Università di Genova
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

Report - Peer to Peer Backup App

Distributed Computing
Master course in Computer Science

Authors:
Cattaneo Kevin - 4944382
Foschi Lorenzo - 4989646

Index

I. Definition of Peer-to-Peer Backup App

This code simulates the operations of a distributed backup system, with a focus on nodes' activities related to data storage, transfer, and recovery. By representing events like node connectivity, failures, and data management, it allows for the evaluation of the system's resilience and efficiency under different scenarios. With the ability to adjust parameters such as data sizes, network speeds, and failure rates, the simulation facilitates an in-depth analysis of the backup system's performance and behavior, aiming to uncover potential weaknesses and strengths within the system's architecture and functionality.

II. Erasure Coding

Before diving into the analysis, let’s remember how Erasure Coding parameters works:

1) We encode data in N blocks, each of size 1/K with K = N - M
2) K will be the number of blocks needed to restore the data: remember that a function of degree K - 1 has K coefficient, and this is the base of Erasure Coding
3) In our simulation we want to recover the data if and only if at least K blocks are still recoverable. In that case we’ll be able to recover them all.

III. Analysis

To analyze the performance of the peer-to-peer backup application we observe the definition and behavior of various parameters:

- n: The number of blocks in which the data is encoded
- k: The number of blocks sufficient to recover the whole node's data
- data_size: The amount of data to back up (in bytes)
- storage_size: The storage space devoted to storing remote data (in bytes)
- upload_speed: The node's upload speed, in bytes per second
- download_speed: The download speed
- average_uptime: The average time spent online
- average_downtime: The average time spent offline
- average_recover_time: The average time after a data loss
- average_lifetime: The average time before a crash and data loss
- arrival_time: The time at which the node will come online
- extension basic: boolean parameter to enable (1) the corruption extension
- extension advanced: involve the recovery extension

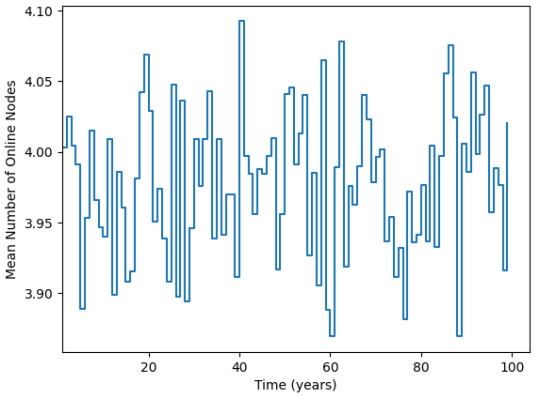
IV. Peer-to-Peer VS Client-Server scenario

In a client-server configuration, the server typically plays a central role in providing services or resources to clients. Clients connect to the server, and the server is responsible for managing their requests.
On the other hand, in a peer-to-peer configuration, nodes interact directly with each other without a central server. This decentralized structure can lead to more variability in node states.
Which scenario can be considered "optimal" depends on the specific goals and requirements of the system. A more balanced distribution among online and disconnected nodes in the peer-to-peer configuration may indicate a more resilient and decentralized system, but it could also pose challenges in terms of data availability and consistency, properties that servers may already satisfy, depending on their configurations.
Moreover taking in consideration factors such as fault tolerance, data redundancy, and overall system stability can help us to obtain an optimal system. We will focus our extensions on these concepts.

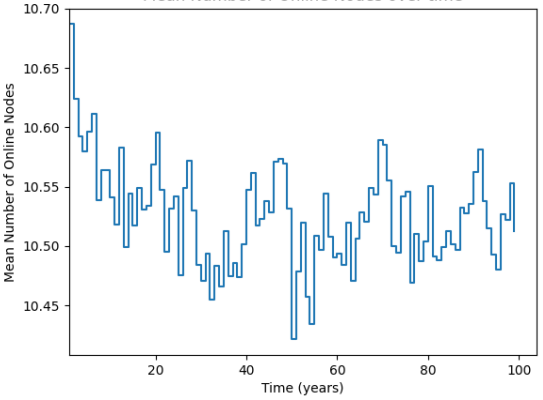
V. Plots

- P2P (*left*, average_lifetime = 1 year)
- Client-Server (*right*, average_lifetime = 2 years)

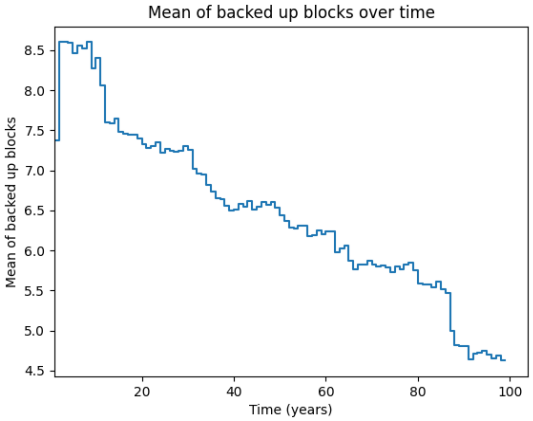
P2P



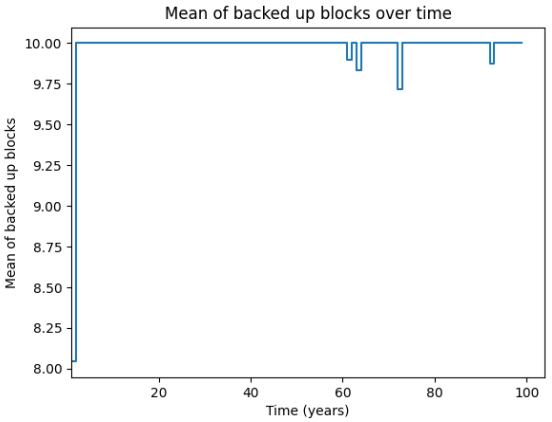
Client-Server



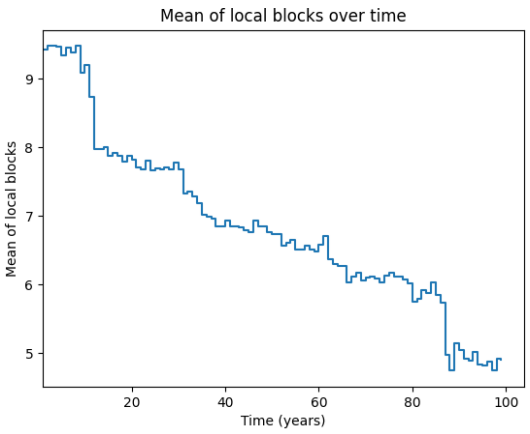
P2P



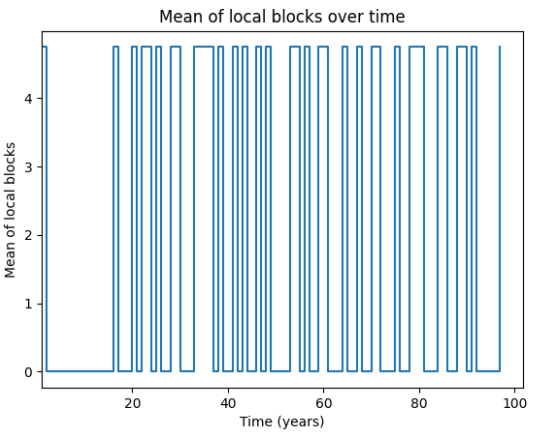
Client-Server



P2P



Client-Server



1. P2P - Plot Analysis

What we can see here is:

- For the first plot, a mean of more or less 4 peers up per year
- The number of local blocks decrease in time since some of them will be lost forever if peers fails after some time and no one has backups for their specific blocks
- The same as stated for local blocks, apply also for backed up ones on other peers, since the other peers fail too and lose all their blocks (both local and remote held for others); so the metric of backed up blocks on the single peer decreases in time.

2. Client-Server - Plot Analysis

What we can see here is:

- For the first plot, we have a high mean of nodes up in the very first years, that is also a lot influenced by our configuration (average_lifetime = 2 years), then the mean is more or less stable on ten nodes “and half”. That depends also on the kind of machines we are observing: since we are tracing a sum of 11 nodes in total, where 10 are servers (more performant than the client machine), we also see a faster recovery respect to a situation with various peers that share the same “hardware” configuration
- The number of local blocks is influenced a lot by the configuration: since servers have no local blocks, the only blocks that are traced are the ones of the client, that depending on the status of client machine (failed or not) could have been lost or not; but here we also see the restore process working: since there is an increase in some years, it means that local blocks of the client, previously backed up, are restored.
- For backed up blocks, we can see how in a year the most of the local blocks of the client are backed up on servers, and in some years there is a little drop, that points out the presence of failing servers that stop holding blocks and lose them; however they are backed up again in the following years.

Please note again, as stated before, that we choose not to include server nodes in the storage plots, since their zeros would affect the mean and would make useless calls to count the number of local or backed up blocks when they have not been changed on the client, the only one who possesses them. That applies for next plots, too.

VI. Extension

Our extension involves the simulation of data corruption (both extension 1 and extension 2) and data recovery (only with extension 2). Our goal was trying to add the real scenario in which some blocks get corrupted throughout the evolution of the system, producing meaningful insights and plots regarding this addition and its impact on the overall simulation performance and behavior.

At first we build the corruption event with DataCorrupted class, automatically scheduled at certain fixed intervals (that the user can specify in the config files): this event will randomly select either local or remotely_held blocks in order to put them in the CORRUPTED state (as soon as these blocks are present, because for example servers only keep blocks for clients).
In our vision CORRUPTED is a special intermediate state in which the block is lost but the keeper isn't actually aware of it. Said this, the next obvious step was building the logic to let nodes discover this corruption. Consequently, there are two different ways for a block to transition from CORRUPTED to LOST, that is the state from which the system will recover the block:

- At the very last time, during the download or the upload, the Node discovers that a corruption happened in the node he wants to backup for himself or in the node that he wants to restore for a peer. At this point the Backup/Restore event isn't scheduled due to the unfortunate discovery: the simulation will be in charge of retrieving this new marked LOST block, if possible.
- *Only with extension 2*: Thanks to the Recovery event with DataRecovered class, nodes are able to witness corruptions in their own nodes, letting the transition in LOST state to happen.
  This other independent event is scheduled at another fixed integrity rate: this allows it to play with different combinations of corruption/integrity frequencies.

  Note that the real restore part is already done by the original code itself. The role of the recovery class is to make aware the nodes that that block isn't more operable (LOST) without a prior restore.

These additions let us draw interesting plots, and also enabled us to infer conclusions about data integrity mechanisms.

```python
class DataCorrupted(NodeEvent):
    """A block of the node (local of held for a remote peer) is corrupted."""

    def process(self, sim: Backup):
        node = self.node

        # Schedule the next corruption event
        sim.schedule(exp_rv(node.corruption_delay), DataCorrupted(node))

        # If the node is offline or failed, we don't corrupt anything in that moment
        if node.failed:
            return

        try:
            # select a random block_id from local and held blocks to corrupt, if any
            block_id = random.choice([i for i in range(sum(node.local_blocks) + len(node.remote_blocks_held))])
        except IndexError:
            # no blocks to corrupt
            return

        if block_id < len(node.local_blocks):
            # if the block_id is in local_blocks, we corrupt it adding it to the set of corrupted blocks
            node.corrupted_blocks.add(block_id)

            sim.log_info(f"Local block {block_id} corrupted on {node}")
        else:
            # if the block_id is in remote_blocks_held, we corrupt it
            idx = block_id - len(node.local_blocks)
            # we add a tuple (Node, int) aka (remote owner of that block, that block id stored in remote_blocks_held) to the set of corrupted blocks
            peer = list(node.remote_blocks_held.keys())[idx] # we get the idx-th peer from the list of remote_blocks_held peers
            peer_block_id = node.remote_blocks_held[peer] # we get the block_id of the block held by the remote owner
            node.corrupted_blocks.add((peer, peer_block_id))
            sim.log_info(f"Block {peer_block_id} corrupted on {node} (remote block held for {peer})")

        years = 1
        if str(format_timespan(sim.t)).split(' ')[1] == 'years,':
            years = int(str(format_timespan(sim.t)).split(' ')[0])
        node.corrupted_blocks_over_time.append((years, len(node.corrupted_blocks)))
```

```python
class DataRecovered(NodeEvent):

    def process(self, sim: Backup):
        node = self.node

        # Schedule the next corruption event
        sim.schedule(exp_rv(node.integrity_delay), DataRecovered(node))

        # If the node is offline or failed, we don't recover anything
        if node.failed:
            return

        # No need for checks if len blocks is zero (even server case) since the for loop will manage this case

        # Check if there are corrupted blocks among the corrupted_blocks
        # cycle through the list of corrupted blocks
        for block in set(node.corrupted_blocks):
            # if the block is a tuple (Node, int) aka (remote owner of that block, that block id stored in remote_blocks_held)
            if isinstance(block, tuple):
                # Remove the tuple from corrupted_block and invalidate the block
                owner, block_id = block
                node.remote_blocks_held.pop(owner, None)
                owner.backed_up_blocks[block_id] = None
                # Now that I'm aware of the corruption, remove the tuple from corrupted_blocks
                node.corrupted_blocks.remove((owner, block_id))
            else:
                # Invalidate the block
                block_id = block
                node.local_blocks[block_id] = False  # Set False in local_blocks
                # Now that I'm aware of the corruption, remove the block_id from corrupted_blocks
                node.corrupted_blocks.remove(block_id)

        # From this moment, the system will try to restore the block, aware that previously it has been corrupted and lost
```

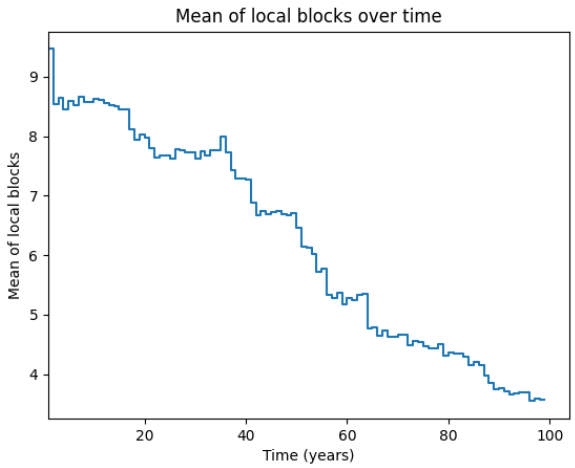
VII. Results

Our analysis focused on understanding the influence of varying parameters on the system's behavior, enabling us to draw plots of meaningful conclusions.
Please note that the following plots represent some possible runs, without any fixed seed, so there can be slight differences with other runs (e.g. a sudden decrease / increase in certain years), but the general behavior should be the same.

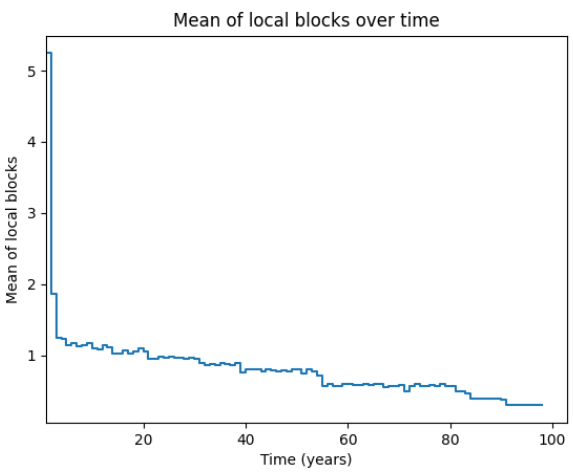
1. Plots of storage with corruption event (only extension 1)

1.1 P2P

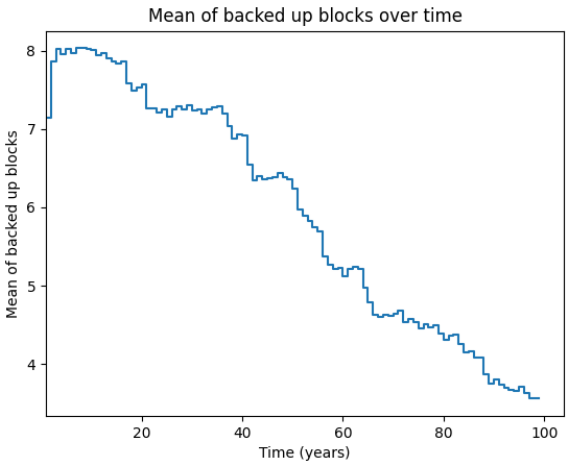
Without corruption



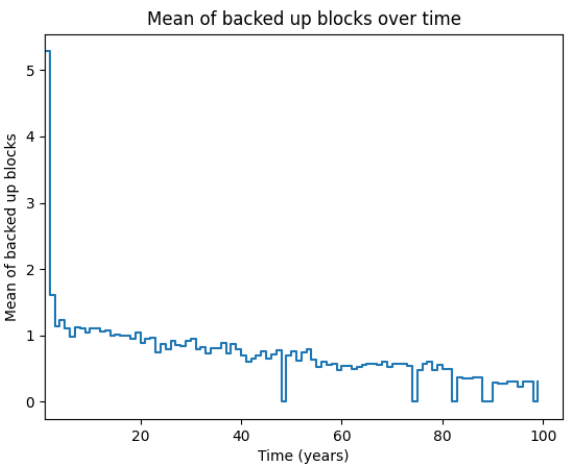
With corruption

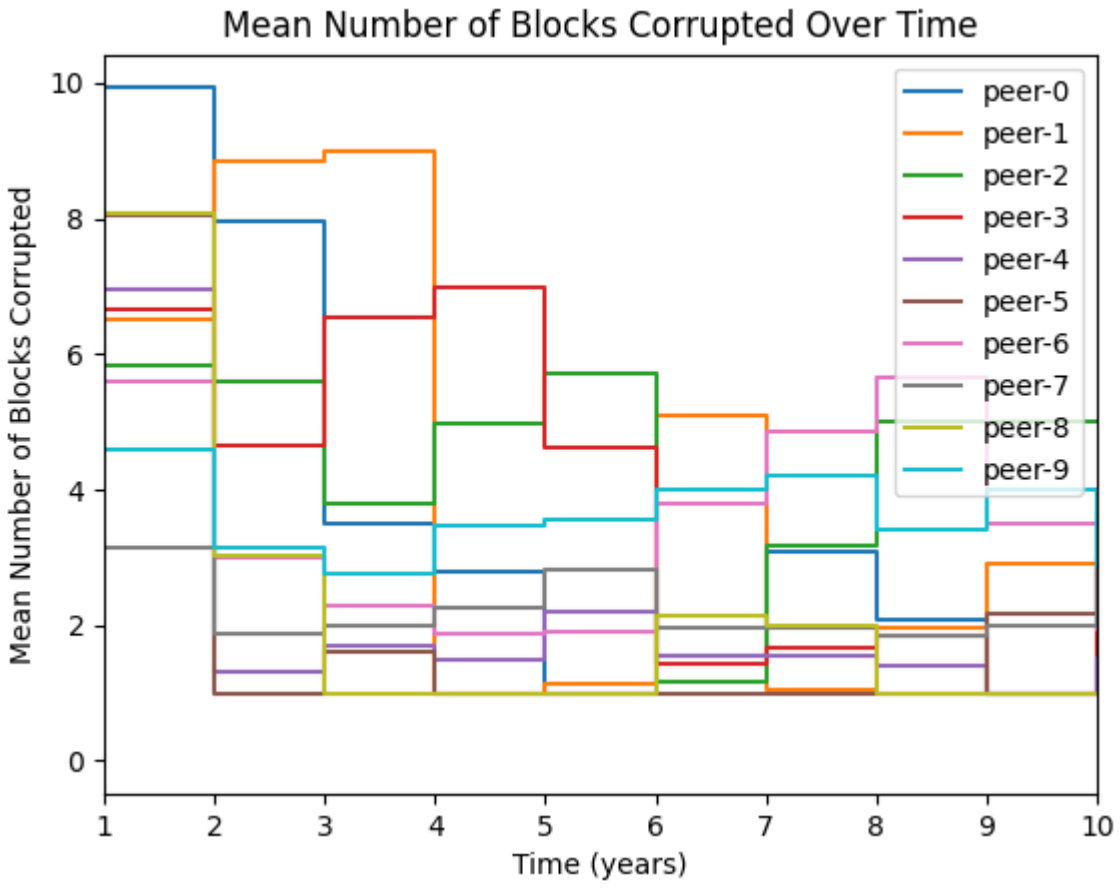


Without corruption



With corruption



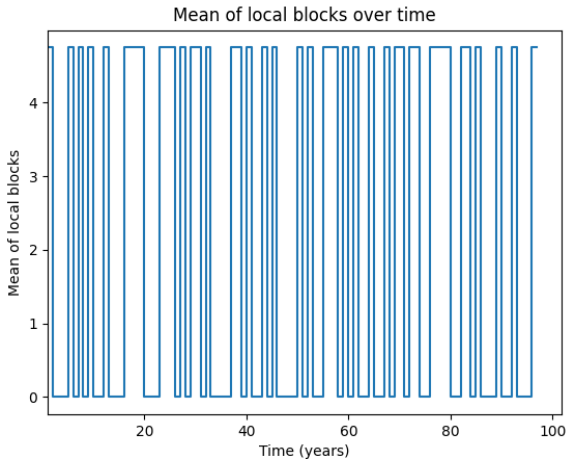


What we observe:

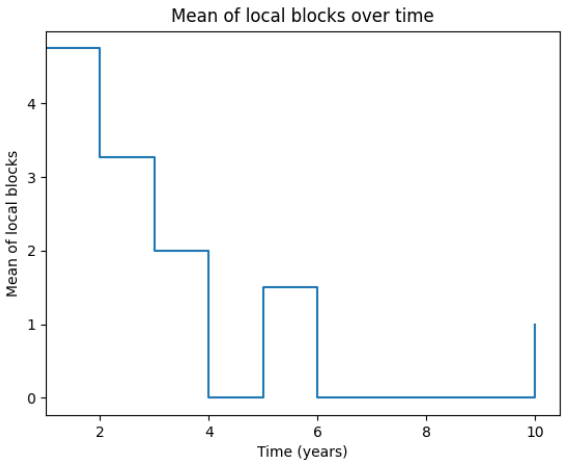
- In the first two plots, with a 100-years simulation, corrupted blocks may not be all restored in time, and mixing it with failures of nodes brings the system to lose much and much more of all the blocks to backup. But still the nodes work for all the duration of the simulation.
- In the last plot, limited in years only between 1-10 in order to make it understable to the human eye, we can see how the corruption influences all the peers, and we can observe how much they get aware of the corruption between themselves and consequently how many blocks they restore.

1.2 Client-Server

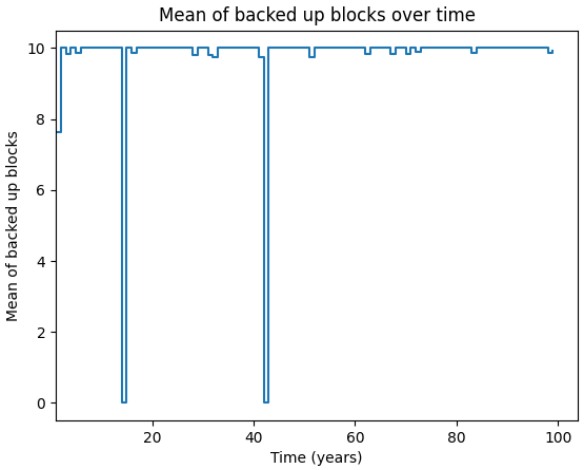
Without corruption



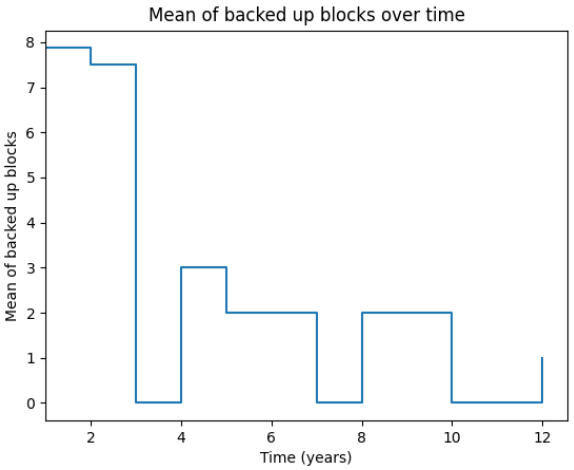
With corruption

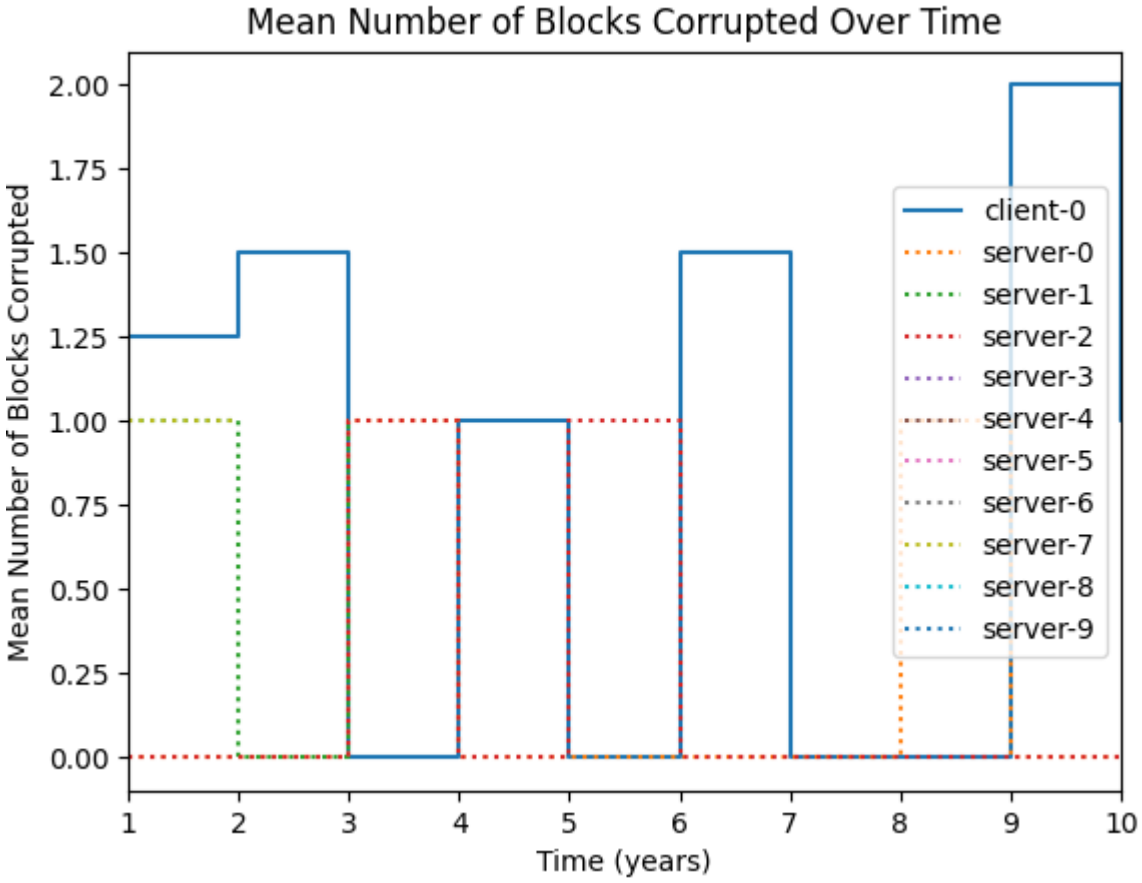


Without corruption



With corruption





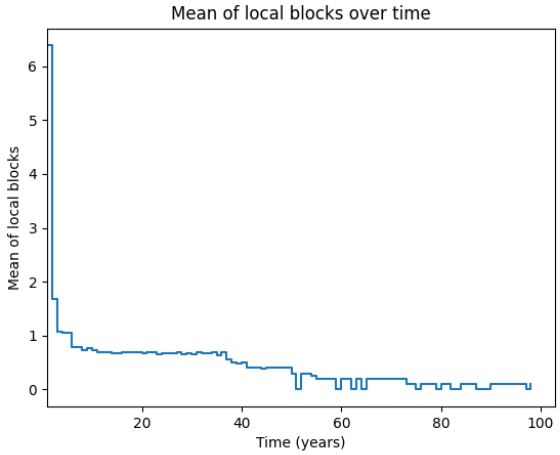
What we observe:

- In the first two plots the simulation, done for 100 years, stops much earlier because of the same motivation as before: corrupted blocks may not be all restored in time, and mixing it with failure of nodes brings the system to lose all the blocks to backup after 10-12 years.
- In the last plot we can see how the corruption influences even the server nodes, stopping to hold the only block they have as backup for the client. That means that if the client fails after this moment and that block on the server hasn't been restored, that block is lost forever. That's the reason behind the plots described before. Note that servers don't talk to each other, so there are less blocks involved and less "dynamicity" in the plot. In this case the plot is manually cropped between 1-10 years.

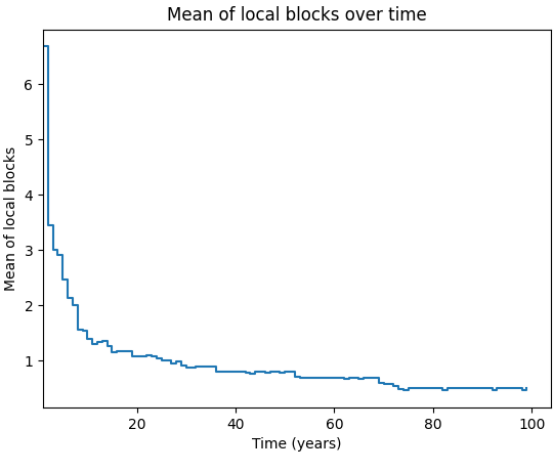
2. Plots of storage with also recovery event (extension 2)

2.1 P2P

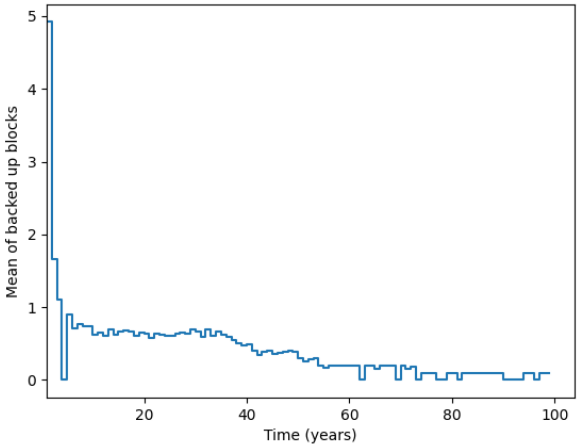
Without recovery



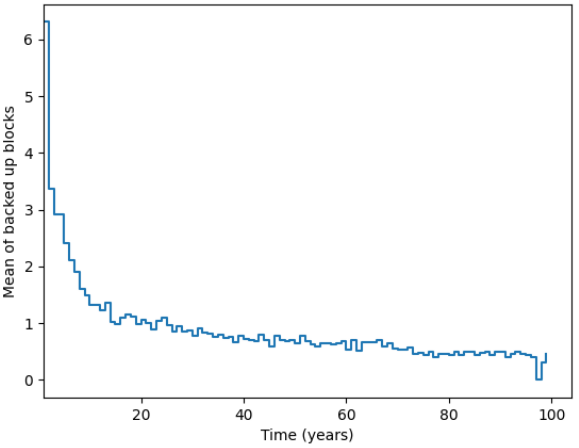
With recovery



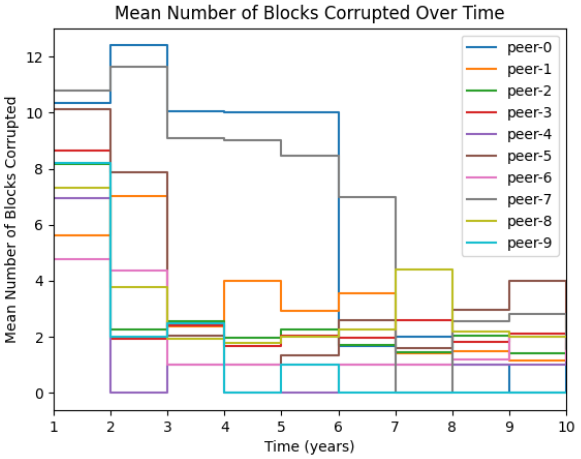
Without recovery



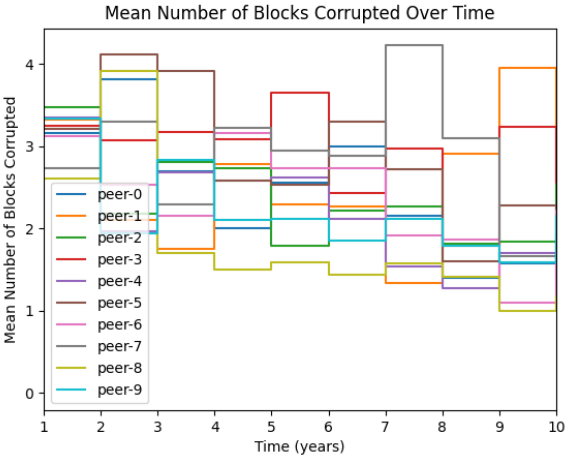
With recovery



Without recovery

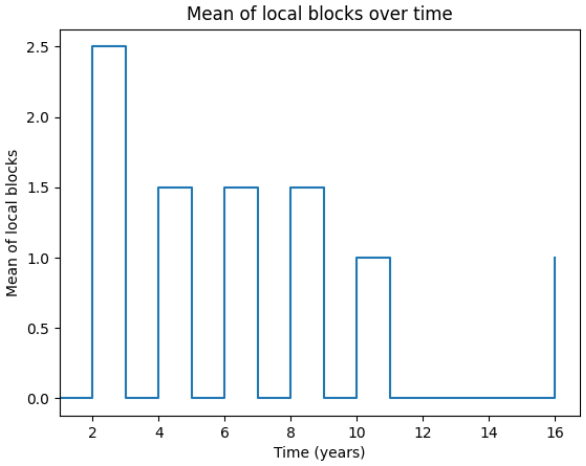


With recovery

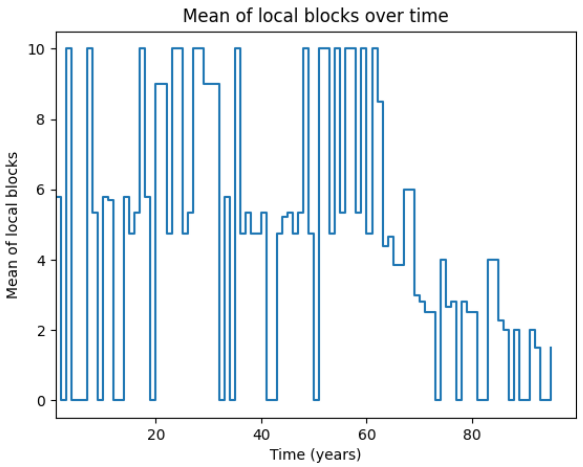


2.2 Client-Server

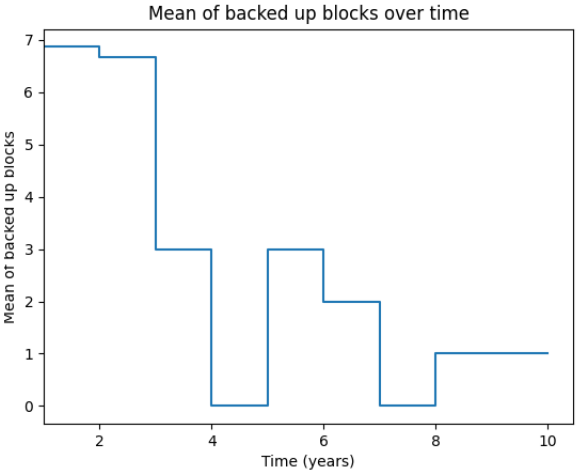
Without recovery



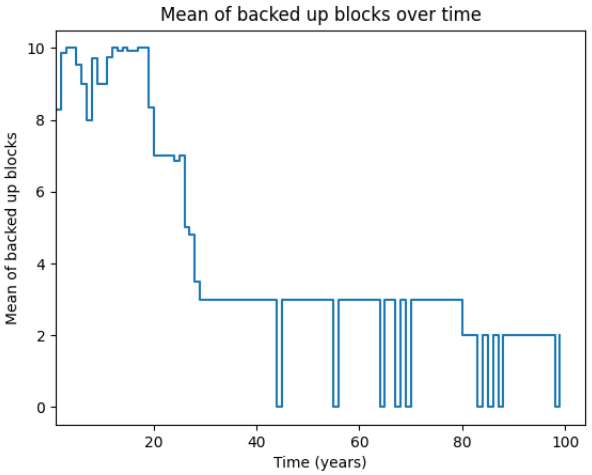
With recovery



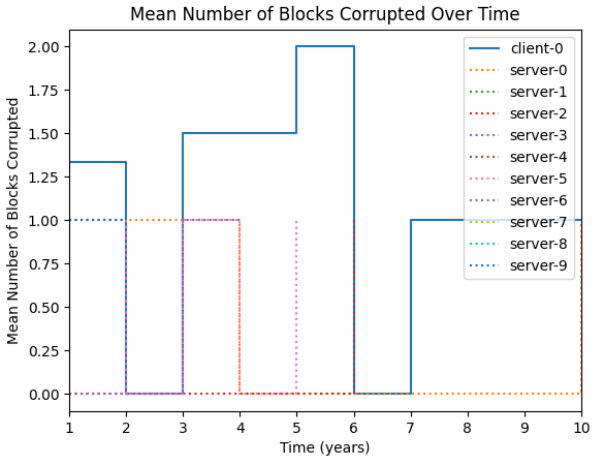
Without recovery



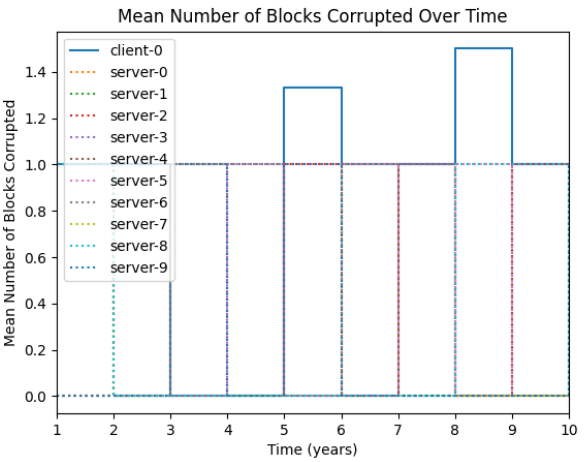
With recovery



Without recovery



With recovery

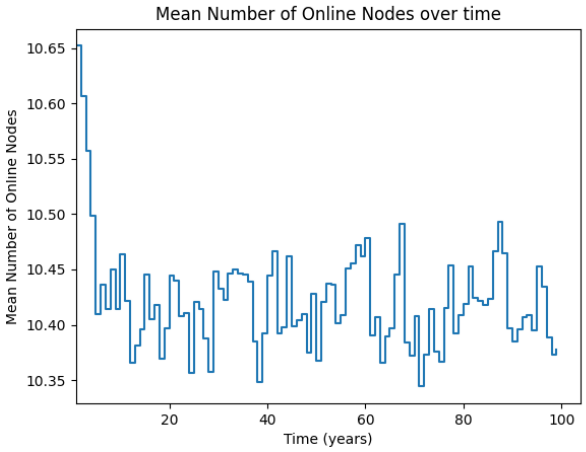


What we observe in both case scenario is more or less the same profile (descent), but with recovery the entire network is more dynamic and storage keeps up along the entire simulation: we can lose blocks and we can restore them when someone try to transfer them but now we also add an internal-node check that can repair “before it is too late” blocks, that is increasing the probability that the block will be restored once it has been set corrupted and the network is aware of that.

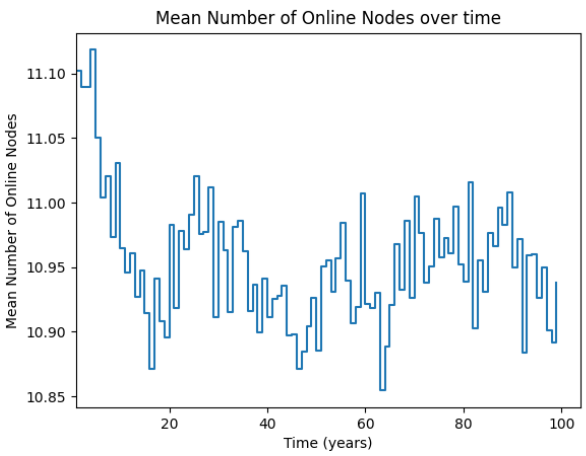
VIII. Configurations

We also decided to play with configurations, in particular with client_server.cfg in order to optimize it. The following plots use all the extensions.

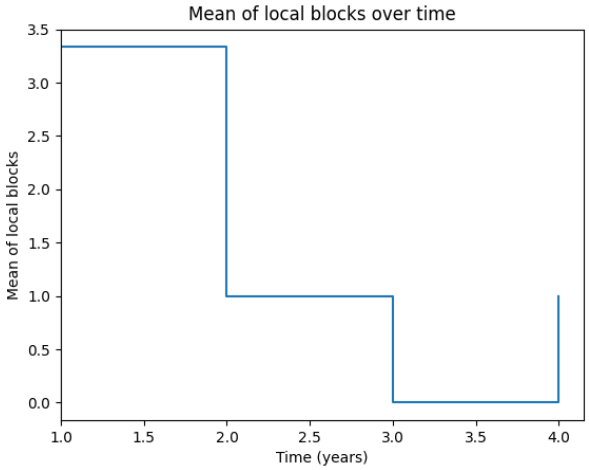
With original configuration



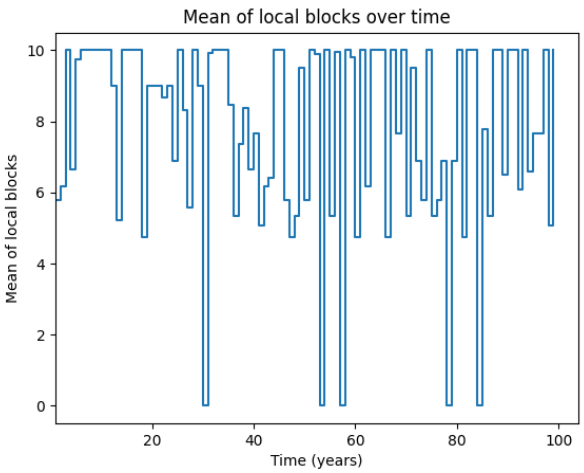
With our configuration



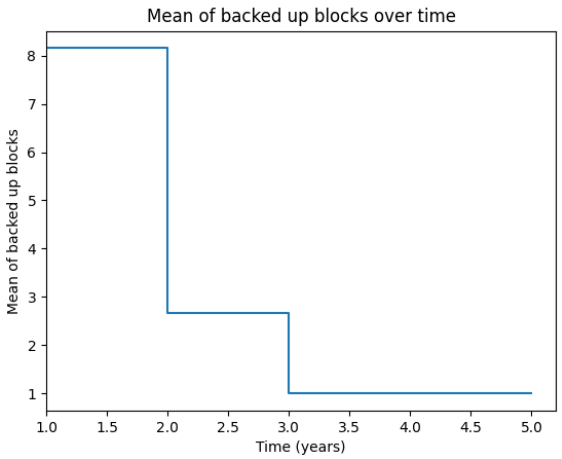
With original configuration



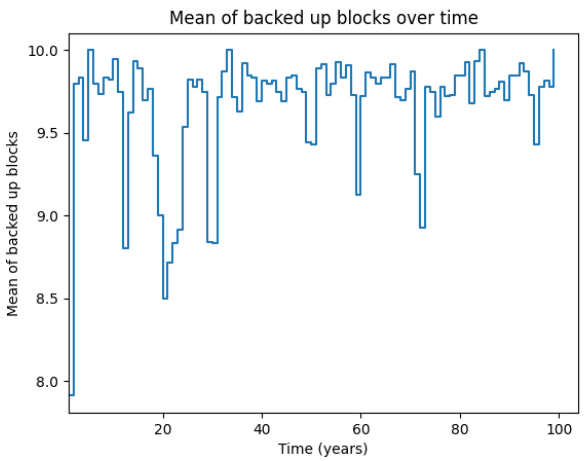
With our configuration



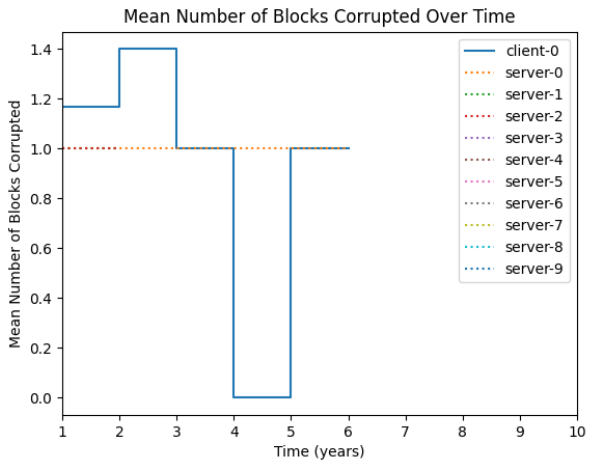
With original configuration



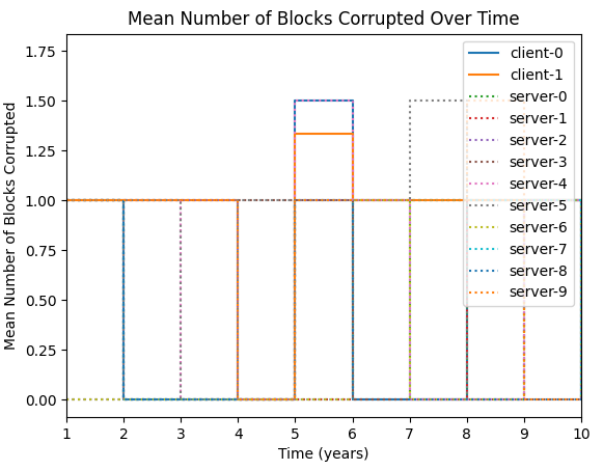
With our configuration



With original configuration



With our configuration



Original configuration + initial extension (corruption/integrity) intervals

```
[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
integrity_delay = 40 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
integrity_delay = 40 days
```

Modified (better avg_recovery_time + redundancy) config + improved extension args

```
[DEFAULT]  # parameters that are the same by default, for all classes
average_lifetime = 2 years
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 = 1 days
corruption_delay = 300 days
integrity_delay = 30 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 = 2 years
integrity_delay = 50 days
```

We can't of course tell what is better than the other: it's trivial since we are touching the "hardware" configuration of our nodes, and changing that also changes in some way the scenario. What we can see is that we can make the nodes last longer, in terms of local and backed up blocks "active" or "not corrupted", by obviously modifying the parameters which involve the corruption and the recovery (integrity) delay, but also increasing the redundancy. In fact, if we add one more client, a new P2P network starts and the exchange of local blocks between clients and servers is more dynamic and in some way more stable, since the corruption may occurs on one block of data of the client, but now another client holds a copy of that: so we have two copies, one on a server and one on another client (peer). That makes the simulation more "stable" and meaningful.

IX. Conclusions

At the end of our analysis, what we can see is that the introduction of our corruption extension highly influenced the entire network, and also the recovery does the same.
The set of actions we made allowed us to analyze the simulator accurately, both changing the configuration of our nodes or just by observing their behavior in an “ideal” situation, without any corruption. Even in this last case we have seen that some local blocks will inevitably get lost permanently, that is when multiple (and unfortunate) failures happen to certain blocks.

X. Appendix

1. Errors in the original code

In class Fail, which is the class - state of Node that fails and loses all the local data, between the removal of the all local_blocks and the remote_blocks_held we add the restoration of all the free space of the node “node.free_space” to the “node.storage_size” amount.
This restoration was missing and we thought that logically the Node wouldn’t have any more free space to save blocks when it continues to decrease when saving downloaded blocks.

The code may still work without this change, if the Node has a big amount of storage in the configuration for the duration of the simulation and so even decreasing won’t affect at the worst case scenario.

2. Setup to run the code

Both p2p.cfg and client_server.cfg have to be modified adding *corruption_delay* and *integrity_delay* parameters in order to make our code work without pointing out any runtime errors. We suggest adding reasonable values, for example the ones used in our_client_configuration.cfg (see appendix #3).

The code has been realized with Python version 3.12.0
To run the code simply call python.exe (or python3) on the storage.py script; to specify parameters use “--[name-of-parameter][space][value]”, without quotes and brackets.

Example: python.exe .\storage.py .\p2p.cfg –max-t “100 years”
Note that the code may take a while depending on the value of arguments and parameters of configuration.
Please note also that the simulation has some randomness and statistics within the scheduling of the events, so run the code multiple times to achieve a clear idea of the

program behavior.

If you want to enable extensions, please call the storage.py script with “--extension [1 or 2]”.

3. Two configuration files

As seen in the plots, we refer to two different file configurations of client-server scenarios. We try to put two different sets of values for varying parameters, one with a very weak hardware configuration (that is the original set of values with the *corruption_delay* and *integrity_delay* extraordinarily small) and so the system suffered more from it.
Then we try a possible good configuration, an optimal one, but still not perfect, that fits our interests.
We can always improve the hardware configuration by for example adding more space and strengthening our machines on transfer speeds or up-times, but it is not the objective.

4. External references

We used the *humanfriendly* library as suggested, so you need to install it to successfully run the code.

XI. Peer review

The following paragraph refers to the corrections and modifications suggested by the peer reviewer of our project. In particular we:

- added code snippets about our extension, in order to avoid to open the source code to have an overview of the implementation;
- improved the overall quality of the report, to make it more clear and readable;
- fixed the double occurrence of class DataCorrupted bug in the *storage.py* script due to an error on the GitHub merges of past commits; we also confirm that the behavior of the script has not changed neither before nor after the fix.