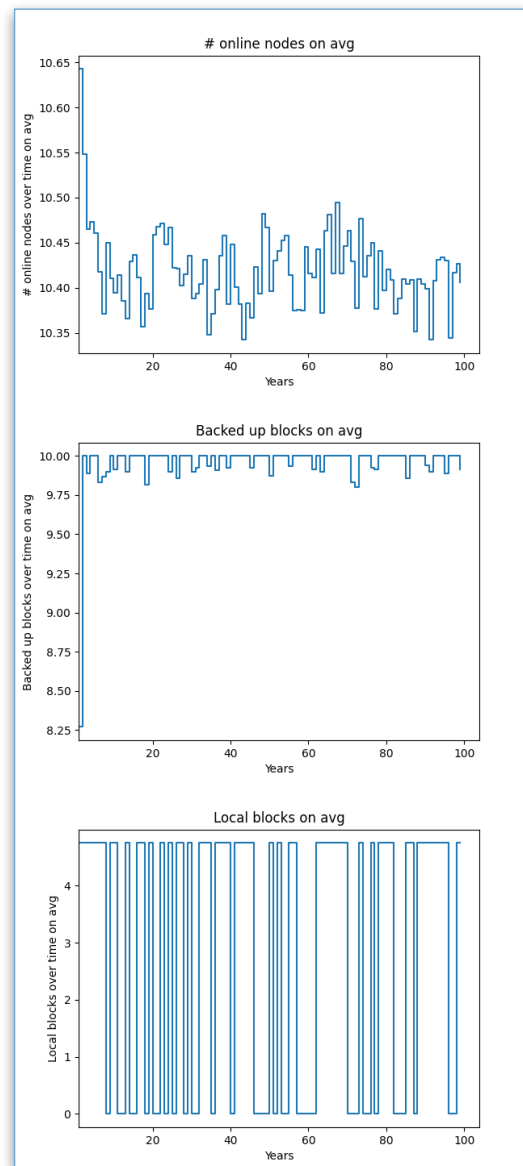
Graphs visualization

Finally, let's analyze and discuss the resultant graphs and plots, which I

P2P



Client-Server



created using a second python script (**my_plot.py**) that used the metrics gathered during the execution of the main script (**storage.py**).

The left plots indicate an average of around 4 peers online per year. Over time, local blocks decrease as peers fail without proper backups. Similarly, backed-up blocks on other peers also drop since those peers eventually fail, losing both their local and remote blocks. This decline in the metric for backed-up blocks reflects the **cumulative** effect of these peer failures.

On the right, we can see the client-server plots: the top one has a high average of nodes online, stabilizing around ten. This is due to the servers' **longer lifetimes** (about 2 years), allowing for **faster recovery**. The client primarily influences local block counts, as servers don't store local blocks, so any fluctuations indicate client failure and recovery. As for backed-up blocks, most client data is backed up on servers within a year, with some drops when servers fail, though these are restored over time (backed up in years).

For the sake of simplicity, I left out server nodes of the storage plots, in order to prevent the distorting of the averages (since they don't hold local blocks). In that way, it is easier to interpret the client's block status over time without unnecessary noise.

Evil nodes extension 🐱

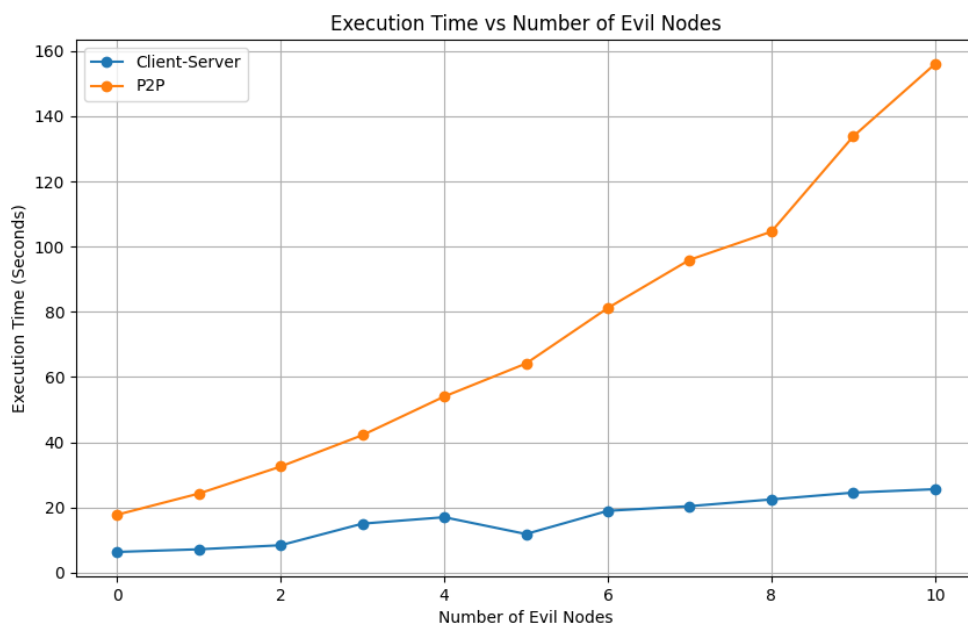
Since this assignment also required to add an extension, I decided to recycle the “e” of “**e**xtension” and develop some **evil** nodes. In particular, some peers intentionally play tricks.

In particular, evil nodes, in this python simulation, maliciously tinkered the honest nodes with behaviors such as flooding honest nodes with fake blocks (marked with a block id equal to 99; because -1 led to negative blocks avg). This flooding takes place when an evil node goes online, and then it schedules uploads of fake blocks for every nodes in the simulation that is not itself, deliberately filling their storage with invalid data. Additionally, evil nodes periodically corrupt existing data blocks (both local and remote ones) by marking them as, indeed, corrupted after a delay defined by the '*corruption delay*' parameter and in both the client-server and peer-to-peer configuration files, in order to disrupt the integrity of the system.

As we will see, this behavior creates an enormous challenge for data recovery, also because as evil nodes increases linearly, their damage is not, and the damage percentage grows quickly. This will be true especially for the peer-to-peer simulations, where decentralization makes much harder to isolate, and also mitigate the evil nodes influences.

In the end, I was able to track and plot various datas and graphs thanks to the implementation of metrics variables. Through them I managed to evaluate percentages of fake or corrupted data blocks, and the overall data loss versus the resiliency of each of the two configurations to the presence of evil nodes.

Evil results and Evil plots 🖤



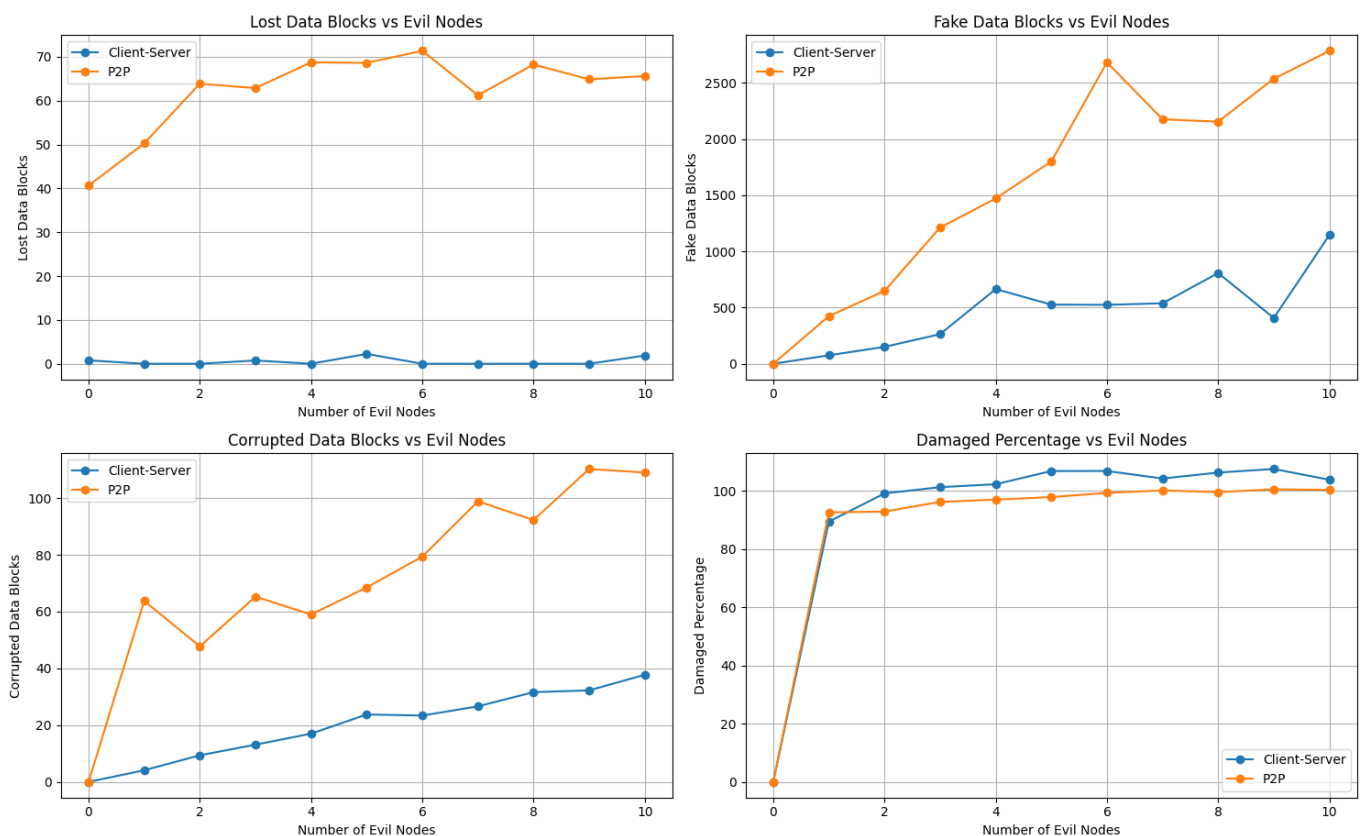
Thanks to some additionally python scripts, I plotted multiple graphs. But the immediately noticeable metric was the execution that went up especially for the peer-to-peer simulation, while for the client-server did not have a high increase. In the following I will compare the execution

time of both variants between 0 and 10 evil nodes, as we can see in the figure above:

- client-server -> from circa **5** seconds to around **20**-ish seconds
- peer-to-peer -> from circa **20** seconds to around **160** seconds.

In the next figure, I wanted to different simulation final metrics between peer-to-peer and client-server, and, as we can imagine, the latter is the more resilient one.

Simulation Results: Client-Server vs P2P

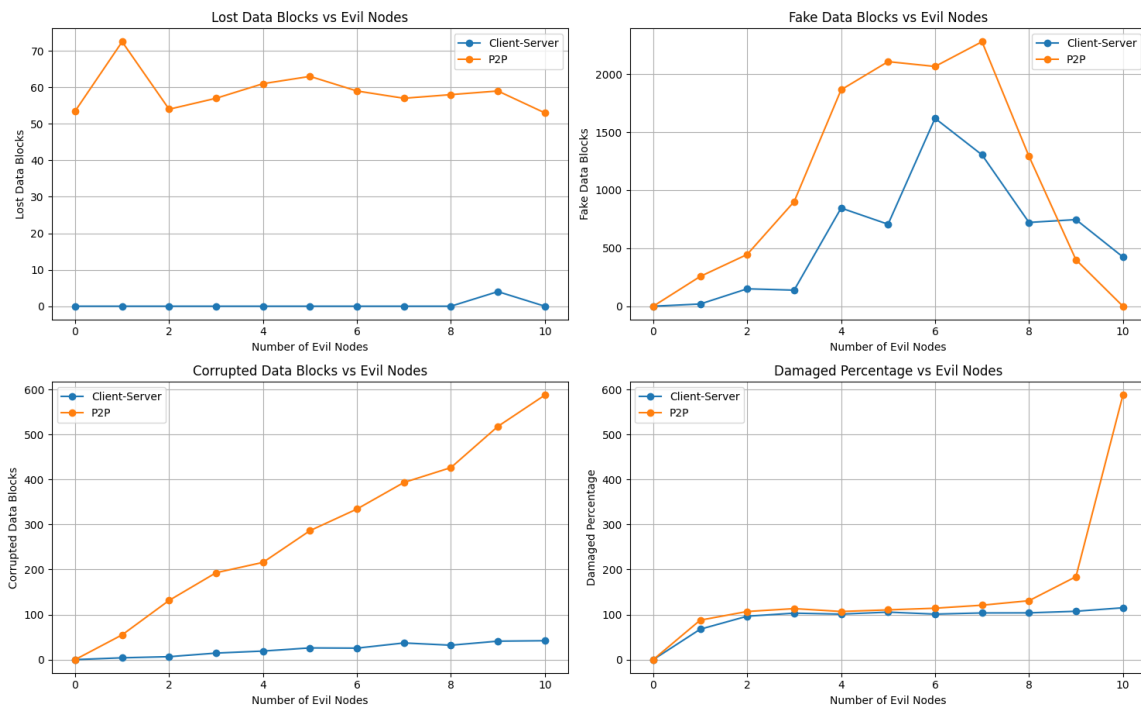


We can immediately see the obvious winner in the firsts three plots:

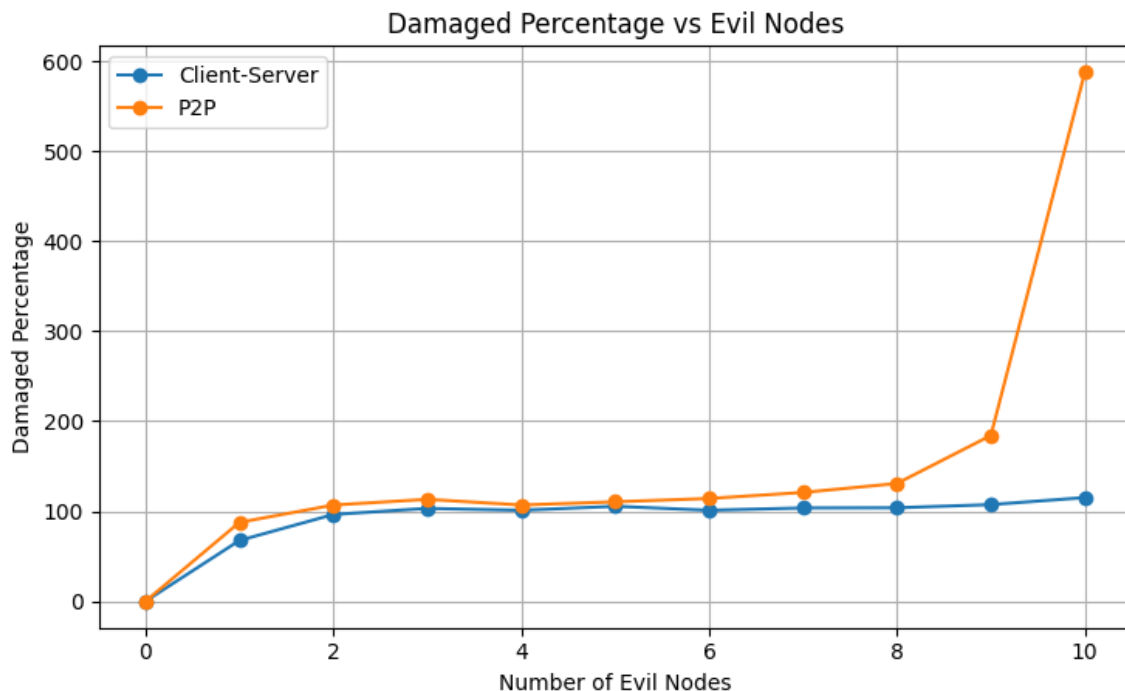
1. the top-left plot shows the peer-to-peer configuration at 0 evil nodes, has already lost on average 40 data blocks, while its "adversary" comes out clean. Sequentially, the first configuration loses 10 more data blocks for each evil node, for then stabilizing around 70 lost blocks for every other number of evil nodes; the client-server

configuration, instead, keeps the graph of lost blocks more or less flat, regardless of the number of evil nodes

2. the top-right graph shows instead how the number of fake data blocks grows to the growth of evil nodes. In this case the client-server configuration is not "immune" as before, but it is clearly more robust; in the worst case (10 evil nodes), it stores on average a little more than a thousand fake blocks, versus the peer-to-peer corresponding that, with the same amount of evil nodes, stores on average 2700 fake blocks
- the number of fake blocks is so high because of the "flooding attack" that I wanted to simulate. In particular this behavior is initialized when evil nodes go online (schedule also fake uploads) , and for a fake block is scheduled for every peer that is not itself (evil nodes flood also other evil nodes, but themselves) leading to an exponential accumulation of fake data.
 - i also implemented a version where evil nodes only upload fake data blocks to honest nodes, checking if '*peer.evil is not True*', leading to 1500 fake block in the worst case for the client-server configuration and 2200 for the peer-to-peer one (as shown in the figure below), and also a peer-to-peer execution time of **48** seconds in the worst case. As we can see in the figure below, also in this case the clear winner is the client-server setup. But ultimately I wanted to keep my simulated evil nodes *bad to the bone*; so I deleted the condition that checked if a peer was evil.



3. the bottom-left plot shows how the data blocks become corrupted with respect to the growth of evil nodes. Also in this case, we can clearly see that the client-server choice is the right one if we want to keep as much clean (uncorrupted) data as possible.
4. finally, the bottom-right plot shows the percentage of damaged data blocks (fake or corrupted data blocks), that shows that no data is ultimately safe in the simulation against my implementation of evil nodes, 2 of them are sufficient for damaging 100% (if not more) of the data blocks
 - note that, this high percentage number is obviously due to the flooding of fake data uploads and because I chose to count damaged data as the sum of fake and corrupted data; hence why the percentages higher the 100%. If we add the '*peer.evil is not True*', we can see that the percentages are more or less all around 100, but having a worst case scenario with the peer-to-peer simulation that truly suffers only under 10 evil nodes (all of the nodes), as we can see in the graph below, zoomed-in from the alternative plot of the 2nd point.



When it comes to analyze the simulation as a whole, in the 100 years simulated, we can see via the resultant plots that as the number of evil peers grows, it grows also the number of problems of the **P2P** system, as we can see in the two above figures (side-by-side), for instance:

- The number of online nodes (on average) grows unstable
- The number of backed up blocks over time (on average) is smaller as evil nodes increase, and there are often collapses
 - The number of local blocks over time (on average) behaves similarly to the above.

Instead, in the 100 years of the **Client-Server** simulation we can immediately see that this second type of system is more resilient to my implementation of **evil** nodes, but it did not executed flawlessly, indeed, the resultant problems were practically the same, but with 3 fundamental differences in the client-server setup:

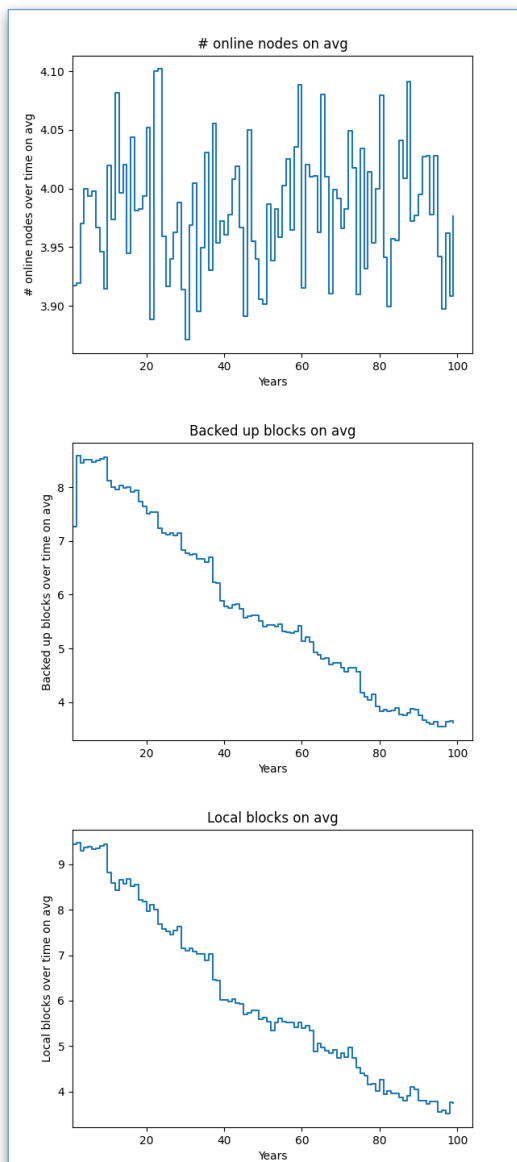
1. the number of online nodes on average was 10 instead of 4 (ignoring the .6 nodes initial peek)
2. the number of **backed up** blocks remained around the same (10), for the whole simulation, without decreasing

3. the **local** blocks was lower (4) and very fluctuating (going also to 0), but in the end they were always able to recover themselves.

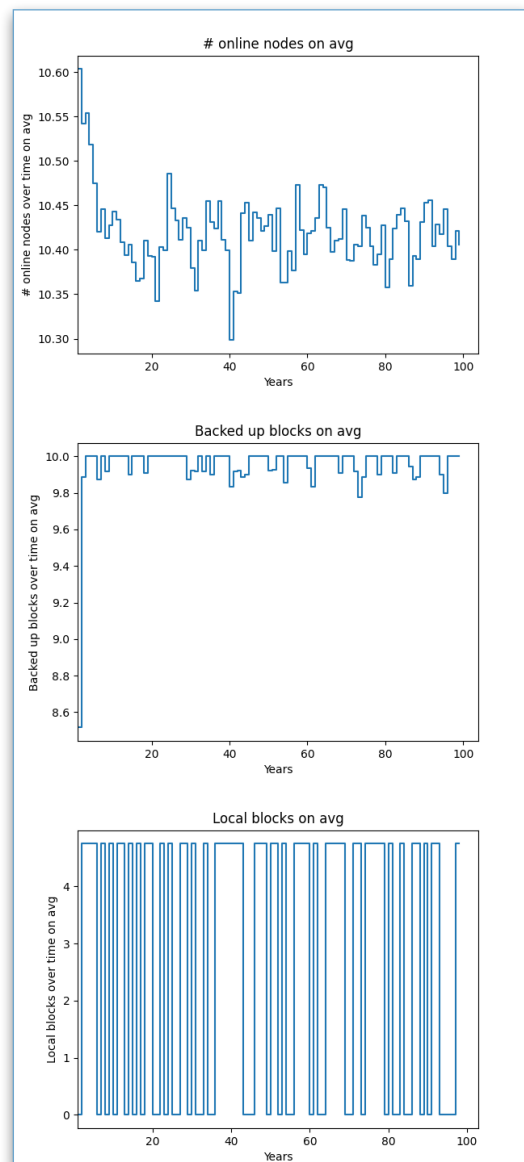
But what about the evil corruption? 🤩

The evil corruption implementation that I implemented in *storage.py* simulates another malicious behavior: evil nodes intentionally corrupt data blocks by marking them as corrupted after a constant '*corruption_delay*' defined in the config file of both setup, then passed through the simulation by initializing the homonym python variable with

P2P with 0 evil peers



Client-Server with 0 evil peers

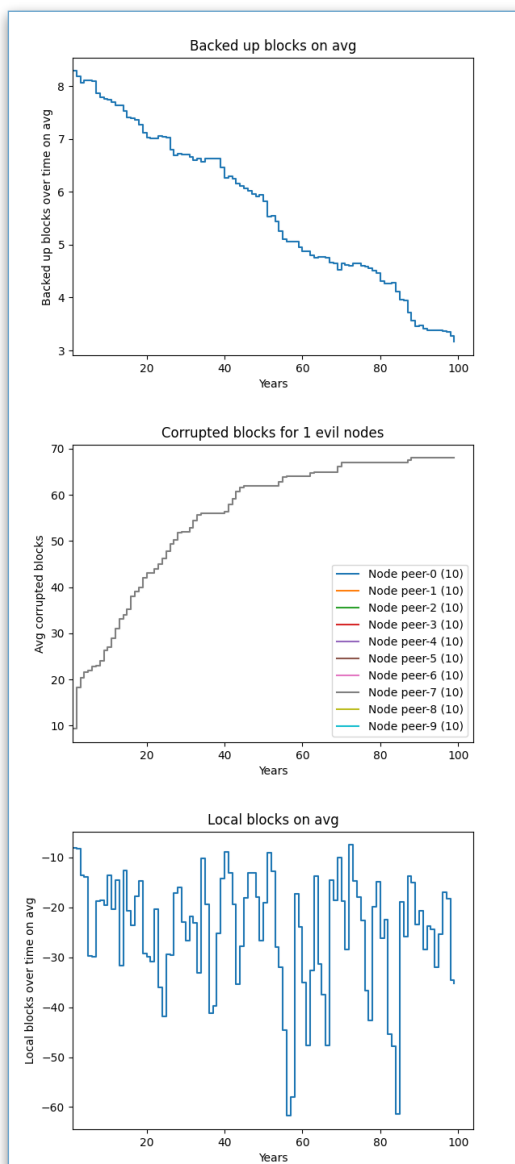


its defined value. In particular, I implemented a *DataCorrupted* event, scheduled only for evil nodes, based on the *corruption_delay* variable described before. When this event is processed, a random block (either locally or remotely held) is selected and marked as corrupted. The corrupted state represents hidden damage that remains undetected until the block is accessed during upload, download, or recovery events, kind of like in a real-world scenario.

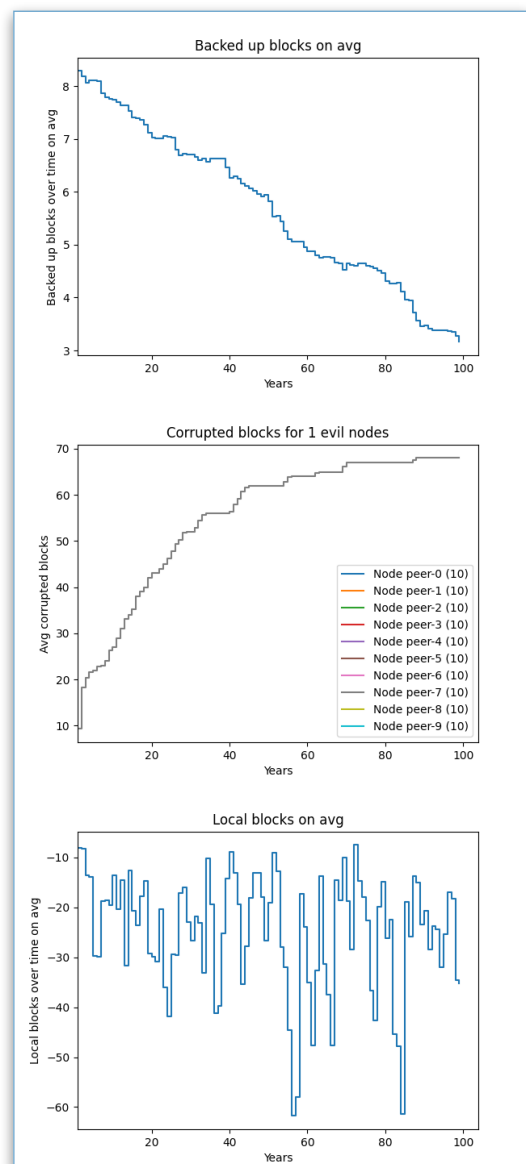
After that:

- if a block is local, the corruption is tracked by adding its block ID to the node's '*corrupted_blocks*' set

Zoomed-in P2P results (1 evil)



Zoomed-in C-S results (1 evil)



- if a node is remotely held, I decided to record the corruption as tuples of peer node and corresponding block ID.

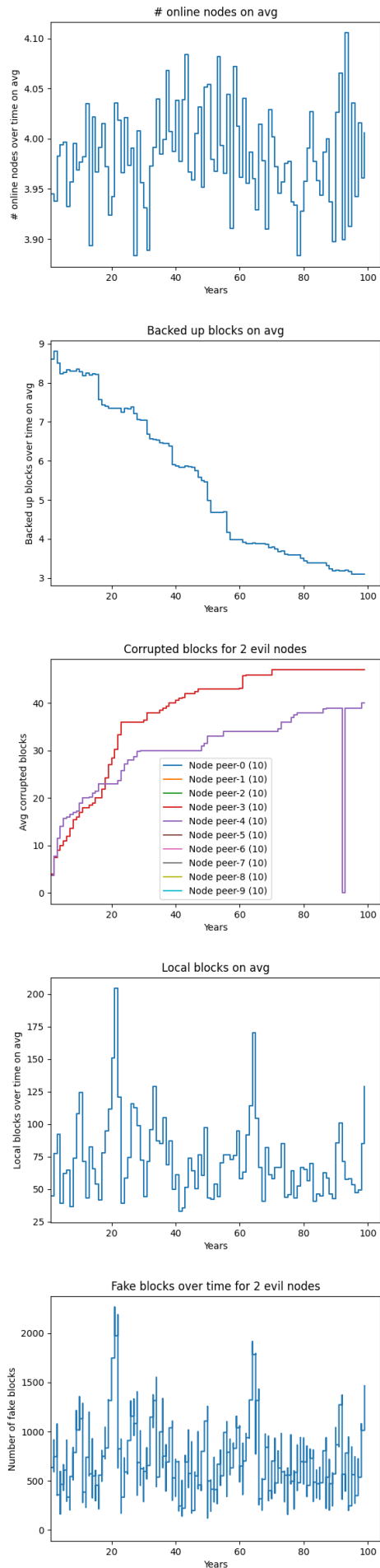
Over time, the overall corruption accumulates, as we can see in the above figures. Additionally, I implemented the computation of the percentage of corrupted blocks, relative to the total number of blocks in the system (as we already saw in the section before).

Initially, we note that for only 1 evil node, the corruption graphs are similar, regardless of the chosen configuration. But, as soon as evil nodes start to grow, we can immediately observe (in the following figures) an overall worsening of the simulation as a whole.

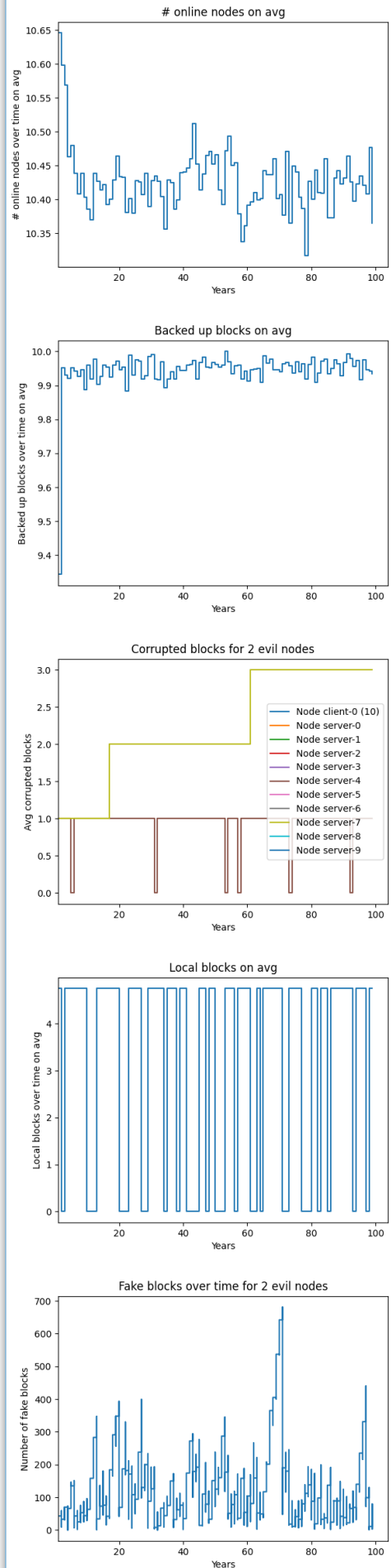
The corruption graphs are the ones at the center of every group of plots, one for each configuration (peer-to-peer or client-server; note that servers do not talk to each other).

Note that, for the sake of brevity for this report, I only included some of the total output plots; even though the total of them are included in the archive of this assignment.

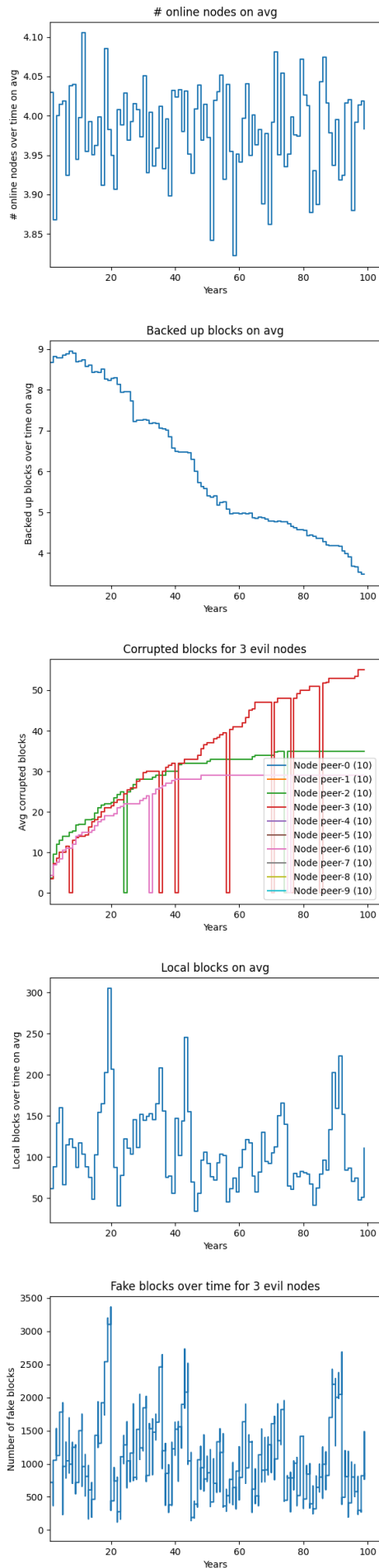
P2P results (2 evils)



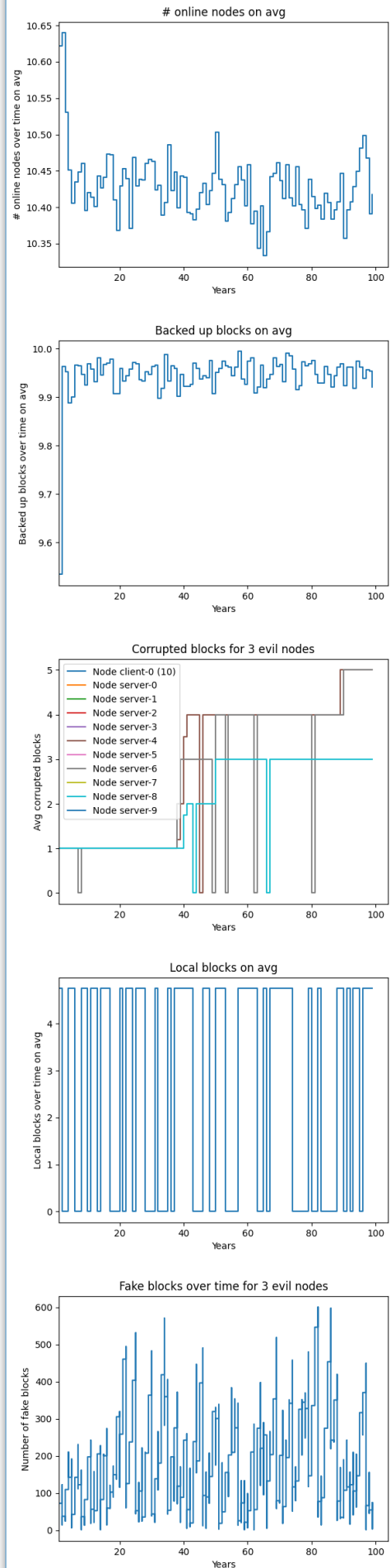
Client-Server results (2 evils)



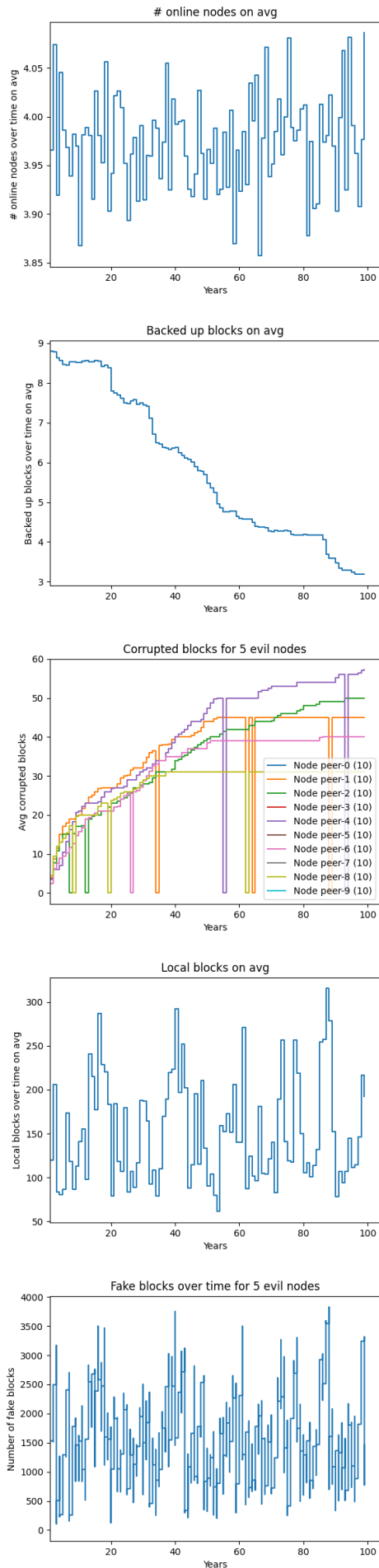
P2P results (3 evils)



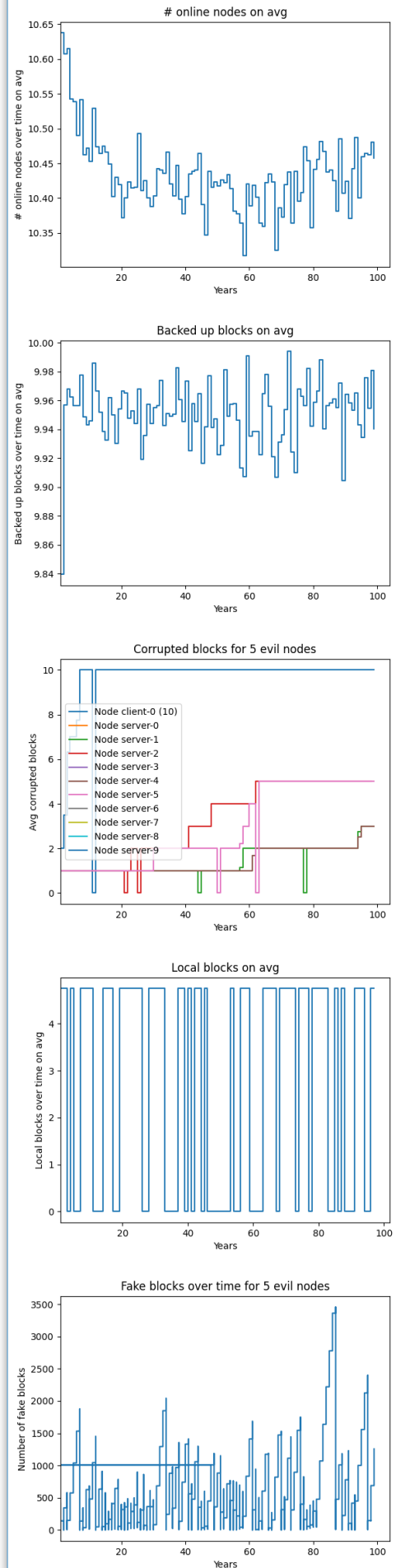
Client-Server results (3 evils)



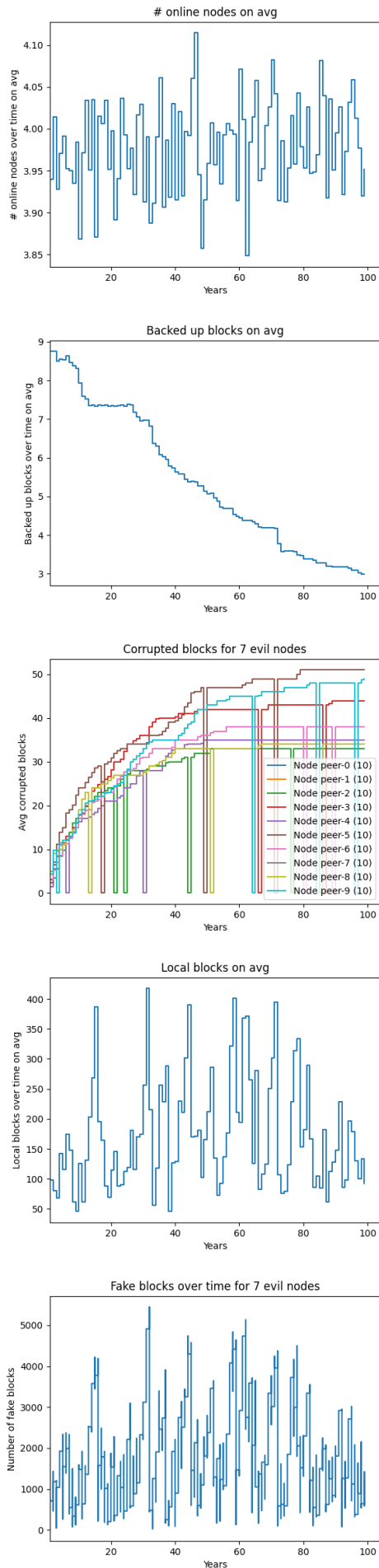
P2P results (5 evils)



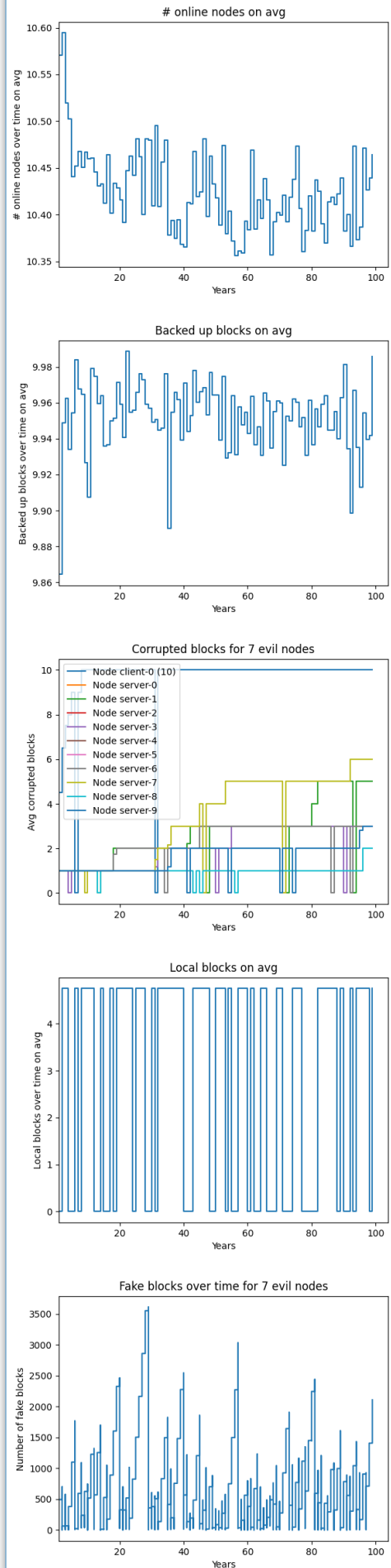
Client-Server results (5 evils)



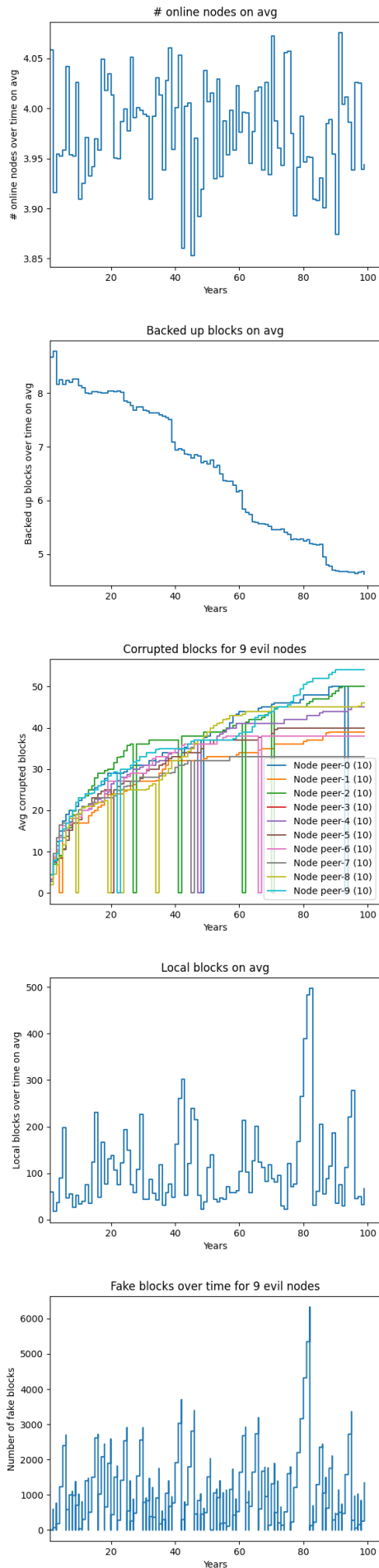
P2P results (7 evils)



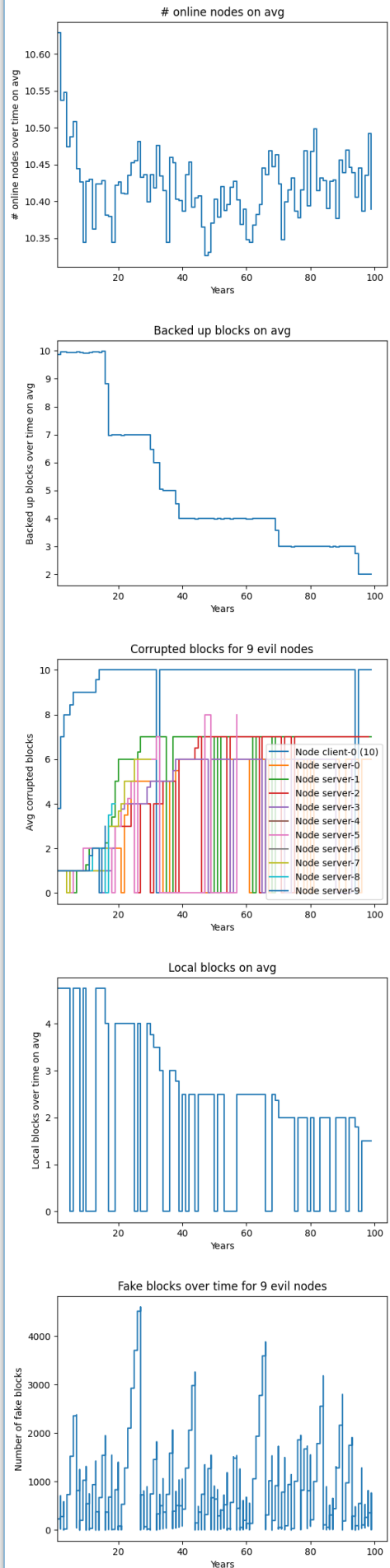
Client-Server results (7 evils)



P2P results (9 evils)



Client-Server results (9 evils)



Analyzing the corruption graphs alongside the other plots, we can immediately see that the average corruption of blocks grows to the growth of evil nodes, and, of course, the simulation is more and more damaged the more evil nodes there are. In particular, we can see that (for both configurations) the number of average online nodes remains the same, this is due to the fact that my extensions attack data blocks and try to flood the network, but do not touch nodes inherently, they only get "*fatigued*". Secondly, we can see that the client-server configuration is much more resilient in keeping the remotely backed up blocks, but "breaks" when the high majority of the nodes are evil (9, as observable in the figure above); instead the peer-to-peer configuration keeps the same disadvantages of the execution without "evilness", but gradually loses remotely backed up blocks quicker (the relative graph seems to grow steeper as the number of evil nodes grows). Lastly, we can clearly see with the naked eye the difficulty of the system in storing local blocks against my evil extensions. In detail, the client-server setup is again the more resilient one, storing on average always around 4 blocks (likely because server do not talk to each other in this simulation, and how *BlockRestoreComplete.update_block_state* counts true or fake blocks in the client-server case), but still "breaks" when the high majority of the nodes are evil (9, seen before); the peer-to-peer configuration, instead, suffers especially against fake data blocks, likely because peers store their own data and also for other peers (being caught in the flooding attack), and we can see that the peers are victim of the flooding attack because they are storing orders of magnitude more data blocks than the client-server corresponding setup.

Conclusions END

In the end, after my analysis of the simulation, with and without the evil extensions, it was clear that the introduction of **evil** nodes highly influenced and "fatigued" the system. The assignment allowed me to also analyze in detail the simulator, also tinkering between the configuration files and changing their values, or just by observing the cases without evil nodes. In particular, I ran the simulation multiple times, also with the aid of some shell script (compressed together with this report) and gathered data about each execution inside 2 CSV files (also available in the compressed archive).

Finally, when it comes to decide which configuration to choose, it is clear that both peer-to-peer and client-server configurations have their strengths and weaknesses. In detail, the client-server configuration demonstrates greater resilience against malicious actors, as the centralized nature of the server allows for stricter control and better isolation of evil nodes. This makes it a more suitable choice for scenarios where reliability and data integrity are critical. On the other hand, the peer-to-peer configuration offers decentralization and fault tolerance, which can be advantageous in environments where scalability and resistance to single points of failure are prioritized. However, its vulnerability to flooding attacks and data corruption by evil nodes highlights the need for robust mechanisms to detect and mitigate such threats.

But ultimately, the choice depends on the needs and the situation of the user.

Peer Review changes 🌴

- Deleted leftover comments
- Added the commands executed to obtain the various figures, with its relative parameters
- Deleted XOR texts (not part of the laboratory)
- Briefly explanation of evil peers
- Plots interpretation
- Plot images zoomed bigger and places vertically (one column for each configuration)
- 0 vs some evil peers situation
- Added stdout prints that highlight which of the nodes (out of the tens) are evil
- Added testing script with recursion
- Added explanation of lost data count growing to the evil nodes number.
 - Added flooding attack
 - Added fake blocks upload
 - Added corruption of blocks event
 - Added data gathering with CSVs, shell scripts and external python scripts
 - Added corruption graphs
 - Added execution times plots

Appendix: or how I learned to stop worrying and run the code ❤️💣

As explained during lectures, the files “discrete_event_sim.py” and “workloads.py” are necessary to the program, otherwise the code will not be executed. Of course, **storage.py** can be executed directly from the command line using the python3 command and interpreter; e.g.



```
Peer-to-Peer-Backup — zsh — zsh (qterm) • zsh — 108x9
Desktop/dc/Peer-to-Peer-Backup on ? main [x?] via • v3.12.3
[> python3 storage.py p2p --e 0
parameters parsed :)
Running simulation...
Simulation completed
Lost data blocks: 0

Desktop/dc/Peer-to-Peer-Backup on ? main [x?] via • v3.12.3
>
```

The following text shows my configuration files for both simulations.

[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

corruption_delay = 100 days

[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
corruption_delay = 100 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
corruption_delay = 100 days