Distributed Storage Lab - Week 4

Written by: Marcell Feher <sw0rdf1sh@ece.au.dk>

Prerequisites

- You have completed the previous Labs
- You have formed a team
- You understand RAID levels

Goals

This week we try out distributed storage concepts for reliability. We implement a simple storage system that can store files using RAID 1. One process can offer a simple REST API to the outside world, and act as the controller. It stores the file metadata in a local sqlite3 database, and sends Protobuf commands to the other processes within your computer using ZeroMQ.

System Architecture

Storing a file with RAID 1 means that file contents are split to chunks, and every chunk is stored in two identical copies (see Fig.1). We consider four devices in our system, thus we will cut the incoming files in half, and store each half on 2 randomly selected nodes.

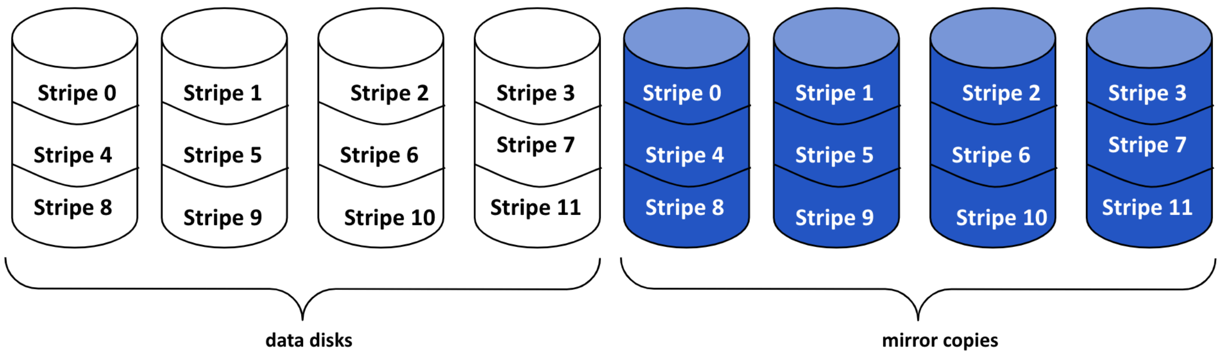
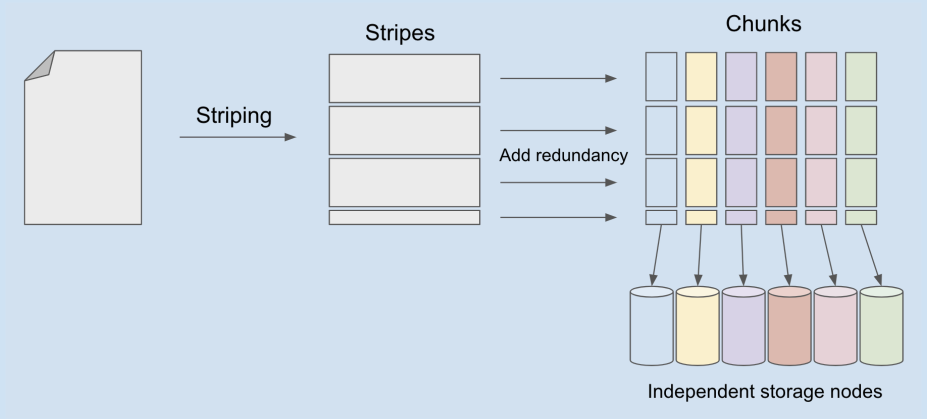


Figure 1: RAID 1 storage visualization.

Note
Real life RAID systems typically have an additional step when storing a new file. First, the file is sliced into fixed length stripes, and redundancy is added to each one independently (see figure below). The stripe size is a user defined setting of the system, a good value depends on the specific workload (e.g. average file size, average bytes read at once, etc). In this lab

we will process the whole file as a single stripe.



We'll use the system architecture shown on Figure 2. One of the "devices" (processes) will act as the controller and provide a REST API to the outside world where clients can submit requests. This component implements RAID 1 and distributes the data chunks to the storage nodes. It uses a local sqlite3 database to store file metadata persistently. We run a Storage Node component, which is responsible for storing and retrieving data chunks. Internally the components communicate via Protobuf messages on ZMQ channels.

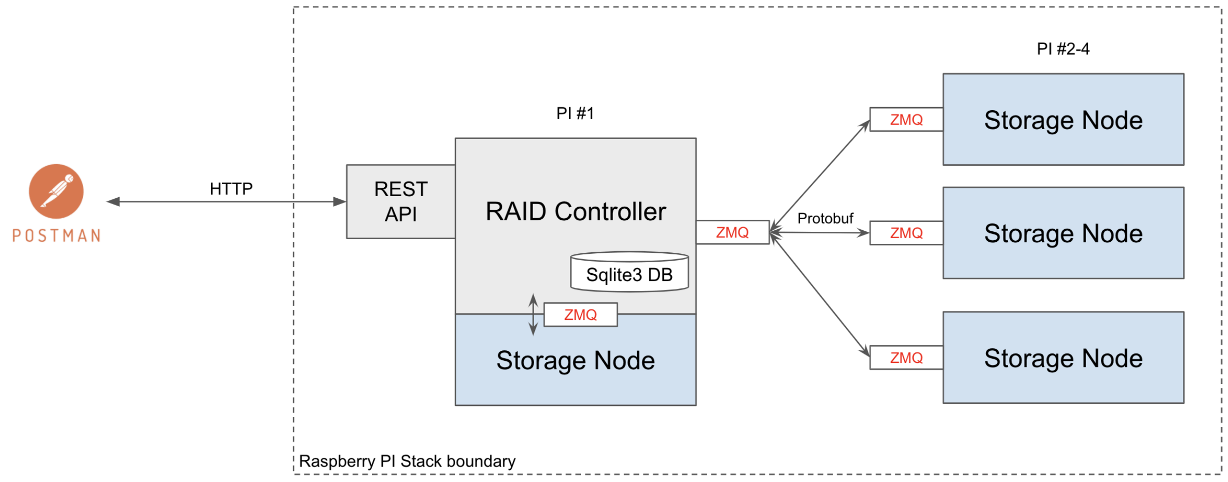


Figure 2: System architecture of the RAID system

Ingest

Data ingest is the process where a new piece of data (file, object, etc) is added to a storage system. In our RAID 1 system the ingest consists of the following high-level steps:

1. The file arrives from the client in a HTTP POST request
2. Decode the serialized file from base64 string to binary

3. Slice the file in half
4. Generate 4 random chunk names, two for each half
5. Send four Protobuf messages to the connected Storage Nodes
    a. 2 with the first half of the file and a chunk name
    b. 2 with the second half and a chunk name
6. The storage nodes each receive one of the four messages and store the data chunk
7. The controller waits for 4 replies and adds the record of the new file in its local sqlite database
8. The controller returns the ID of the new file to the client in a HTTP response

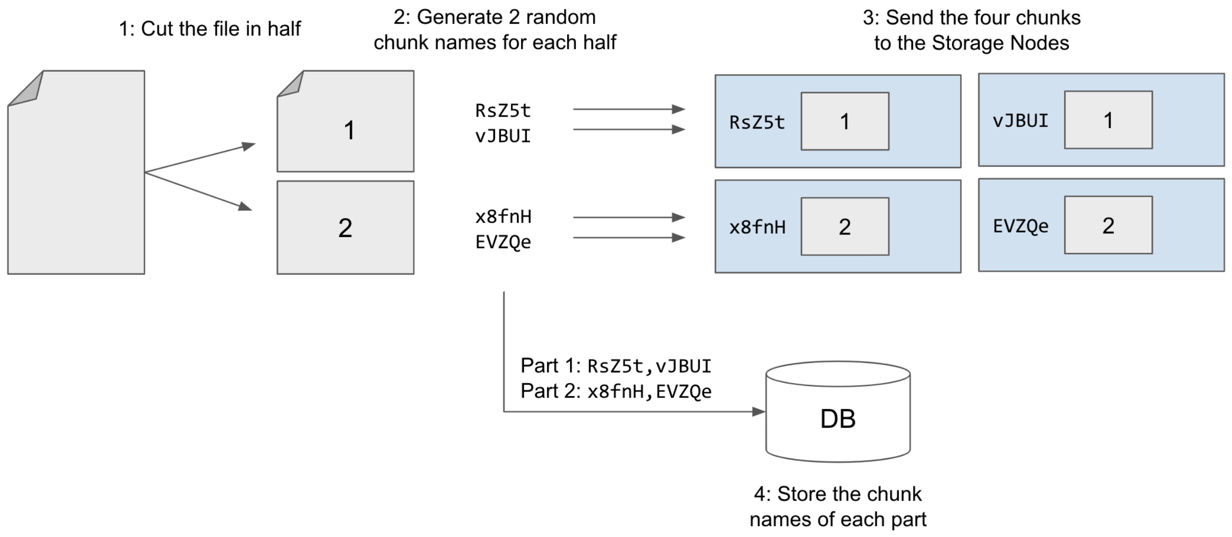

1: Cut the file in half
2: Generate 2 random chunk names for each half
3: Send the four chunks to the Storage Nodes
1
2
RsZ5t
vJBUI
x8fnH
EVZQe
RsZ5t 1
vJBUI 1
x8fnH 2
EVZQe 2
Part 1: RsZ5t,vJBUI
Part 2: x8fnH,EVZQe
DB
4: Store the chunk names of each part

Figure 3: Conceptual ingest process

Download

When a file is requested by the client, the following steps happen within our system:

1. The incoming HTTP request includes the File ID
2. The controller looks up the File record in the sqlite database. The File should have two parts and multiple chunk names for each part (2 in our case).
3. The controller randomly selects one of the replicas of each chunk and requests it from the storage nodes
4. The nodes that store the requested chunks send them to the controller
5. The controller waits until both chunks has arrived, glues them together and returns the file to the client in a HTTP response

From the processes above we can identify all necessary parts of our distributed system:

- A Flask app that provides a REST API

- A Controller that implements RAID and uses the Storage Nodes to persistently store the data chunks. Technically this could be a separate component from the Flask app, but since the whole system is really simple, we'll implement these together.
- An Sqlite database where the Controller keeps track of the files and their chunks
- Four Storage Nodes that accept two kinds of requests from the controller:
    - A "*Store Chunk*" message, that consists of a name and a chunk of data. The storage node is responsible for saving the data persistently and assigning the received name to it, so that it can be retrieved by the same name later. We will use the local file system to store the chunk and set the file name to the given name. Note, that we could have used the same technique as we did in the Flask lab: generate a random filename and use a local sqlite DB to map this to the given name. Since we already receive a random name and the file system lookup is fast enough for our purposes, we'll skip this and just store the chunk with the same name.
    - The "*Get Chunk*" message asks for a chunk by its name. If the node has this chunk, it should return it in a response message.

The Controller and Storage Nodes communicate via ZMQ channels and Protobuf messages. Let's address the ZMQ setup first.

ZMQ Channels

The two dialogues within the system (store and get chunk request-response) look very similar, but there is a fundamental difference between them. When the Controller issues the Store Chunk command, it sends four messages with different contents (chunk names and data). However, when the Get Chunk commands are sent, the message contents are identical and should be delivered to every Storage Node. Therefore, we will use two different ZMQ socket types for these messages: PUSH-PULL for storing the chunks and PUB-SUB for retrieving them (see Fig.4).

An added benefit of PUSH-PULL is that load balancing between the Storage Nodes comes for free. If the four Store messages are sent out in quick succession, each of them will be pulled by a different Storage Node. Additionally, the Controller doesn't need to know the IP addresses of the Nodes, and it's even okay if not all Nodes are available at all times. Of course, if less than 4 nodes are available then the storage system wouldn't be RAID 1 any more, but the four chunks would still be stored persistently, just on fewer nodes. It's also fine if additional Storage Nodes join the system: they can participate in storing new chunks without any coordination.

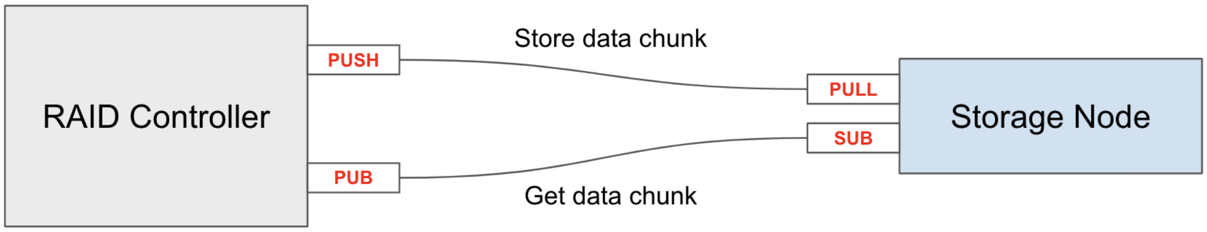


Figure 4: Different ZMQ sockets are used for the two dialogues

Protobuf Messages

Regarding the messages, there are four of them between the components:

- Store Chunk request: includes the chunk name and data
- Store Chunk response: just a signal of success, doesn't need to have any contents
- Get Chunk request: chunk name only
- Get Chunk response: chunk name and data

Two of these types have both text and binary components. ZeroMQ allows mixing different types of data in a single message called a Multipart Message. In this case a message consists of multiple "frames", where the encoding of each frame is independent of any other frames. This way we can compose a message where the first frame is a string (just the filename, or a serialized Protobuf message object), and the second is raw binary data. For example, the following code snippet shows how to send a serialized Protobuf message and binary data together:

```
task = messages_pb2.storedata_request()
task.filename = name

# Send a multipart message by passing a list to socket.send_multipart()
send_task_socket.send_multipart([
    task.SerializeToString(),
    file_data_1
])
```

Receiving and processing a multipart message is similarly easy:

```
# Receive a multipart message with recv_multipart()
msg = receiver.recv_multipart()
# Reconstruct the Protobuf message from the first frame
task = messages_pb2.storedata_request()
task.ParseFromString(msg[0])
```

```
chunkname = task.filename

# The raw data is the second frame
data = msg[1]
```

To demonstrate using Flask, ZMQ and Protocol Buffers together, we will define Protobuf message schemas for the two request messages, even though only one piece of information is being sent (the chunk name). Besides serving as an example, the Protobuf message definition makes the code easily extensible later.

We will use the following message definitions. Since the payload is identical we could have used a single one, but it would be more complicated to extend to the protocol later. In real systems it's common to define a separate request and response for each message.

```
syntax = "proto3";

message storedata_request
{
   string filename = 1;
}

message getdata_request
{
   string filename = 1;
}
```

Database

The controller needs to store the metadata of the uploaded files in a searchable persistent store. At the second lab we used a simple sqlite table for the same purpose, when the REST API stored the file ID, name, size, content type and timestamp. This time we'll modify that schema to fit the RAID 1 storage strategy.
Our controller needs to store the 4 random generated chunk names, and which belong to the first and second half of the file. It doesn't need to store the locations of the chunks (e.g. Storage Node index), since it broadcasts the Get Chunk request to all 4 nodes, and they respond if they have the chunk.
We will use the following, slightly modified table schema:

```sql
CREATE TABLE `file` (
  `id` INTEGER PRIMARY KEY AUTOINCREMENT,
  `filename` TEXT,
  `size` INTEGER,
  `content_type` TEXT,
  `created` DATETIME DEFAULT CURRENT_TIMESTAMP,
  `part1_filenames` TEXT,
  `part2_filenames` TEXT
);
```

The last two columns will store two chunk names each, concatenated to a single string with a comma as a delimiter (e.g. "*RsZ5t,vJBUI*").
In the following sections we will implement the system.

Running Locally

We will build the distributed storage system on a single computer. Since we are using ZMQ, the code will be very easily scalable to multiple nodes, all we need to do is update a few addresses.

To start, create a new folder for this week's lab and download the **`rest-server_week4_starter.py`** starter template, and rename it to **`rest-server.py`** on your computer. This Flask application is the finished file storage REST API that we worked on at week 2, with the helper functions moved to the beginning. Our goal is to change the storage model of the uploaded files from simply writing the data to the local hard drive to adding RAID 1 redundancy and distributing the chunks to 4 storage nodes.

We have the following tasks:

1. Create a new Sqlite3 database that can store the new representation of the stored files (4 chunk names instead of 1 blob name)
2. Define the Protobuf messages between our components
3. Change the implementation of add_files(), download_file() and delete_file() in rest-server.py from writing and reading the file using the local file system to sending and receiving chunks to the storage nodes on ZMQ channels as described above
4. Implement the Storage Node component
5. Test the system

The first two steps are straightforward, let's complete them quickly.

Task 1: Create an Sqlite3 database for the File records.

Download the **`create_table.sql`** file from Blackboard to and move it next to your rest-server.py. Create a new database with this table definition using the following command in a terminal:
Windows: **`sqlite3.exe files.db ".read create_table.sql"`**
macOS and Linux: **`sqlite3 files.db ".read create_table.sql"`**

Task 2: Save the protobuf message definition above as `messages.proto`, and generate Python code from it.
Run the following command after saving the file: **`protoc messages.proto --python_out=.`**
This should produce the `messages_pb2.py` source file in the same folder.

Task 3: Set up the ZMQ channels in `rest-server.py`
We will need 3 ZMQ sockets in the rest server: a PUSH for sending "Store Chunk", a PUB for issuing "Get Chunk" requests and a PULL to receive responses. After creating the sockets, we'll wait a second to give time for the Storage Nodes to connect. This implementation assumes that the worker nodes are running when the server is started. A good place to create these is after the helper functions but before instantiating the Flask application.

```
# Initiate ZMQ sockets
context = zmq.Context()

# Socket to send tasks to Storage Nodes
send_task_socket = context.socket(zmq.PUSH)
send_task_socket.bind("tcp://*:5557")

# Socket to receive messages from Storage Nodes
response_socket = context.socket(zmq.PULL)
response_socket.bind("tcp://*:5558")

# Publisher socket for data request broadcasts
data_req_socket = context.socket(zmq.PUB)
data_req_socket.bind("tcp://*:5559")

# Wait for all workers to start and connect.
time.sleep(1)
print("Listening to ZMQ messages on tcp://*:5558")
```

Also add the following lines to the imports section of rest-server.py (they include every new import we will need)

```python
import zmq # For ZMQ
import time # For waiting a second for ZMQ connections
import math # For cutting the file in half
import random # For selecting a random half when requesting chunks
import messages_pb2 # Generated Protobuf messages
import io # For sending binary data in a HTTP response
```

Task 4: Update the implementation of `add_files()` to use RAID 1 and the Storage Nodes!
Currently this endpoint stores the file locally with a random generated filename. To change this to RAID 1 as discussed earlier, we need to cut the file in half and store both halves twice. We will generate a random chunk name for all 4 chunks on the server and ask the Storage Nodes to assign this name to the stored chunk. This way the server doesn't need to store where the chunks are located (the addresses of the Storage Nodes). This makes our life easier and our code shorter.
Delete the `blob_name = write_file(file_data)` line and its comment, and replace it with the following code:

```python
# RAID 1: cut the file in half and store both halves 2x
file_data_1 = file_data[:math.ceil(size/2.0)]
file_data_2 = file_data[math.ceil(size/2.0):]

# Generate two random chunk names for each half
file_data_1_names = [random_string(8), random_string(8)]
file_data_2_names = [random_string(8), random_string(8)]
print(f"Filenames for part 1: {file_data_1_names}")
print(f"Filenames for part 2: {file_data_2_names}")

# Send 2 'store data' Protobuf requests with the first half and chunk names
for name in file_data_1_names:
   task = messages_pb2.storedata_request()
   task.filename = name
   send_task_socket.send_multipart([
       task.SerializeToString(),
       file_data_1
   ])

# Send 2 'store data' Protobuf requests with the second half and chunk names
for name in file_data_2_names:
   task = messages_pb2.storedata_request()
   task.filename = name
   send_task_socket.send_multipart([
       task.SerializeToString(),
       file_data_2
   ])

# Wait until we receive 4 responses from the workers
for task_nbr in range(4):
   resp = response_socket.recv_string()
   print(f"Received: {resp}")

# At this point all chunks are stored, insert the File record in the DB
```

Since our database schema changed as well, update the query string and parameters list to the following:

```
# Insert the File record in the DB
db = get_db()
cursor = db.execute(
   "INSERT INTO `file`(`filename`, `size`, `content_type`, `part1_filenames`,
`part2_filenames`) VALUES (?,?,?,?,?)",
   (filename, size, content_type, ','.join(file_data_1_names), ','.join(file_data_2_names))
)
db.commit()
```

Task 5: Update the implementation of `download_file()` to use RAID 1 and the Storage Nodes!

Instead of simply reading the file at blob_name, we'll do the following:

1. Load the File record from the DB (same as before)
2. Randomly select one chunk name for each half of the file
3. Send two "Get Chunk" messages on the PUB socket, one for each chunk
4. Wait for the two chunks to arrive on the PULL socket
5. Reconstruct the original file from the two halves and return it to the caller

Delete the return statement of the current implementation, and add the following code (after the "*File requested: ...*" print command):

```python
# Select one chunk of each half
part1_filenames = f['part1_filenames'].split(',')
part2_filenames = f['part2_filenames'].split(',')
part1_filename = part1_filenames[random.randint(0, len(part1_filenames)-1)]
part2_filename = part2_filenames[random.randint(0, len(part2_filenames)-1)]

# Request both chunks in parallel
task1 = messages_pb2.getdata_request()
task1.filename = part1_filename
data_req_socket.send(
   task1.SerializeToString()
)
task2 = messages_pb2.getdata_request()
task2.filename = part2_filename
data_req_socket.send(
   task2.SerializeToString()
)

# Receive both chunks and insert them to
file_data_parts = [None, None]
for _ in range(2):
   result = response_socket.recv_multipart()
   # First frame: file name (string)
   filename_received = result[0].decode('utf-8')
   # Second frame: data
   chunk_data = result[1]

   print(f"Received {filename_received}")

   if filename_received == part1_filename:
       # The first part was received
       file_data_parts[0] = chunk_data
   else:
       # The second part was received
       file_data_parts[1] = chunk_data

print("Both chunks received successfully")
```

```
# Combine the parts and serve the file
file_data = file_data_parts[0] + file_data_parts[1]
return send_file(io.BytesIO(file_data), mimetype=f['content_type'])
```

Task 6: Implement the Storage Node component
This standalone Python program runs forever and listens on two ZMQ channels. A PULL socket on port 5557 receives "Store Chunk" requests from the controller, which consists of two frames: a serialized **`storedata_request`** Protobuf message followed by the raw binary chunk data. A separate SUB socket listens to "Get Chunk" requests on port 5559, which is a simple ZMQ message that holds a serialized **`getdata_request`** message. The Storage Node responds on a PUSH socket on port 5558 to the controller.

Let's start by creating a new file called **`storage-node.py`**, initialize the sockets and set up an empty subscription so we receive every chunk request:

```
import zmq
import messages_pb2

import sys
import os
import random
import string

context = zmq.Context()

# Socket to receive Store Chunk messages from the controller
pull_address = "tcp://localhost:5557"
receiver = context.socket(zmq.PULL)
receiver.connect(pull_address)
print(f"Listening on {pull_address}")

# Socket to send results to the controller
sender = context.socket(zmq.PUSH)
sender.connect("tcp://localhost:5558")

# Socket to receive Get Chunk messages from the controller
subscriber = context.socket(zmq.SUB)
subscriber.connect("tcp://localhost:5559")
```

```
# Receive every message (empty subscription)
subscriber.setsockopt(zmq.SUBSCRIBE, b'')
```

So far we have seen components listening on a single socket. In this case, a forever running loop (while True) was used to wait for a new incoming message. To listen on multiple sockets at the same time, we need a Poller object. We can register the sockets we want to listen to in parallel, and the Poller takes care of receiving messages. Add the following code after setting up the sockets:

```
# Use a Poller to monitor two sockets at the same time
poller = zmq.Poller()
poller.register(receiver, zmq.POLLIN)
poller.register(subscriber, zmq.POLLIN)

while True:
   try:
       # Poll all sockets
       socks = dict(poller.poll())
   except KeyboardInterrupt:
       break

   # At this point one or multiple sockets have received a message

   if receiver in socks:
       # Incoming message on the PULL
       # TODO handle message
       pass

   if subscriber in socks:
       # Incoming message on the SUB socket
       # TODO handle message
       pass
#
```

We have covered setting up the sockets that are receiving messages. The next step is to implement the main functionality of the Storage Node: storing and retrieving the chunks. Since we will use the file system for this, let’s copy over the **`random_string()`** and **`write_file()`** functions from the controller to the beginning of **`storage-node.py`**.

We will use this Python program two ways: first, run 4 instances of it locally, simulating a distributed system, then deploy it on the Raspberry PIs and run one instance per computer. In the real system each Storage Node instance will only have access to the chunks they stored, but not to the others'. When we are testing locally, it's best to also separate the chunks saved by each Storage Node instance. To do this, we will allow the user to set a folder name where the chunks should be stored to and retrieved from, and we'll start the four local instances with four different folder name arguments. Add the following code snippet after the write_file() function, before starting to work with ZMQ:

```
# Read the folder name where chunks should be stored from the first program argument
# (or use the current folder if none was given)
data_folder = sys.argv[1] if len(sys.argv) > 1 else "./"
if data_folder != "./":
   # Try to create the folder
   try:
       os.mkdir('./'+data_folder)
   except FileExistsError as _:
       # OK, the folder exists
       pass
print(f"Data folder: {data_folder}")
```

This allows the program to read the first argument and use it as a folder name where chunks should be stored. It uses the **`os.mkdir()`** command to create the folder if it doesn't exist. For example, starting it with the **`python storage-node.py node1`** command will create the **/node1** subfolder and store the chunks there. When the program is started without any arguments (**`python storage-node.py`**), it uses the current folder to store chunks.

Everything is ready to implement the Store Chunk and Get Chunk message handlers. Store Chunk requests are multipart ZMQ messages that consist of two frames: a `storedata_request` Protobuf object in the first, and the raw chunk data in the second. After parsing both frames we simply call the `write_file()` function to store the chunk in our data folder with the given name, and return the file name on the PUSH socket.

```
if receiver in socks:
   # Incoming message on the 'receiver' socket where we get tasks to store a chunk
   msg = receiver.recv_multipart()
   # Parse the Protobuf message from the first frame
   task = messages_pb2.storedata_request()
   task.ParseFromString(msg[0])
```

```
    # The data is the second frame
    data = msg[1]

    print(f"Chunk to save: {task.filename}, size: {len(data)} bytes")

    # Store the chunk with the given filename
    chunk_local_path = data_folder+'/'+task.filename
    write_file(data, chunk_local_path)
    print(f"Chunk saved to {chunk_local_path}")

    # Send response (just the file name)
    sender.send_string(task.filename)
```

A Get Chunk request is a serialized `getdata_request` object. We use the **`with open(...)`** Python technique to try opening the chunk file in our data folder. The advantage of this method over the traditional **`file = open(...)`** call is that the file is closed automatically when it leaves the scope. If the requested chunk is found, we send it back on the PUSH socket as a multipart ZMQ message, where the first frame is the chunk name and the second is the data itself.

```
if subscriber in socks:
    # Incoming message on the 'subscriber' socket where we get retrieve requests
    msg = subscriber.recv()

    # Parse the Protobuf message from the first frame
    task = messages_pb2.getdata_request()
    task.ParseFromString(msg)

    filename = task.filename
    print(f"Data chunk request: {filename}")

    # Try to load the requested file from the local file system,
    # send response only if found
    try:
        with open(data_folder+'/'+filename, "rb") as in_file:
            print(f"Found chunk {filename}, sending it back")

            sender.send_multipart([
                bytes(filename, 'utf-8'),
                in_file.read()
            ])
    except FileNotFoundError:
        # The chunk is not stored by this node
        pass
```

This is all we need to test the whole system locally.

Task 7: Test the system on your local computer, using Postman!
First, start the Storage Node program four times in different terminal windows, passing different folder names:

```
$ python storage-node.py node1
Data folder: node1
Listening on tcp://localhost:5557
```

```
$ python storage-node.py node2
Data folder: node2
Listening on tcp://localhost:5557
```

```
$ python storage-node.py node3
Data folder: node3
Listening on tcp://localhost:5557
```

```
$ python storage-node.py node4
Data folder: node4
Listening on tcp://localhost:5557
```

Then start the controller:

```
$ python rest-server.py
Listening to ZMQ messages on tcp://*:5558
 * Serving Flask app "rest-server" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://localhost:9000/ (Press CTRL+C to quit)
```

The REST API is identical to Week 2, so you can test the system with Postman using the same requests as we did before. The base URL is http://localhost:9000.

Additional Tasks

Task 8 (optional): Extend the system with the DELETE functionality.
Use a ZMQ socket and message format that fits best for this request.