WEEK 5

September 28, 2024

Student Information

Full Name: Deniz Karakoyun
Id Number: 2580678

1 Introduction

ZeroMQ (ØMQ) is a high-performance asynchronous messaging library aimed at use in scalable distributed or concurrent applications. It provides a variety of messaging patterns and can be used to build complex communication architectures.

2 Basic Concepts

ZeroMQ provides several messaging patterns, including:

- **Request-Reply** - A pattern where a client sends a request and waits for a reply from the server.
- **Publish-Subscribe** - A pattern where publishers send messages to a topic and subscribers receive messages from the topics they are subscribed to.
- **Push-Pull** - A pattern where tasks are distributed among workers.
- **Dealer-Router** - A pattern providing an advanced form of request-reply with asynchronous request routing.

3 Installation

To use ZeroMQ in Python, you need the ‘pyzmq‘ package. You can install it using pip:

```
pip install pyzmq
```

4 Basic Code Examples

4.1 Request-Reply Pattern

4.1.1 Server Code

The following code shows a simple ZeroMQ server that replies to client requests.

```python
import zmq

# Create a context
context = zmq.Context()

# Create a REP (Reply) socket
socket = context.socket(zmq.REP)
socket.bind("tcp://127.0.0.1:5555")

while True:
    # Wait for a request from the client
    message = socket.recv_string()
    print(f"Received␣request:␣{message}")

    # Send a reply back to the client
    socket.send_string(f"Hello,␣{message}")
```

4.1.2 Client Code

The following code shows a simple ZeroMQ client that sends requests to the server and receives replies.

```python
import zmq

# Create a context
context = zmq.Context()
```

```python
# Create a REQ (Request) socket
socket = context.socket(zmq.REQ)
socket.connect("tcp://127.0.0.1:5555")

# Send a request
socket.send_string("World")

# Receive a reply
message = socket.recv_string()
print(f"Received reply: {message}")
```

4.2 Publish-Subscribe Pattern

4.2.1 Publisher Code

The following code shows a simple ZeroMQ publisher that sends messages on a topic.

```python
import zmq
import time

# Create a context
context = zmq.Context()

# Create a PUB (Publish) socket
socket = context.socket(zmq.PUB)
socket.bind("tcp://127.0.0.1:5556")

while True:
    # Send a message on topic "A"
    socket.send_string("A Hello")
    time.sleep(1)
    # Send a message on topic "B"
    socket.send_string("B World")
    time.sleep(1)
```

4.2.2 Subscriber Code

The following code shows a simple ZeroMQ subscriber that receives messages from the publisher.

```python
import zmq

# Create a context
context = zmq.Context()

# Create a SUB (Subscribe) socket
socket = context.socket(zmq.SUB)
socket.connect("tcp://127.0.0.1:5556")

# Subscribe to topic "A"
socket.setsockopt_string(zmq.SUBSCRIBE, "A")

while True:
    message = socket.recv_string()
    print(f"Received message: {message}")
```

5 API Overview

The ZeroMQ API provides various functions to create and manage sockets, and send and receive messages. Some common methods include:

- `context.socket(type)`: Creates a new socket of the specified type.
- `socket.bind(endpoint)`: Binds the socket to an endpoint.
- `socket.connect(endpoint)`: Connects the socket to an endpoint.
- `socket.send(message)`: Sends a message through the socket.
- `socket.recv()` : Receives a message from the socket.

6 ZeroMQ Overview

ZeroMQ is a high-performance asynchronous messaging library that allows for building scalable, distributed systems. In this experiment, the client sends messages of various sizes to the server, which processes the message and replies.

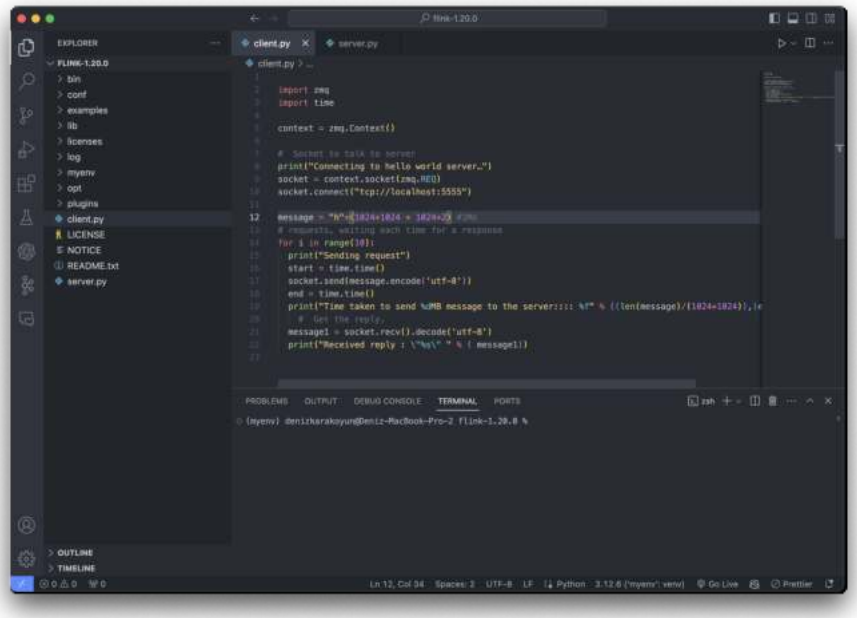


Figure 1: Client Code

7 Client Code

The client code receives messages from the client using ZeroMQ's REP socket, measures the time taken to receive the message, and replies to the client. Below is the server code:

Listing 1: Client Code

```
import zmq
import time

context = zmq.Context()

#  Socket to talk to server
print("Connecting to hello world server")
socket = context.socket(zmq.REQ)
socket.connect("tcp://localhost:5555")

message = "h"*(1024*1024 * 1024*2) #1Mb
# requests, waiting each time for a response
for i in range(10):
```

```
print("Sending␣request")
start = time.time()
socket.send(message.encode('utf-8'))
end = time.time()
print("Time␣taken␣to␣send␣%dMB␣message␣to␣the␣server::::␣%f" % ((
    len(message)/(1024*1024)),(end-start)))
  #  Get the reply.
message1 = socket.recv().decode('utf-8')
print("Received␣reply␣:␣\"%s\"␣" % ( message1))
```

Part by Part Explanation of Client Code

1. **Imports:**
   - `zmq` is imported to use ZeroMQ for socket communication.
   - `time` is imported to measure the time taken to send messages and receive responses.
2. **Creating Context:**
   - A `Context` object is created to manage the sockets used by the client. This context is necessary for creating sockets in ZeroMQ.
3. **Creating REQ Socket:**
   - A request socket (`zmq.REQ`) is created. This type of socket is used to send requests to a server and wait for replies.
4. **Connecting to Server:**
   - The client connects to a server running on `localhost` at port `5555`. The communication is over TCP.
5. **Preparing Message:**
   - A message is created as a string of `2GB` (`1024 * 1024 * 1024 * 2` bytes) by repeating the character `"h"`. This is intended to test the handling of large messages.
6. **Sending Requests and Measuring Time:**
   - **Loop**: Iterates 10 times, sending a request each time.

- **Timing**: Measures the time before and after sending the message using `time.time()`.
- **Send Message**: The message is encoded to bytes (`message.encode('utf-8')`) and sent to the server.
- **Print Time Taken**: The time taken to send the message is printed along with the size of the message in megabytes.

7. **Receiving Reply:**
   - **Receive Message**: Waits for and receives the server's reply.
   - **Decode**: Decodes the byte string reply back to a regular string using `decode('utf-8')`.
   - **Print Reply**: Prints the reply received from the server.

8 Server Code

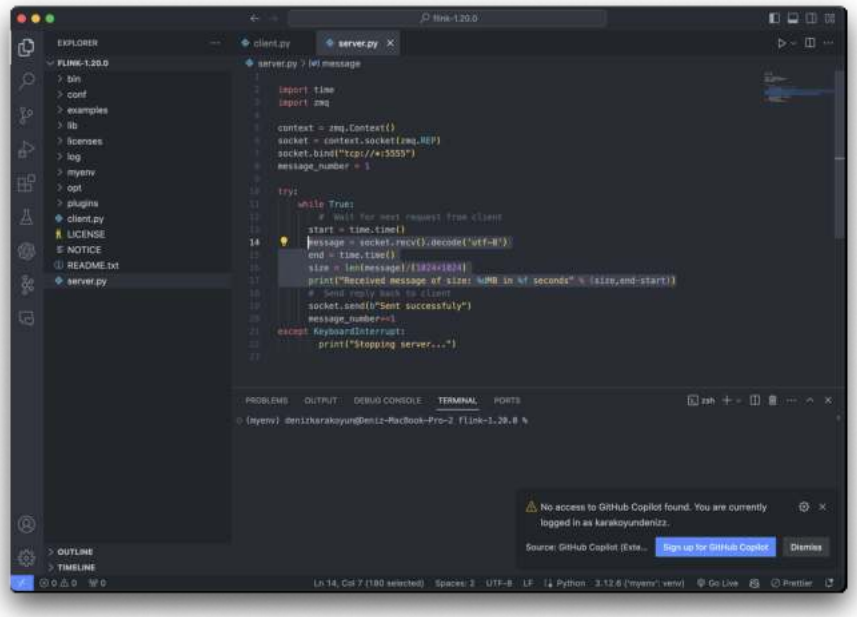


Figure 2: Server Code

The server code receives messages from the client using ZeroMQ's REP socket, measures the time taken to receive the message, and replies to the client. Below is the server code:

Listing 2: Server Code

```
import time
import zmq

context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5555")
message_number = 1

try:
    while True:
        start = time.time()
        message = socket.recv().decode('utf-8')
        end = time.time()
        size = len(message)/(1024*1024)
        print("Received message of size: %dMB in %f seconds" %
              (size, end-start))

        # Send reply back to client
        socket.send(b"Sent successfully")
        message_number += 1
except KeyboardInterrupt:
    print("Stopping server...")
```

8.1 Part by Part Explanation of Server Code

1. **Imports:** Similar to the client, `zmq` is used for ZeroMQ and `time` for timing operations.

2. **Creating Context:** A context object is created to manage the server's sockets.

3. **Creating REP Socket:** The server creates a reply (`REP`) socket. This type of socket waits for a request from the client and then sends a reply.

4. **Binding to Port:** The server binds the socket to port 5555 on all available interfaces.

5. **Infinite Loop:** The server enters an infinite loop waiting for incoming messages from clients.

6. **Receiving and Timing:** The server waits for a message from the client using `socket.recv()`. It calculates the time taken to receive the message and logs the message size.

7. **Replying to Client:** The server sends a reply ("Sent successfully") to the client after processing each message.

8. **Graceful Shutdown:** The server catches a `KeyboardInterrupt` (Ctrl+C) and gracefully stops the server by exiting the loop.

9 Timing Results

Below is the timing result table, with the message size, delivery time (from client to server), and receive time (from server to client).

Message Size (MB)	**Delivery Time (seconds)**	**Receive Time (seconds)**
1	0.000073	0.000543
2	0.000422	0.001730
4	0.000560	0.002320
8	0.000460	0.003271
16	0.001493	0.007853
32	0.004511	0.017101
64	0. 008909	0.032268
128	0.021017	0.078984
256	0.046990	0.158952
512	0.094557	0.304910
1024	0.212708	0.989352
2048	0.396469	1.216786

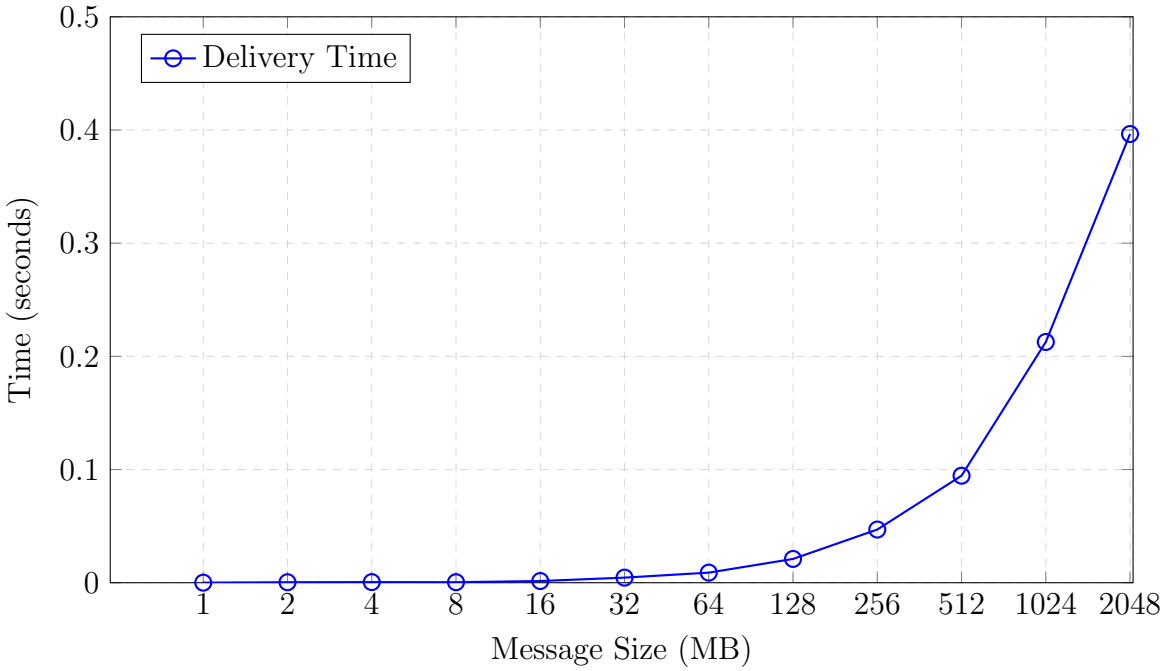


Figure 3: Message Size vs Time for Delivery

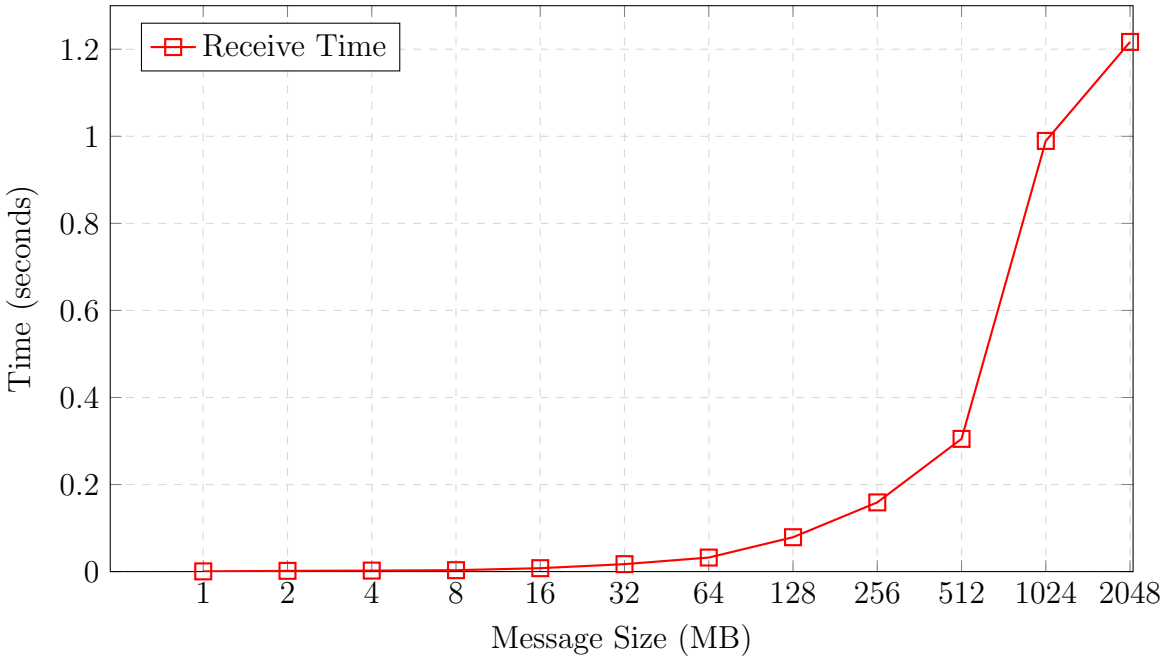


Figure 4: Message Size vs Time for and Receive

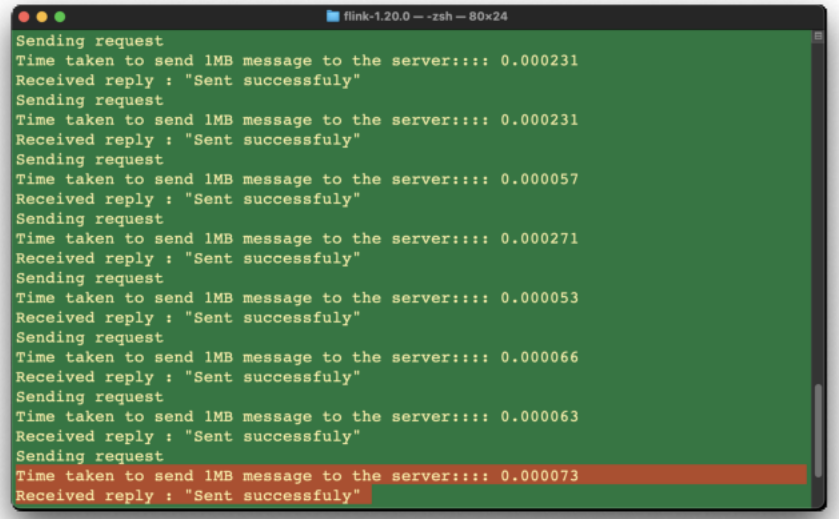


Figure 5: Delivery Time of 1MB Message

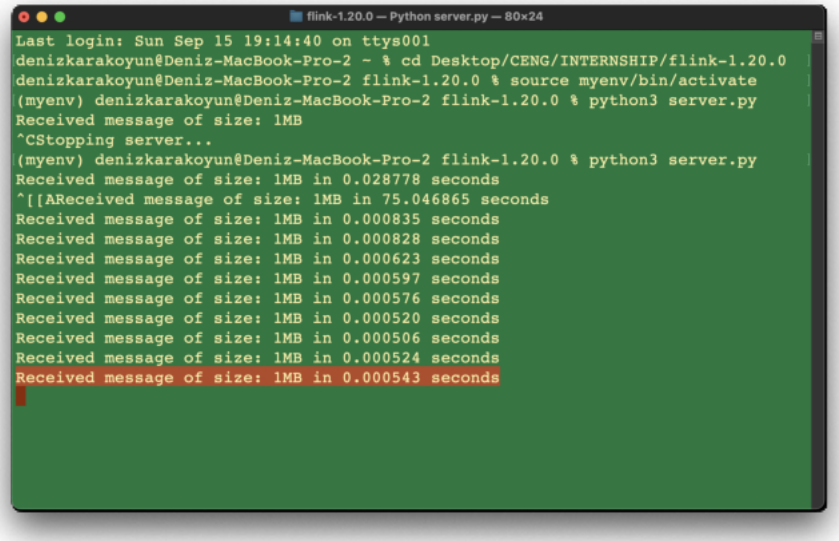


Figure 6: Receive Time of 1MB Message

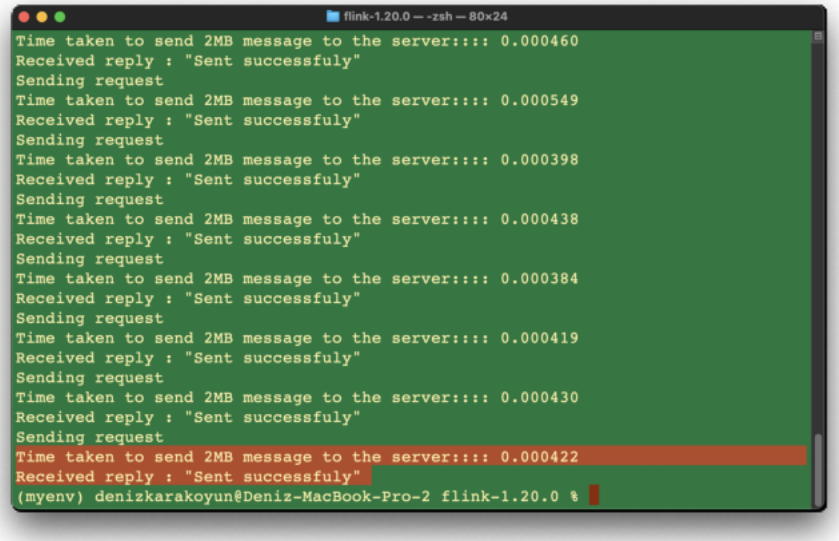


Figure 7: Delivery Time of 2MB Message

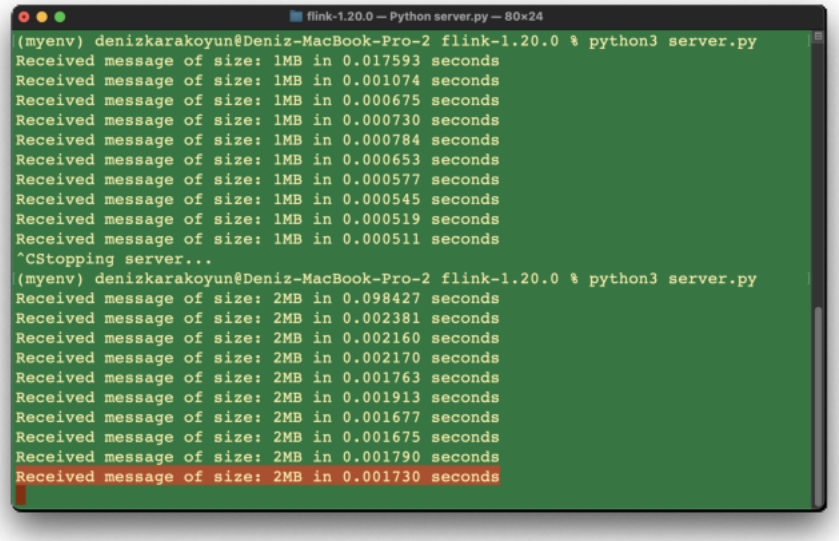


Figure 8: Receive Time of 2MB Message

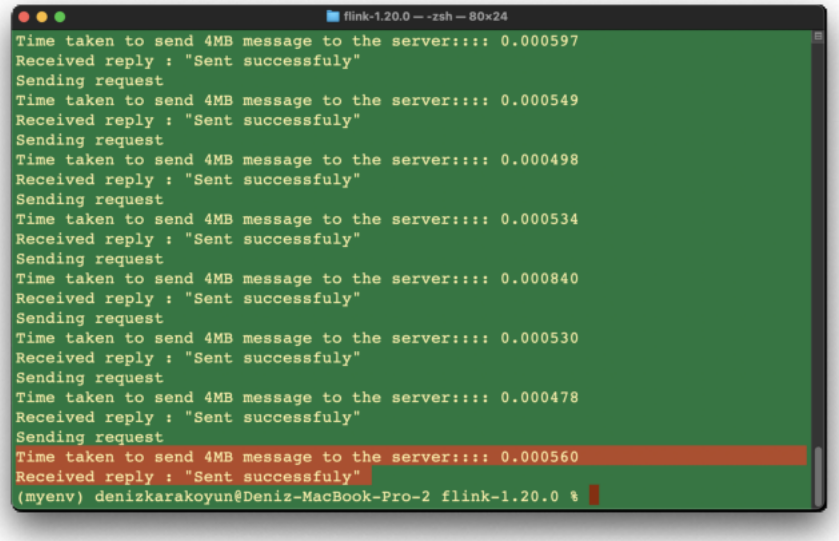


Figure 9: Delivery Time of 4MB Message

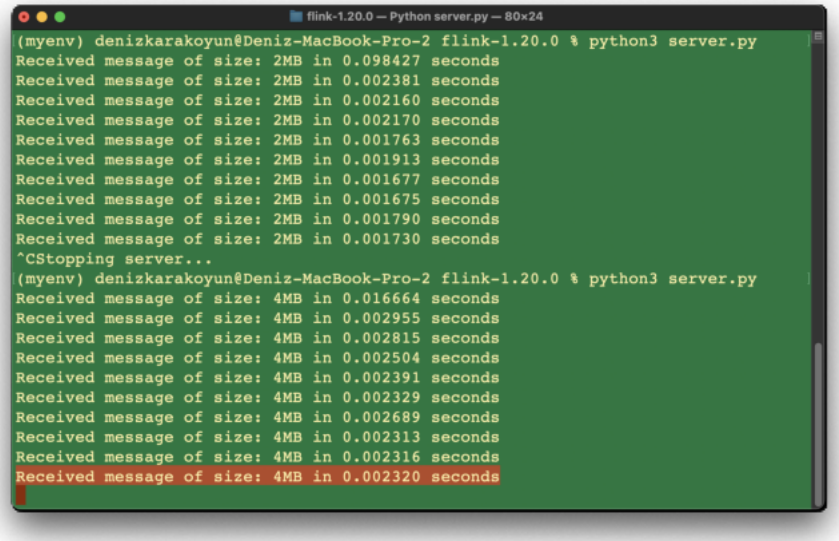


Figure 10: Receive Time of 4MB Message

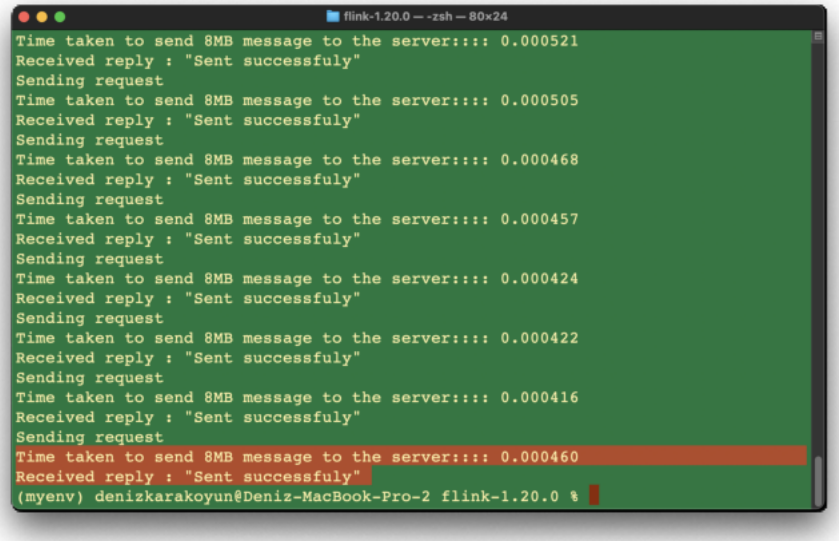


Figure 11: Delivery Time of 8MB Message

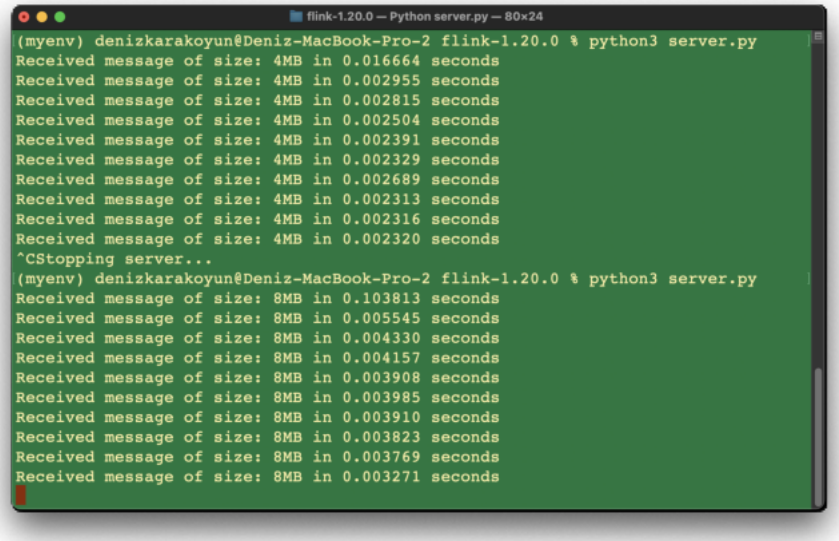


Figure 12: Receive Time of 8MB Message

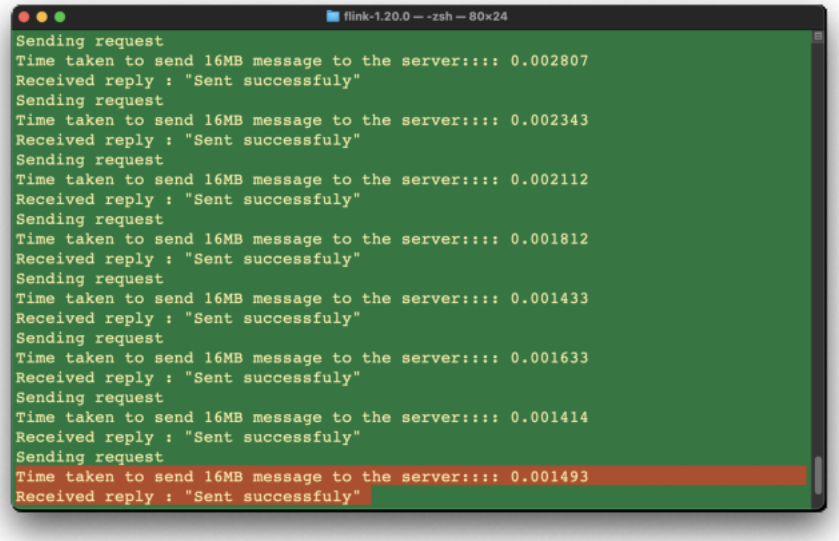


Figure 13: Delivery Time of 16MB Message

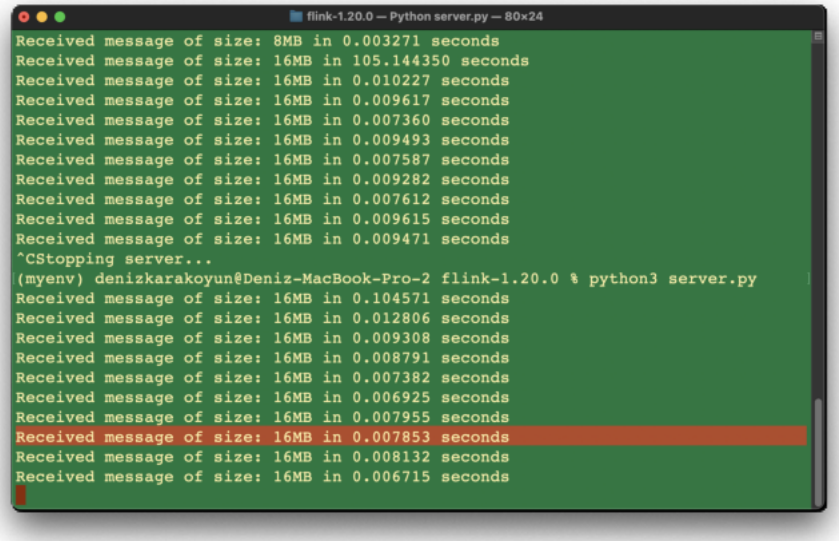


Figure 14: Receive Time of 16MB Message

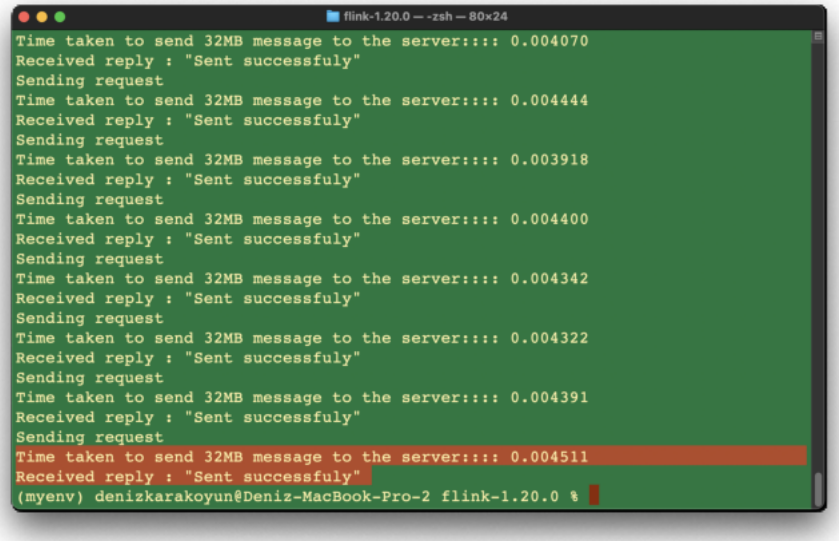


Figure 15: Delivery Time of 32MB Message

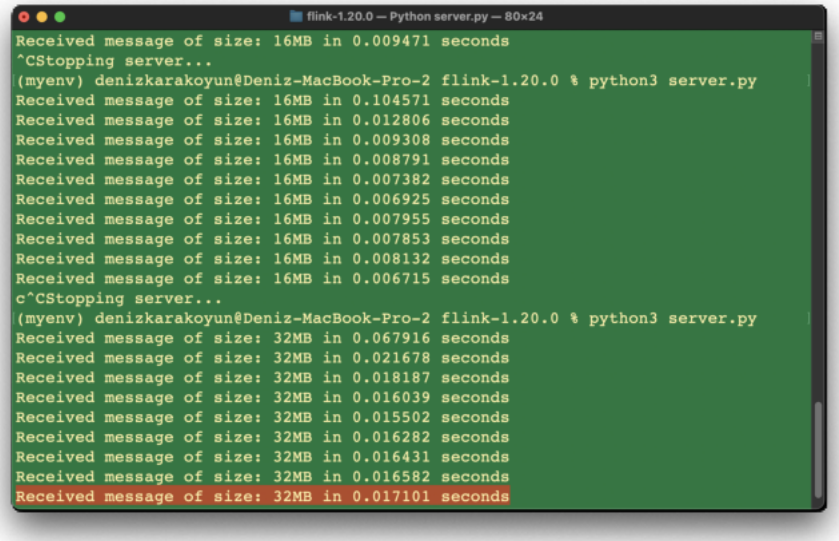


Figure 16: Receive Time of 32MB Message

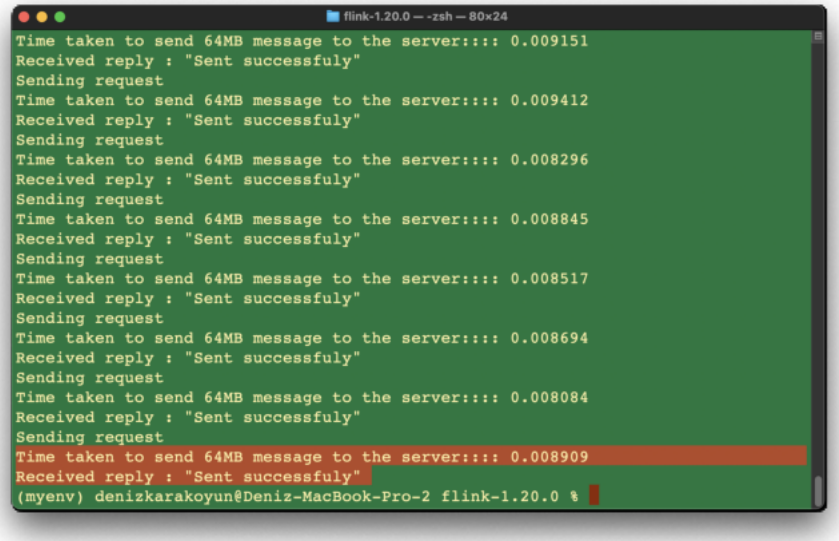


Figure 17: Delivery Time of 64MB Message

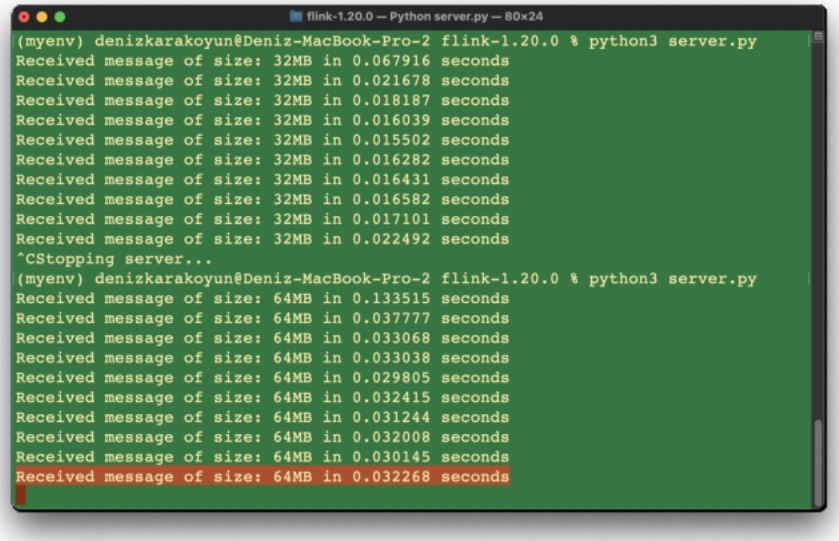


Figure 18: Receive Time of 64MB Message

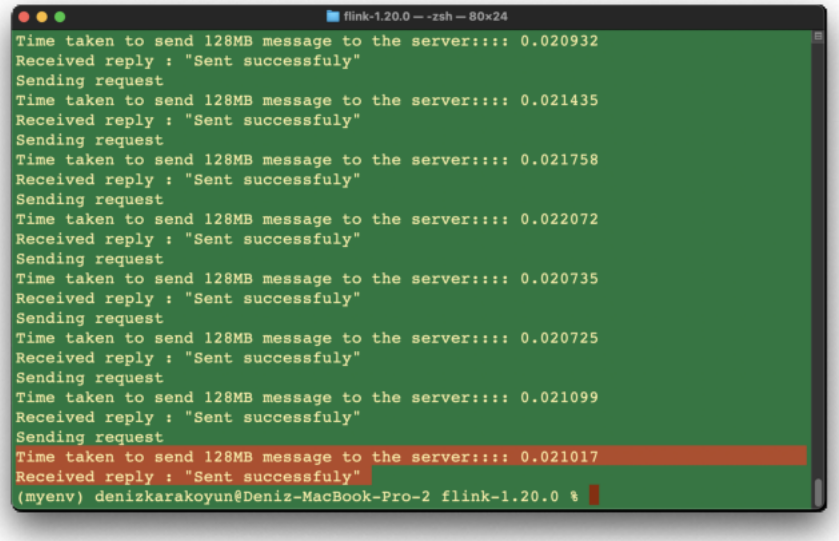


Figure 19: Delivery Time of 128MB Message

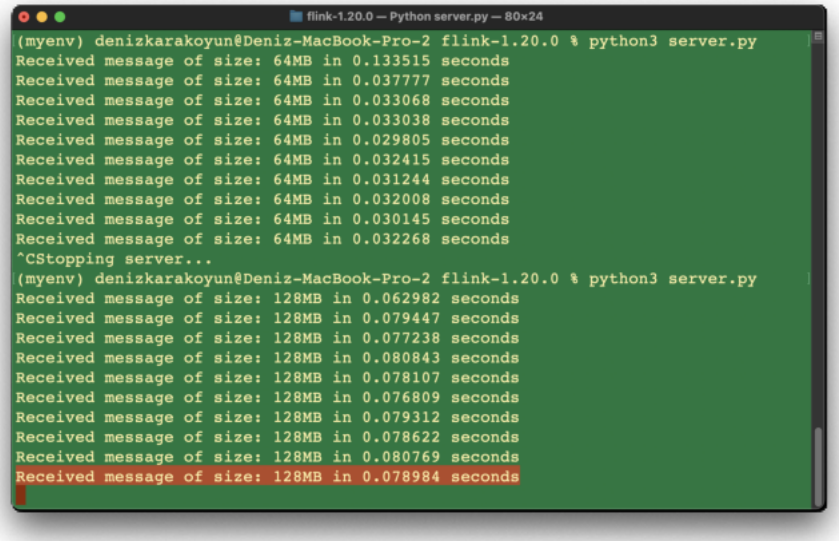


Figure 20: Receive Time of 128MB Message

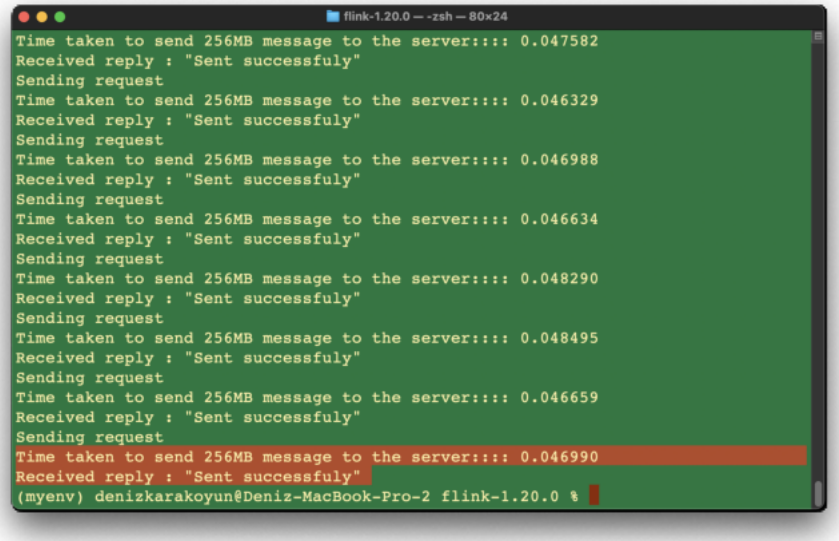


Figure 21: Delivery Time of 256MB Message

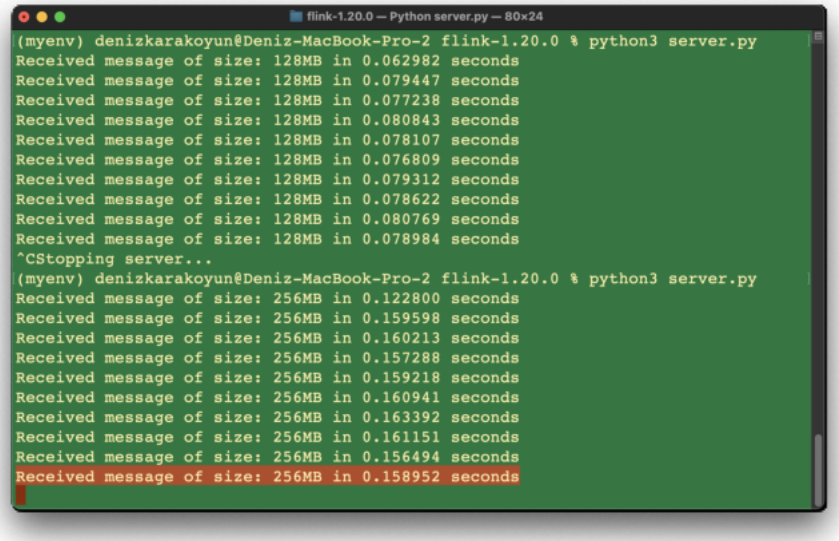


Figure 22: Receive Time of 256MB Message

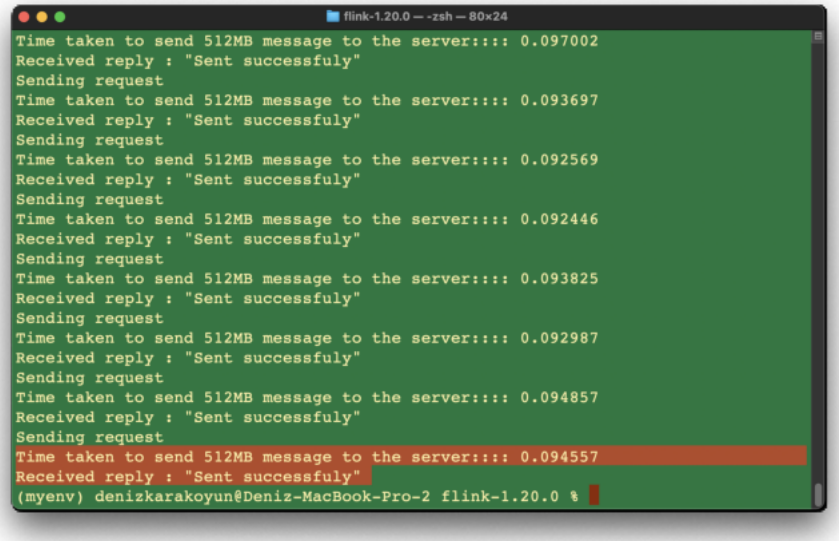


Figure 23: Delivery Time of 512MB Message

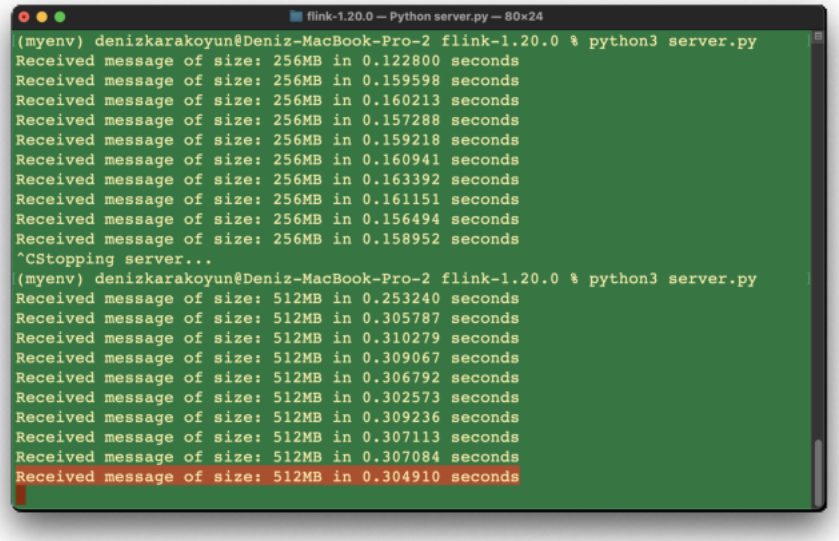


Figure 24: Receive Time of 512MB Message

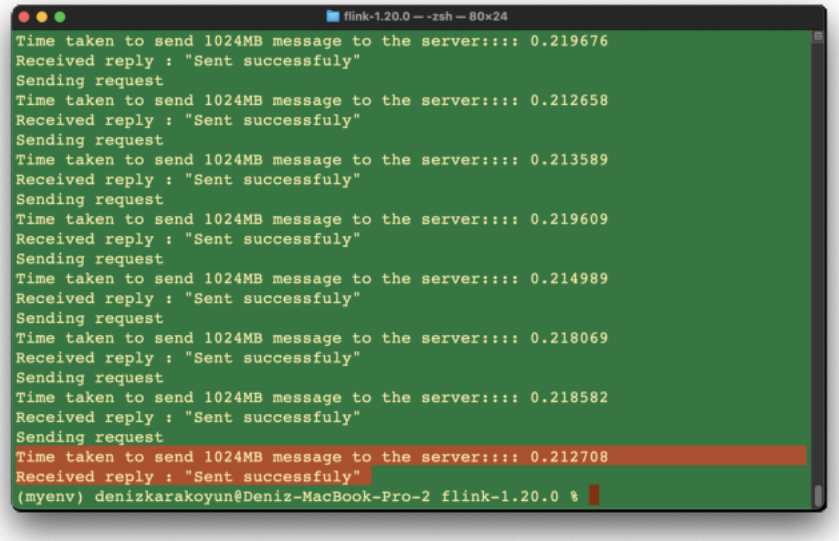


Figure 25: Delivery Time of 1024MB Message

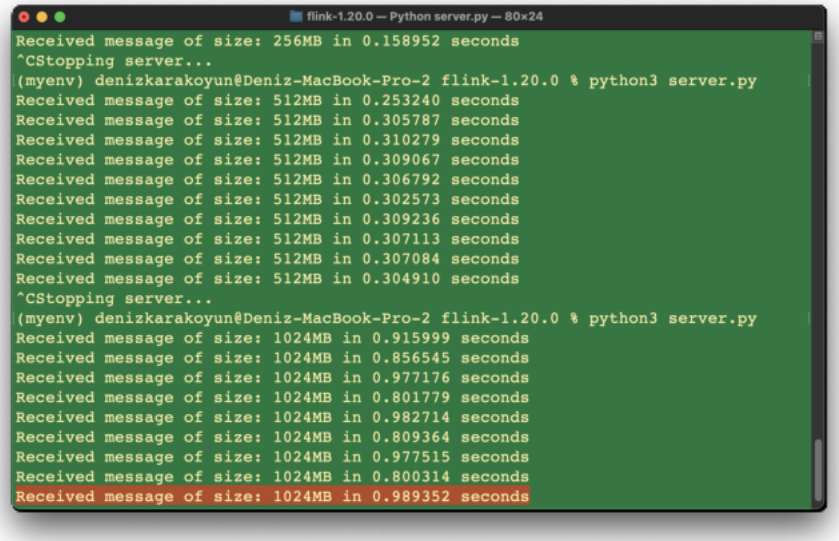


Figure 26: Receive Time of 1024MB Message

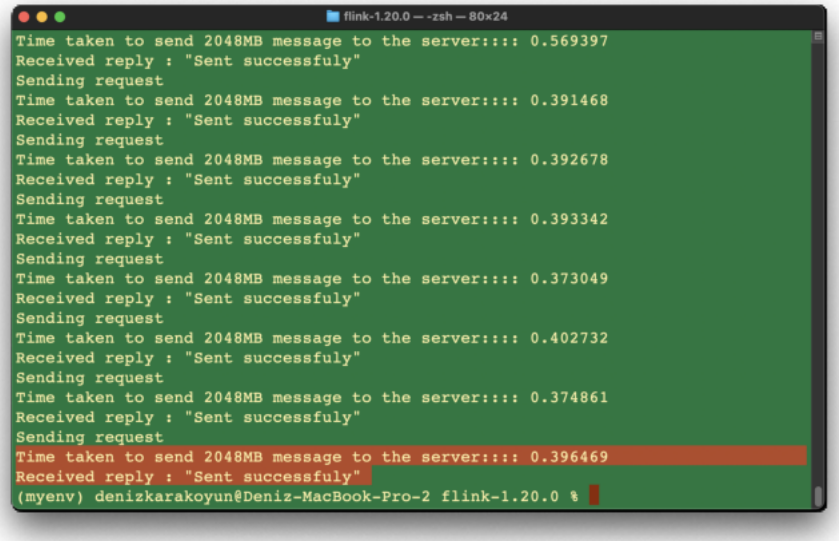


Figure 27: Delivery Time of 2048MB Message

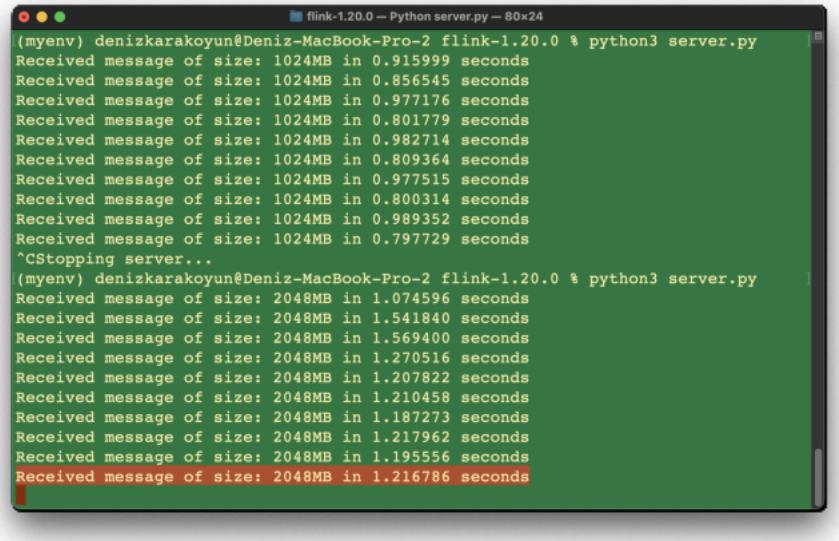


Figure 28: Receive Time of 2048MB Message

10 Introduction to Single Producer Multiple Consumer (ZeroMQ PUB-SUB Pattern)

In ZeroMQ, the **Publish-Subscribe (PUB-SUB)** pattern allows a single producer (or publisher) to send messages to multiple consumers (or subscribers). This pattern is highly useful for broadcasting messages to many receivers in real time. Each consumer that subscribes to a topic receives a copy of the messages sent by the producer for that specific topic.

10.1 Key Concepts

- **Producer (Publisher)**: Sends messages to multiple consumers via a PUB socket.
- **Consumers (Subscribers)**: Subscribe to topics using a SUB socket to receive relevant messages.
- **Loose Coupling**: Messages are sent without acknowledgment from consumers, meaning some messages may be lost if subscribers are not ready.

10.2 How PUB-SUB Works

- The producer broadcasts messages to all consumers subscribed to the ‘topic1‘ topic.
- Multiple consumers can subscribe to the same topic and receive the same message.
- This system is useful for log aggregation, monitoring, or broadcasting updates to multiple receivers.

11 Implementation

11.1 Producer Code (Publisher)

The producer sends messages using a PUB socket. It binds to a specific address and port, and consumers connect to this address to receive messages.

Producer Code

```python
import zmq
import time

def main():
    context = zmq.Context()
    socket = context.socket(zmq.PUB)
    socket.bind("tcp://*:5556")

    print("Publisher started. Waiting for subscribers to connect...")
    time.sleep(1)  # Give some time for subscribers to connect

    message = "h" * (1024 * 1024 * 2)  # 2MB message
    topic = "long_messages"

    for i in range(20):
        print(f"Sending message {i + 1}")
        start = time.time()
        socket.send_string(f"{topic} {message}", copy=False)  # Send message with topic
        end = time.time()
        print(f"Time taken to send {len(message) / (1024 * 1024)}MB message: {end - start:.6f} seconds")
        time.sleep(1)  # Wait a bit before sending the next message

if __name__ == "__main__":
    main()
```

Explanation:

- `context = zmq.Context()`: Creates a ZeroMQ context that manages sockets.
- `socket = context.socket(zmq.PUB)`: Creates a PUB socket that sends messages.
- `socket.bind("tcp://*:5556")`: Binds the producer to port 5556 for communication.
- `socket.send_string()`: Publishes messages along with the topic (‘topic1‘) to all subscribers.
- `time.sleep()`: Introduces a delay between successive messages.

11.2 Consumer Code (Subscriber)

Each consumer subscribes to the topic using a SUB socket and receives messages from the producer. Multiple consumers can connect to the same producer.

Consumer Code

```python
import zmq
import time

def main():
    context = zmq.Context()
    socket = context.socket(zmq.SUB)
    socket.connect("tcp://localhost:5556")

    topic = "long_messages"
    socket.setsockopt_string(zmq.SUBSCRIBE, topic)

    print("Subscriber started. Waiting for messages...")

    while True:
        try:
            start = time.time()  # Start time for receiving the message
            message = socket.recv().decode('utf-8')  # Receive single message
            end = time.time()  # End time after receiving the message

            # Extract topic and actual message
            message_topic, actual_message = message.split(maxsplit=1)
            size = len(actual_message) / (1024 * 1024)  # Calculate size in MB

            if message_topic == topic:
                print(f"Received message of size: {size:.2f}MB in {end - start:.6f} seconds")
            else:
                print(f"Received message with unexpected topic: {message_topic}")

        except KeyboardInterrupt:
            print("Stopping subscriber...")
            break

if __name__ == "__main__":
    main()
```

Explanation:

- `socket = context.socket(zmq.SUB)`: Creates a SUB socket that listens for messages.
- `socket.connect()`: Connects to the producer's address (‘tcp://localhost:5556‘).
- `socket.setsockopt_string(zmq.SUBSCRIBE, topic)`: Subscribes to a specific topic (‘topic1‘).
- `socket.recv_string()`: Receives the message sent by the producer and prints it.

12 Timing Results

Below is the timing result table, with the message size, delivery time (from client to server), and receive time (from server to client).

Message Size (MB)	**Delivery Time (seconds)**	**Receive Time 1 (seconds)**	**Receive Time 2 (seconds)**
1	0.000559	1.005368	1.006101
2	0.001790	1.004067	1.006090
4	0.002889	1.004235	1.009415
8	0.002089	0.995442	1.006794
16	0.004188	0.985381	1.004705
32	0.017802	0.995953	1.022613
64	0.021653	0.994809	1.019397
128	0.049141	0.973699	1.045973
256	0.072796	0.950010	1.026559
512	0.132825	0.897066	1.032544
1024	0.305721	0.848178	1.407824
2048	0.465221	1.270312	1.112928

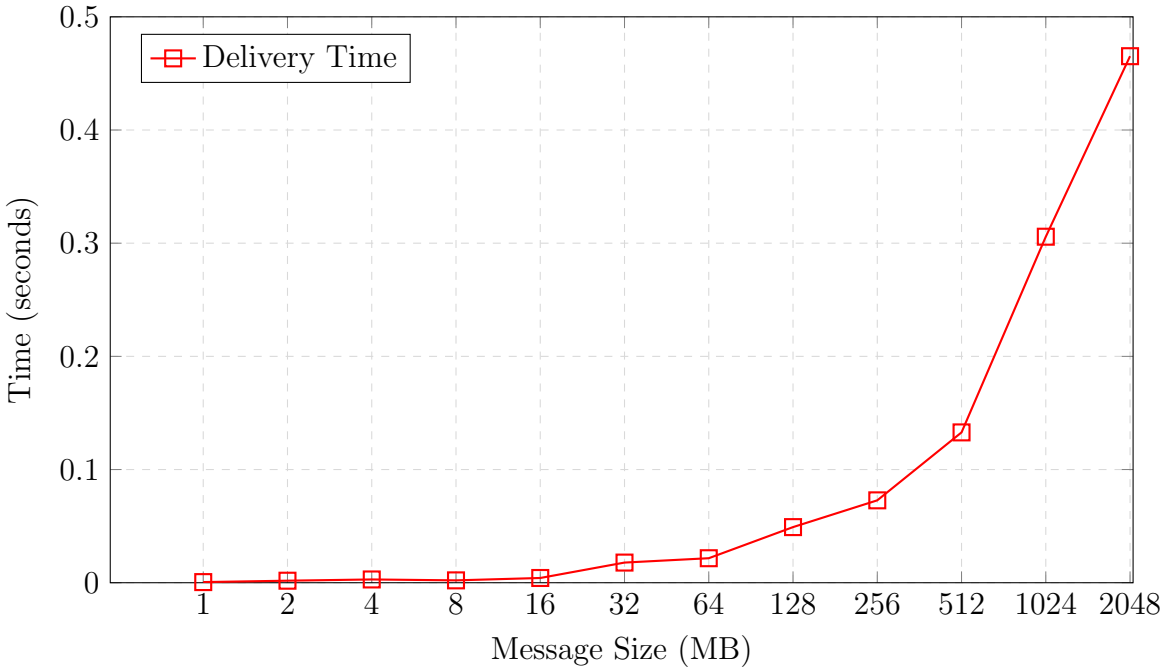


Figure 29: Message Size vs Delivery Time

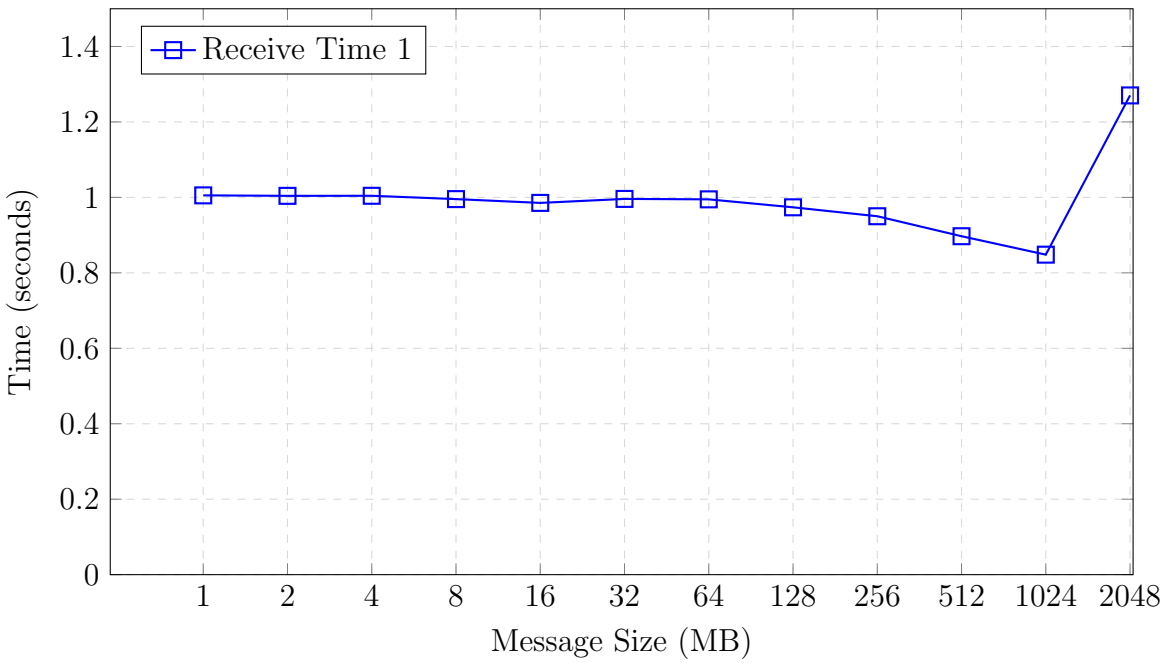


Figure 30: Message Size vs Receive Time 1

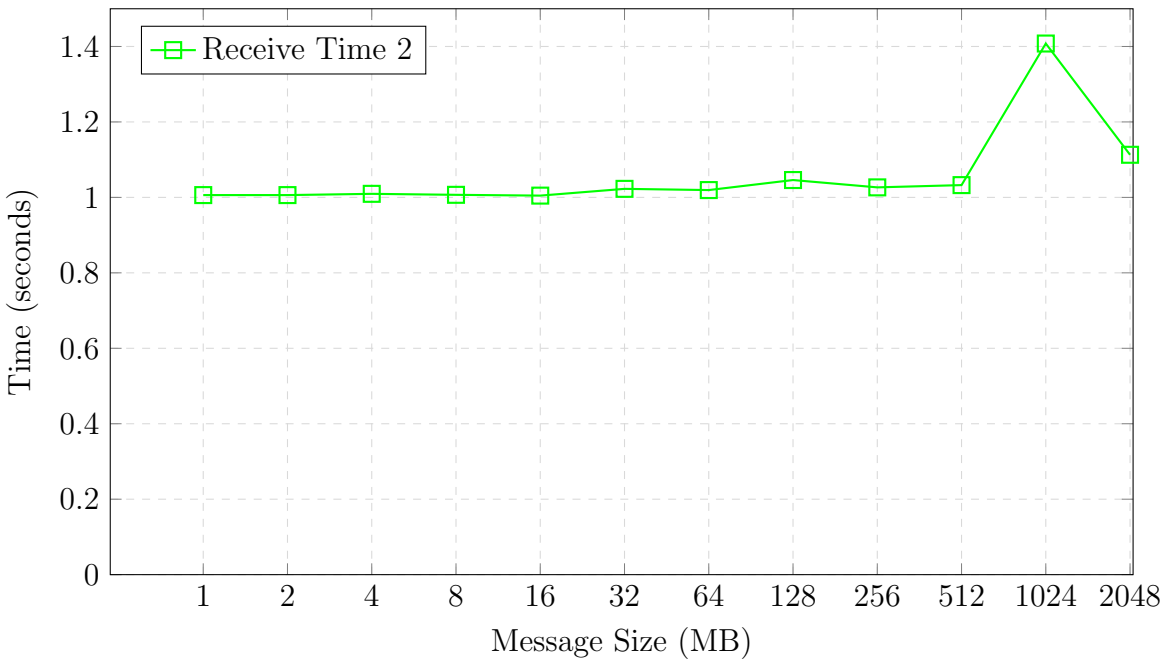


Figure 31: Message Size vs Receive Time 2

13 Conclusion

As observed from the timing results, the delivery time generally increases with the message size. This trend is expected, as larger message sizes require more time to transmit.

For both receive times (1 and 2), we observe a relatively stable period for smaller message sizes. However, as the message size approaches 1024MB, there is a noticeable decrease in the number of successfully received messages, particularly for receive time 2, where the time jumps significantly for the 1024MB message. This suggests that as message sizes grow larger, the system's ability to maintain consistent performance diminishes, potentially due to resource constraints or network limitations.

At 1024MB, the receive time becomes more variable, indicating potential buffering or packet loss issues, as seen in the jump in receive times for both 1024MB and 2048MB messages.

14 Datas

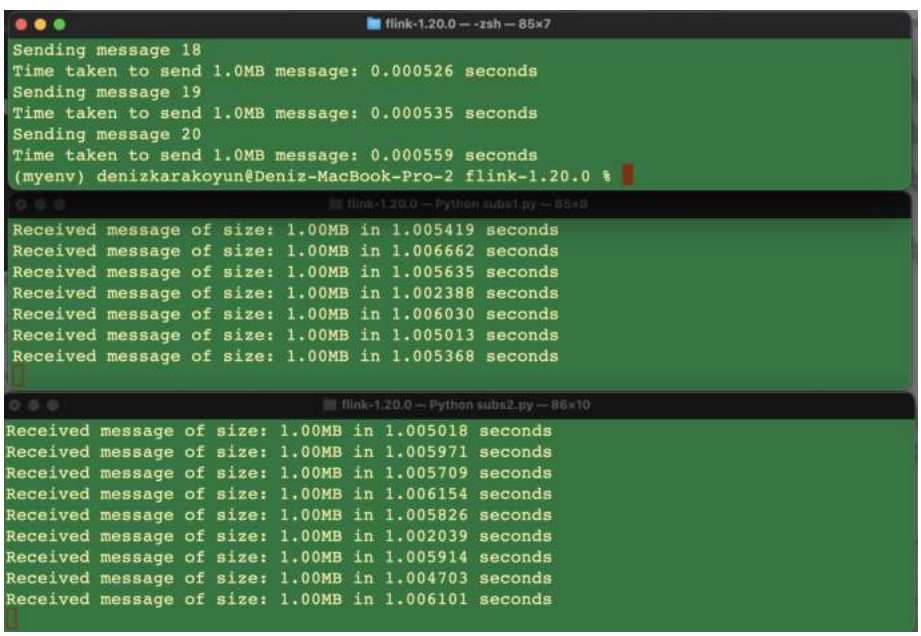


Figure 32: Timeline of 1MB Message

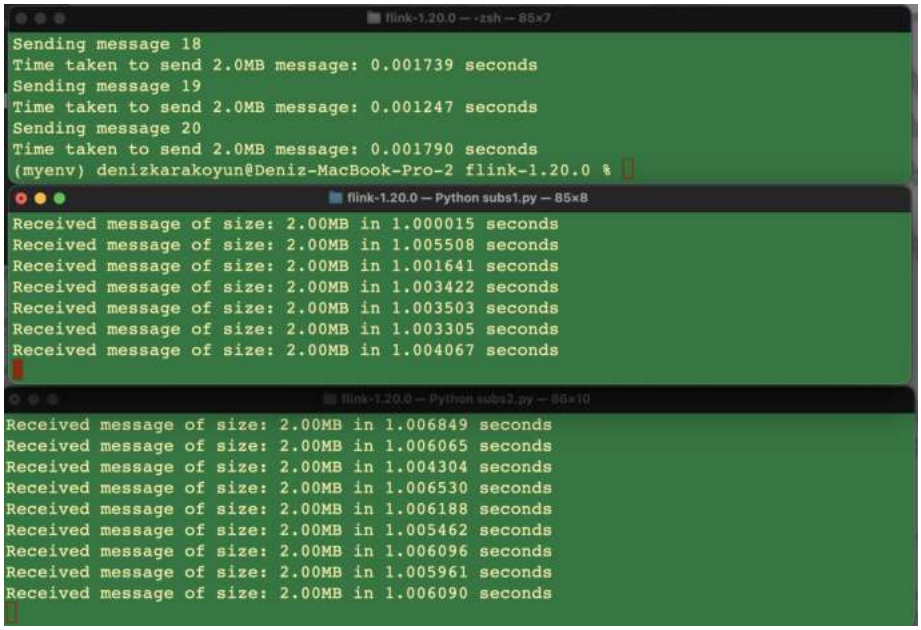


Figure 33: RTimeline of 2MB Message

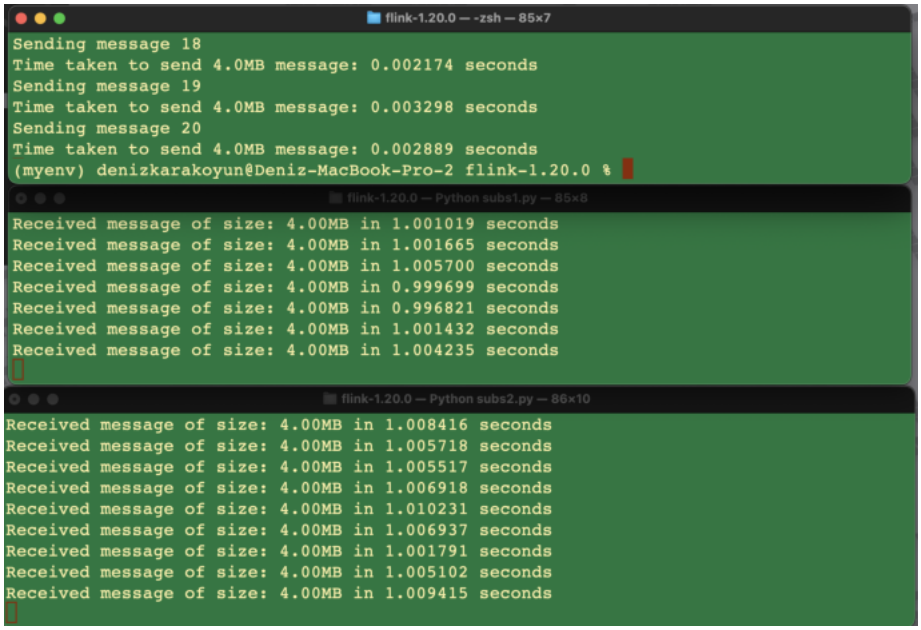


Figure 34: Timeline of 4MB Message

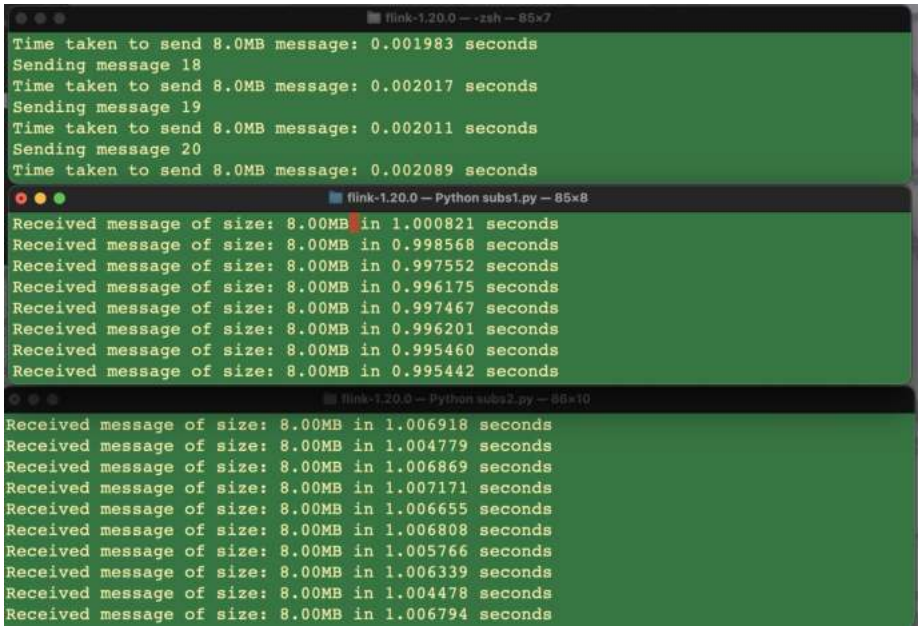


Figure 35: Timeline of 8MB Message

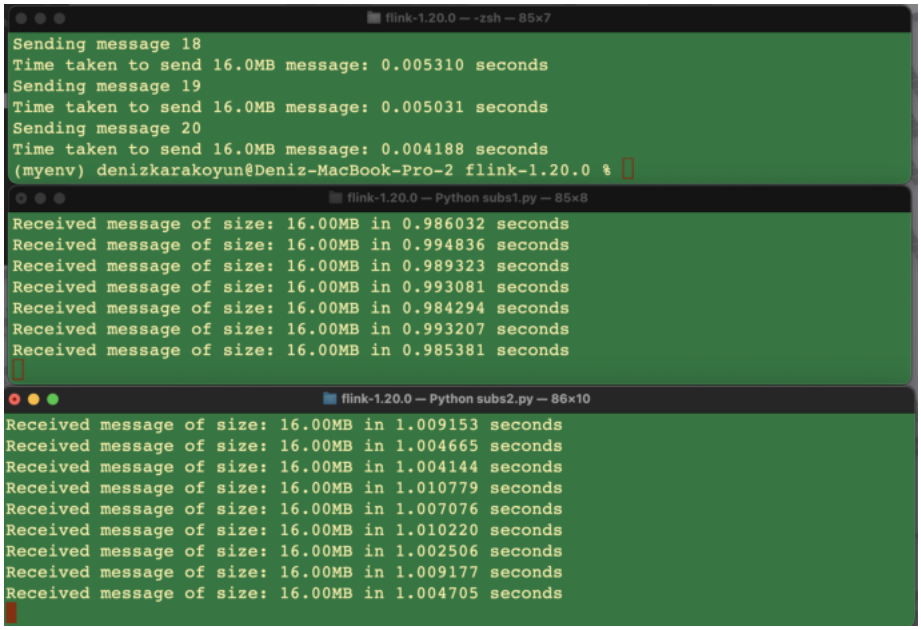


Figure 36: Timeline of 16MB Message

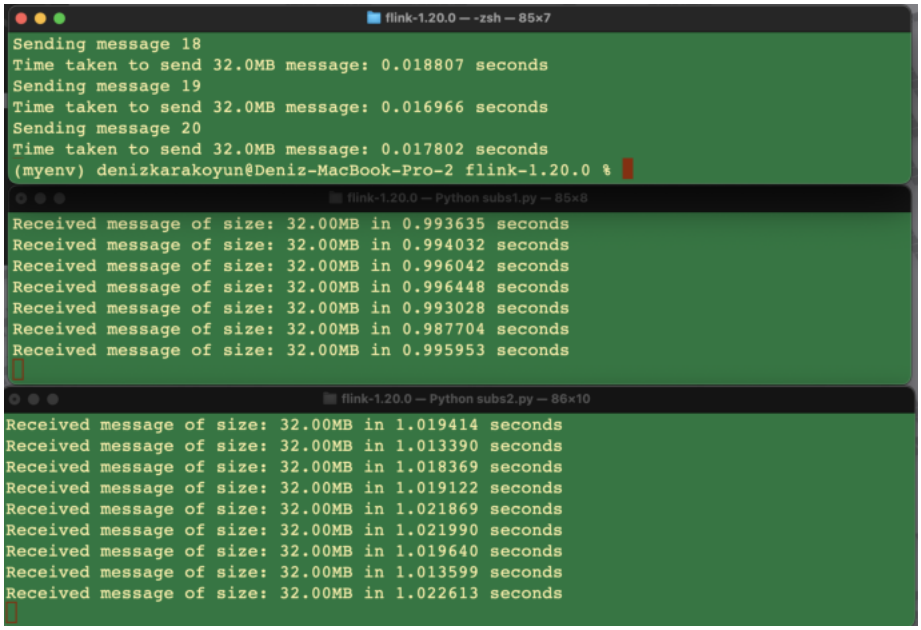


Figure 37: Timeline of 32MB Message

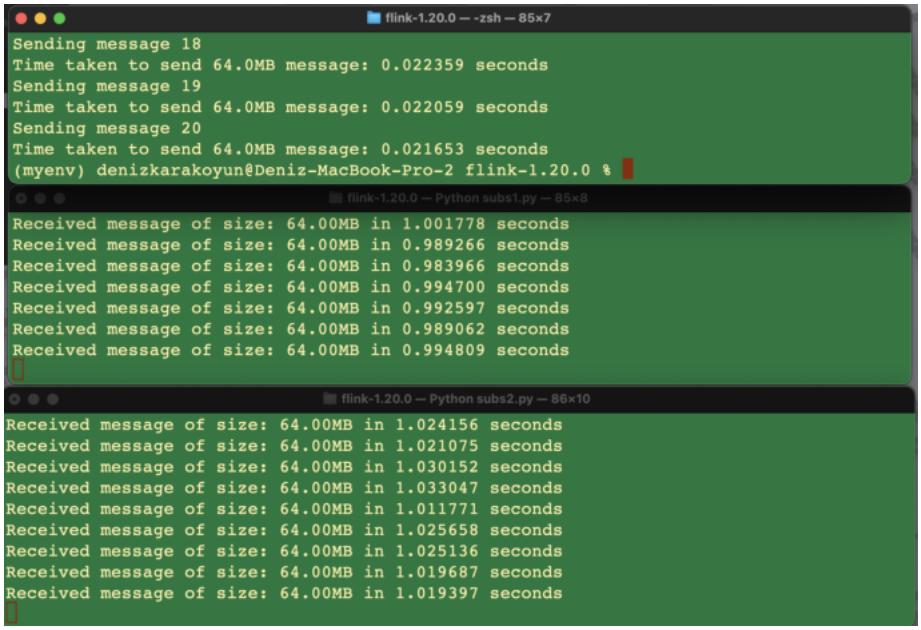


Figure 38: Timeline of 64MB Message

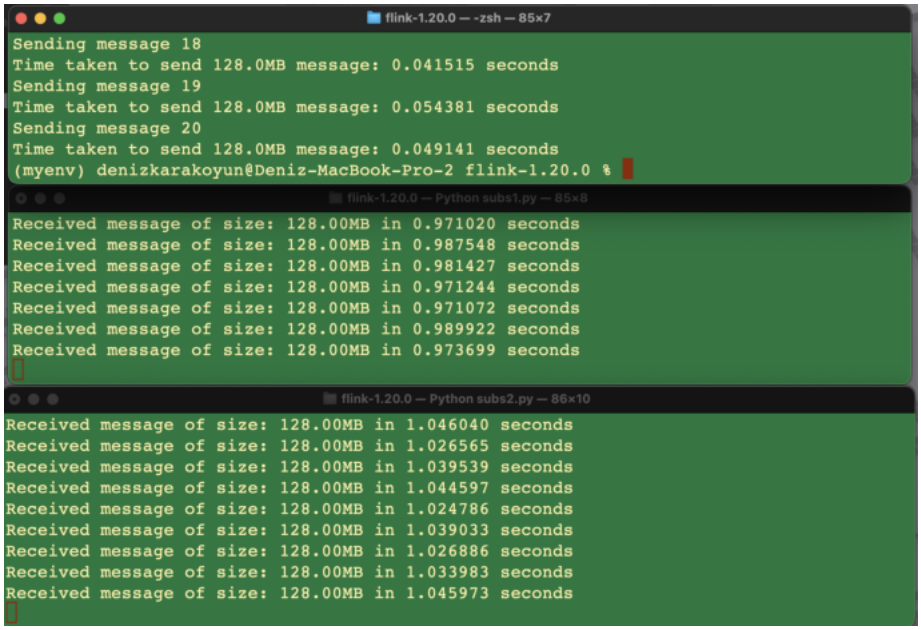


Figure 39: Timeline of 128MB Message

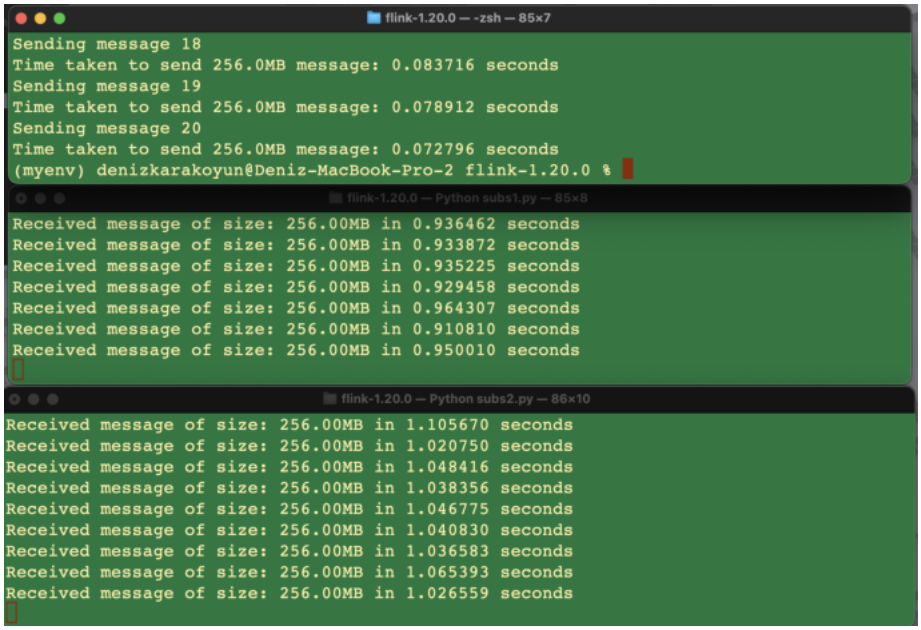


Figure 40: Timeline of 256MB Message

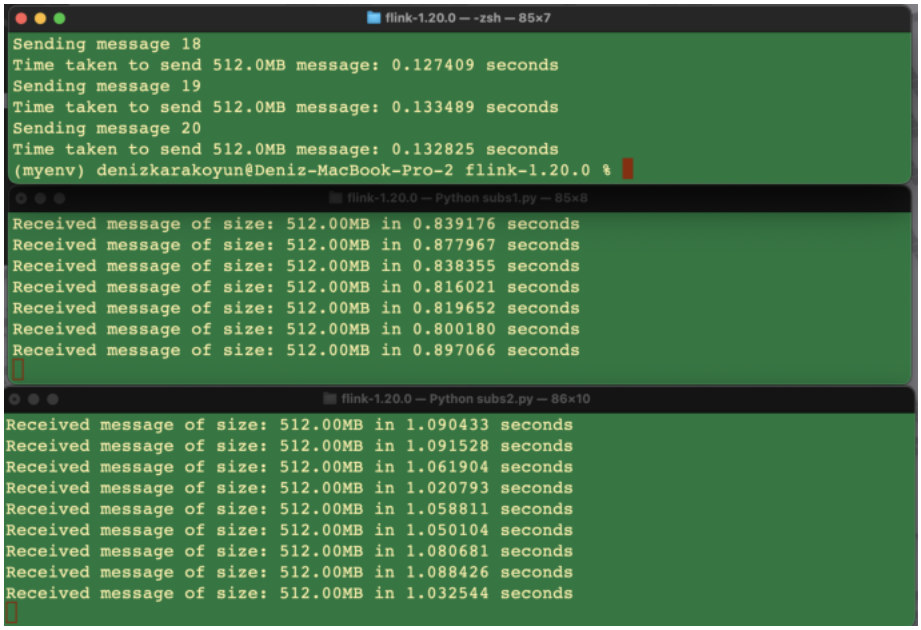


Figure 41: Timeline of 512MB Message

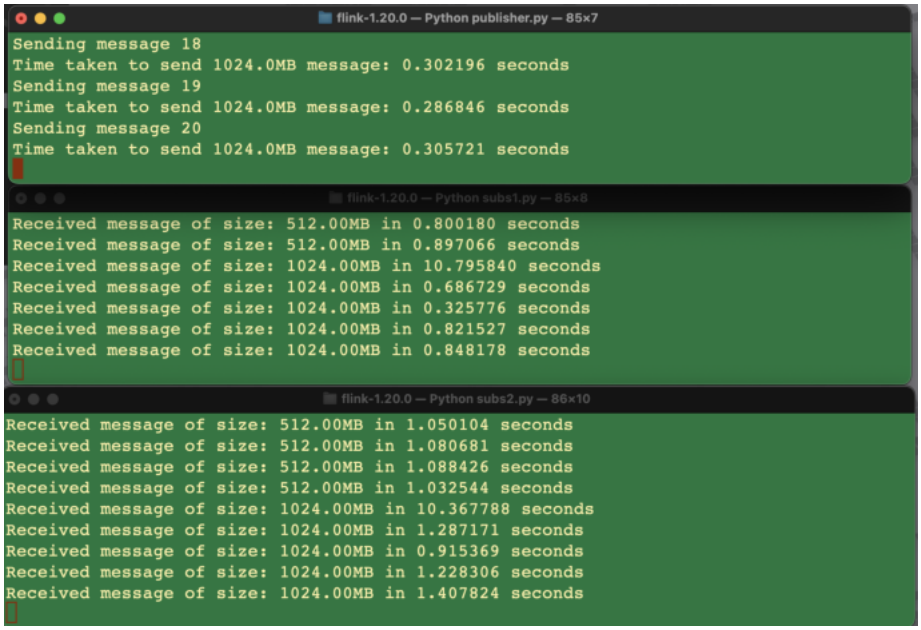


Figure 42: Timeline of 1024MB Message

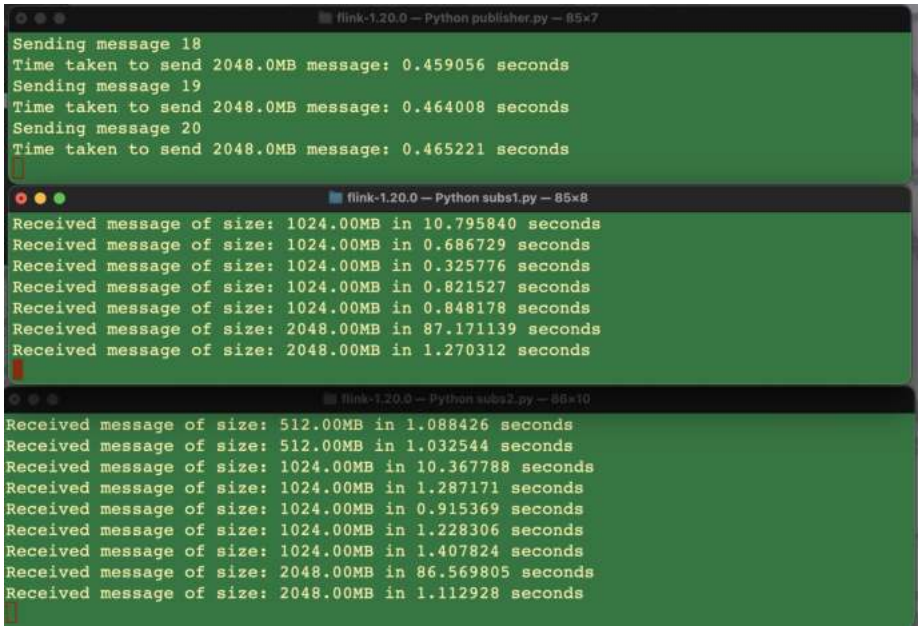


Figure 43: Timeline of 2048MB Message

Multiple Consumer Single Producer Messaging in ZeroMQ

In this architecture, a single producer sends messages to multiple consumers. This setup allows for load balancing and concurrent processing of messages, enhancing throughput and efficiency.

Key Concepts

1. **Single Producer:** The entity that generates and sends messages. This could be any application that produces data for processing, such as a logging service, data generator, or event source.

2. **Multiple Consumers:** These entities receive messages from the producer. Each consumer processes messages independently, allowing for parallel processing. This means that while one consumer processes a message, others can simultaneously process different messages.

3. **Load Balancing:** ZeroMQ uses the `PUSH` and `PULL` patterns to distribute mes-

sages. The producer sends messages to a PUSH socket, which then distributes them to connected PULL sockets (consumers). This balances the load among consumers, preventing any single consumer from being overwhelmed.

4. **Message Distribution:** When the producer sends a message, ZeroMQ routes it to one of the available consumers based on its load-balancing strategy. This can be round-robin or another algorithm that ensures even distribution.

5. **Asynchronous Processing:** Consumers can process messages asynchronously, allowing them to start processing a new message before finishing the previous one. This increases the overall throughput of the system.

Example Code

Producer 1 Code

```
import zmq
import time

def main():
    context = zmq.Context()
    socket = context.socket(zmq.PUSH)
    socket.connect("tcp://localhost:5557") # Connect to consumer

    for i in range(20):
        message1 = "h" * (1024 * 1024 * 2048) # Create a 2GB message
        message = f"{message1} 1" # Append "1" to the message
        start = time.time() # Start timer
        socket.send_string(message) # Send message to the consumer
        end = time.time() # End timer
        print(f"Producer 1 sending {(len(message)-2)/(1024*1024)}MB message
            in {end-start:.6f} seconds")

if __name__ == "__main__":
    main()
```

Producer 2 Code

```
import zmq
import time
```

```python
def main():
    context = zmq.Context()
    socket = context.socket(zmq.PUSH)
    socket.connect("tcp://localhost:5557") # Connect to consumer

    for i in range(20):
        message1 = "h" * (1024 * 1024 * 2048) # Create a 2GB message
        message = f"{message1} 1" # Append "1" to the message
        start = time.time() # Start timer
        socket.send_string(message) # Send message to the consumer
        end = time.time() # End timer
        print(f"Producer 2 sending {(len(message)-2)/(1024*1024)}MB message
            in {end-start:.6f} seconds")

if __name__ == "__main__":
    main()
```

Consumer Code

```python
import zmq
import time

def main():
    context = zmq.Context()
    socket = context.socket(zmq.PULL)
    socket.bind("tcp://*:5557") # Bind to receive messages from producers

    print("Consumer started. Waiting for messages...")

    while True:
        try:
            start = time.time() # Start timer
            message, number = socket.recv_string().split() # Receive message
            end = time.time() # End timer
            print(f"Consumer received from Producer{number} {len(message)
                /(1024*1024)}MB message in {end - start:.6f} seconds")
        except KeyboardInterrupt:
            print("Stopping consumer...")
            break
```

```python
if __name__ == "__main__":
    main()
```

Summary

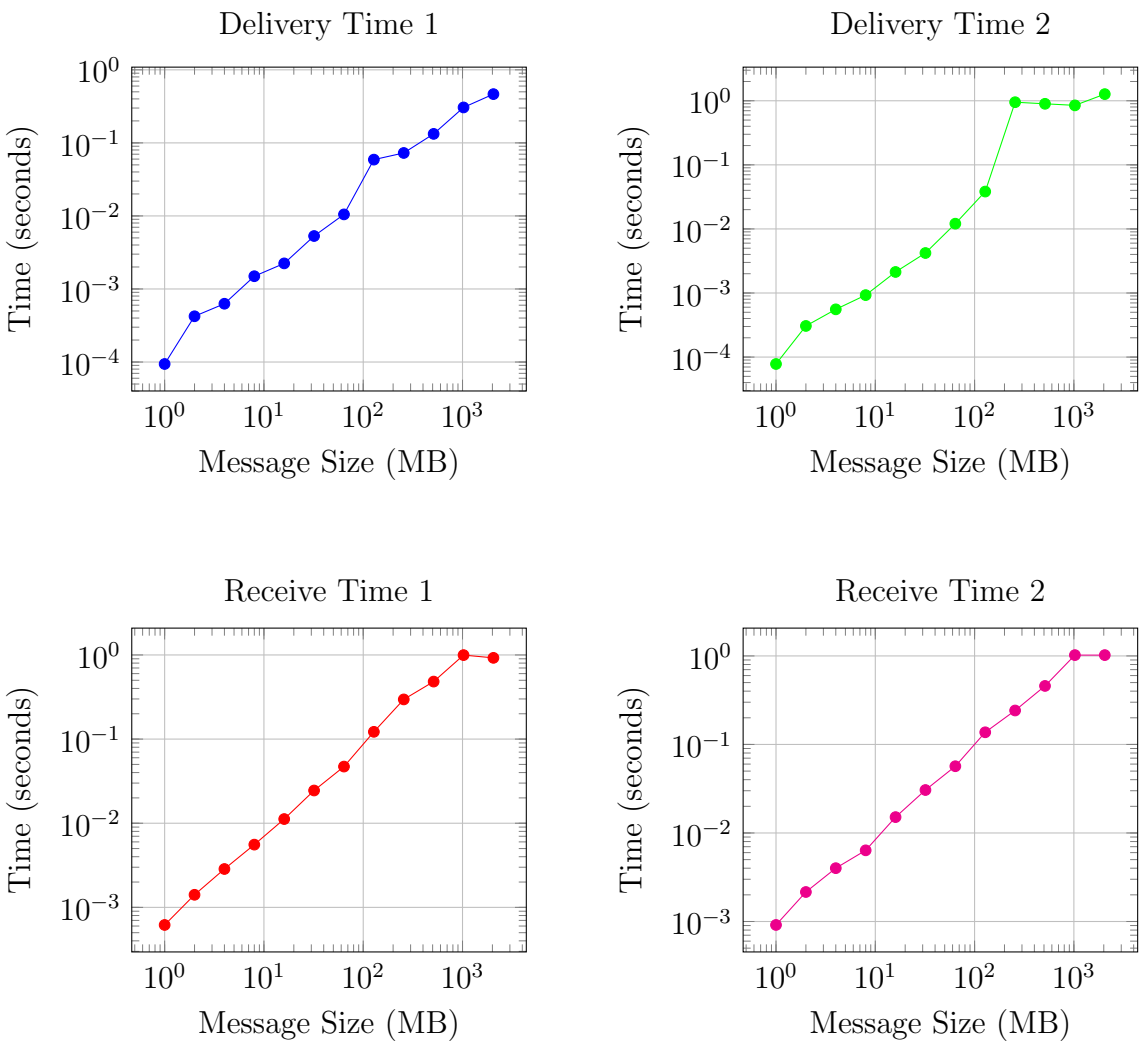
This setup demonstrates a typical scenario where a single producer generates large messages and distributes them to multiple consumers using ZeroMQ. The architecture efficiently handles message distribution, allowing for parallel processing and load balancing among consumers. By running multiple producer instances, you can see how ZeroMQ handles the load and ensures that messages are processed in a balanced manner across consumers.

Message Size (MB)	Delivery Time 1 (seconds)	Delivery Time 2 (seconds)	Receive Time 1 (seconds)	Receive Time 2 (seconds)
1	0.000094	0.000078	0.000618	0.000914
2	0. 000423	0.000306	0.001411	0.002154
4	0. 000628	0.000556	0.002858	0.004003
8	0.001495	0.000927	0.005567	0.006367
16	0.002240	0.002129	0.011214	0. 015109
32	0.005310	0.004208	0.024491	0. 030561
64	0.010499	0.012074	0.047140	0.056744
128	0. 059005	0. 038298	0.122018	0.137208
256	0.072796	0.950010	0.296167	0.241835
512	0.132825	0.897066	0.482487	0.458126
1024	0.305721	0.848178	0.996039	1.021796
2048	0.465221	1.270312	0.923809	1.021796

Conclusion

The analysis of the delivery and receive times as a function of message size reveals significant trends in the performance of the messaging system. As the message size increases, both delivery and receive times exhibit a clear upward trajectory, demonstrating a direct correlation between message size and latency. Notably, the delivery time for larger messages, particularly beyond 64 MB, escalates dramatically, indicating potential bottlenecks in processing and transmission. The variation between the two delivery time measurements suggests that different configurations or conditions may impact performance, highlighting the importance of system tuning. Similarly, the receive times reflect a sharp increase at higher message sizes, reinforcing the need for optimizations to handle large data efficiently. Overall, these results emphasize the critical relationship between message size and performance metrics, guiding future enhancements to improve throughput and reduce latency in high-volume messaging scenarios.

15 Datas


Delivery Time 1
Time (seconds)
Message Size (MB)
Delivery Time 2
Time (seconds)
Message Size (MB)
Receive Time 1
Time (seconds)
Message Size (MB)
Receive Time 2
Time (seconds)
Message Size (MB)

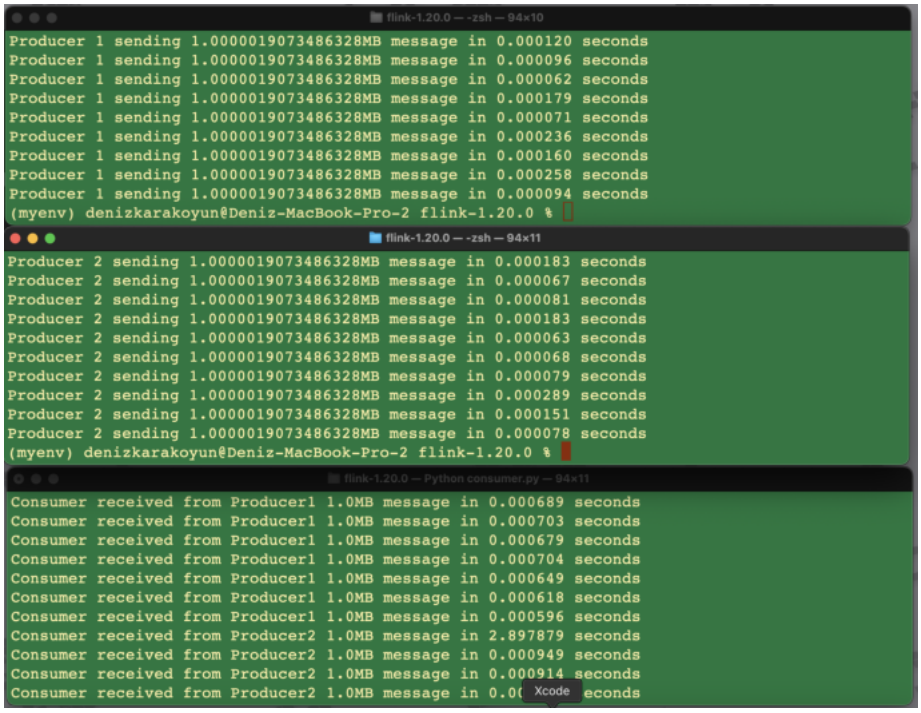


Figure 44: Timeline of 1MB Message

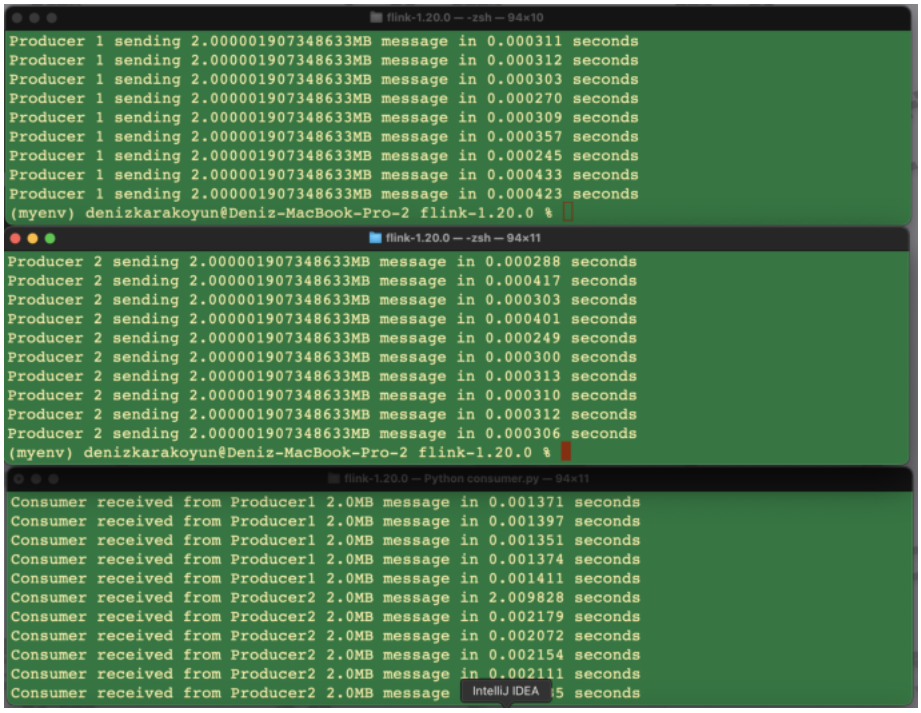


Figure 45: RTimeline of 2MB Message

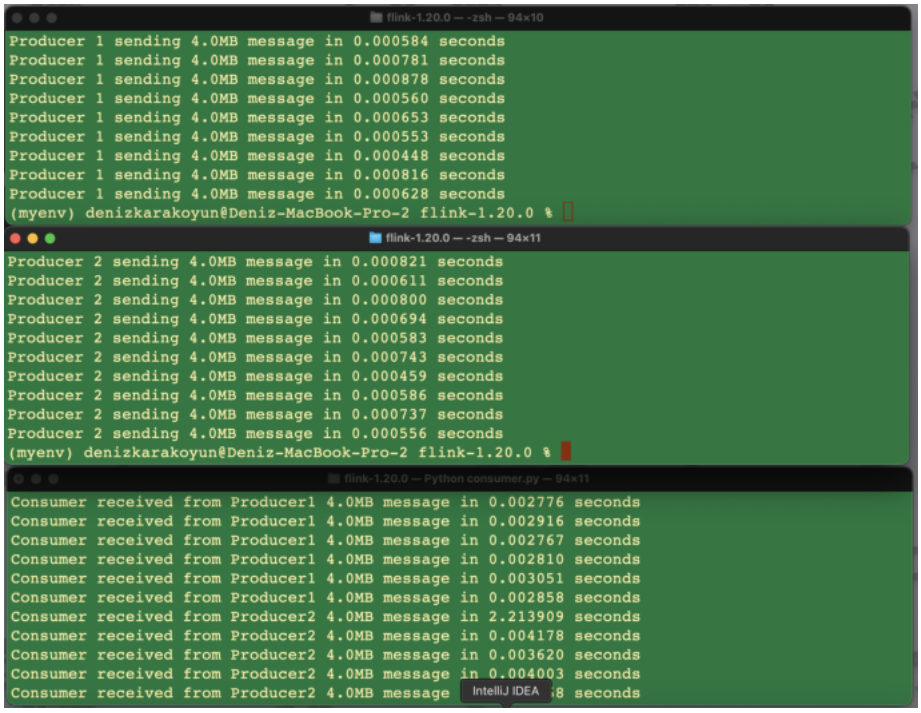


Figure 46: Timeline of 4MB Message

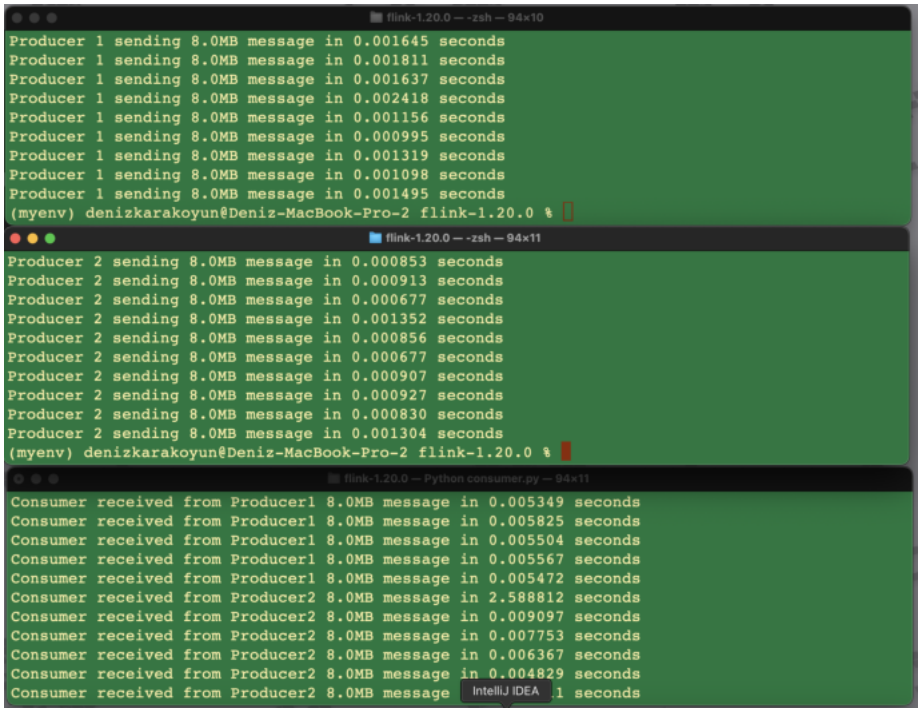


Figure 47: Timeline of 8MB Message

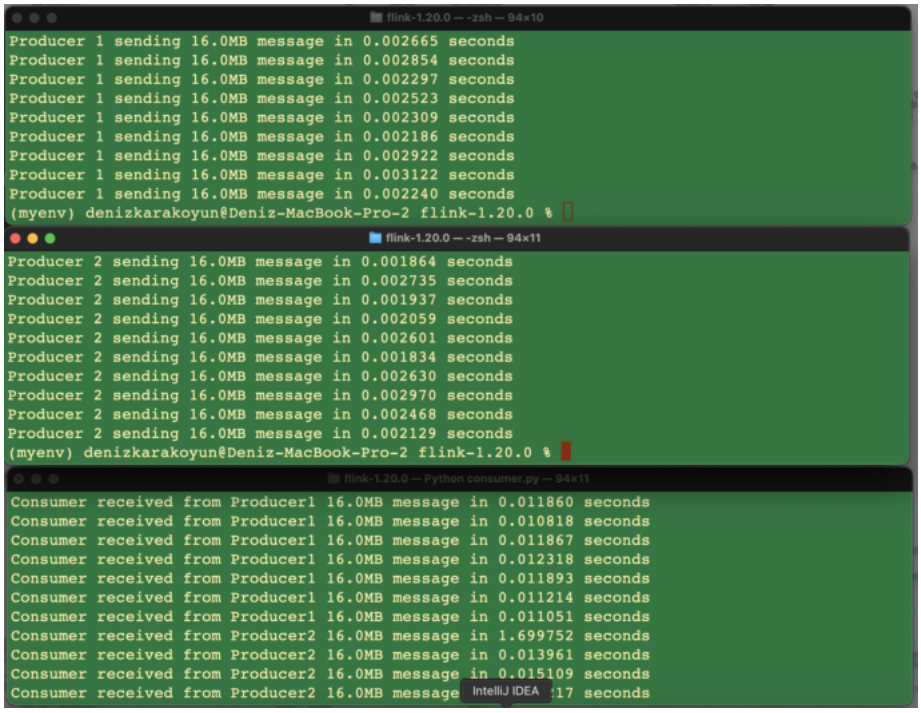


Figure 48: Timeline of 16MB Message

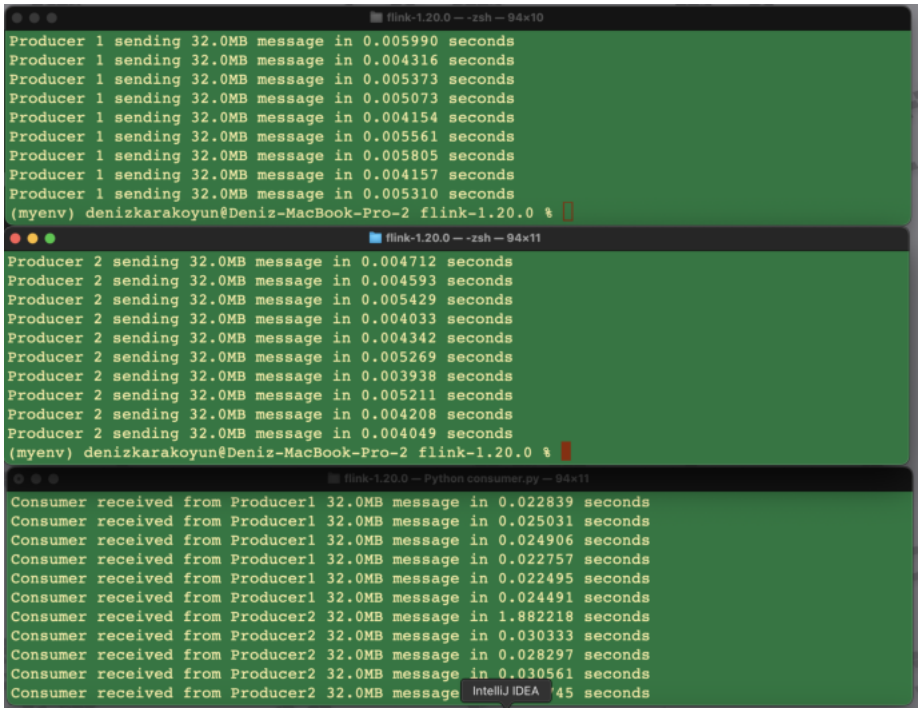


Figure 49: Timeline of 32MB Message

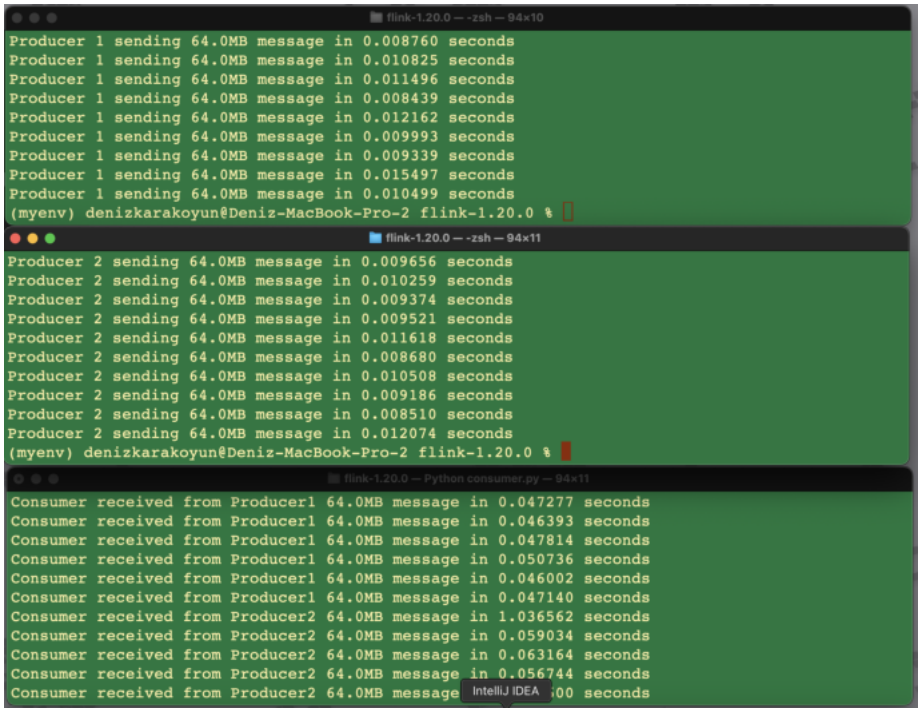


Figure 50: Timeline of 64MB Message

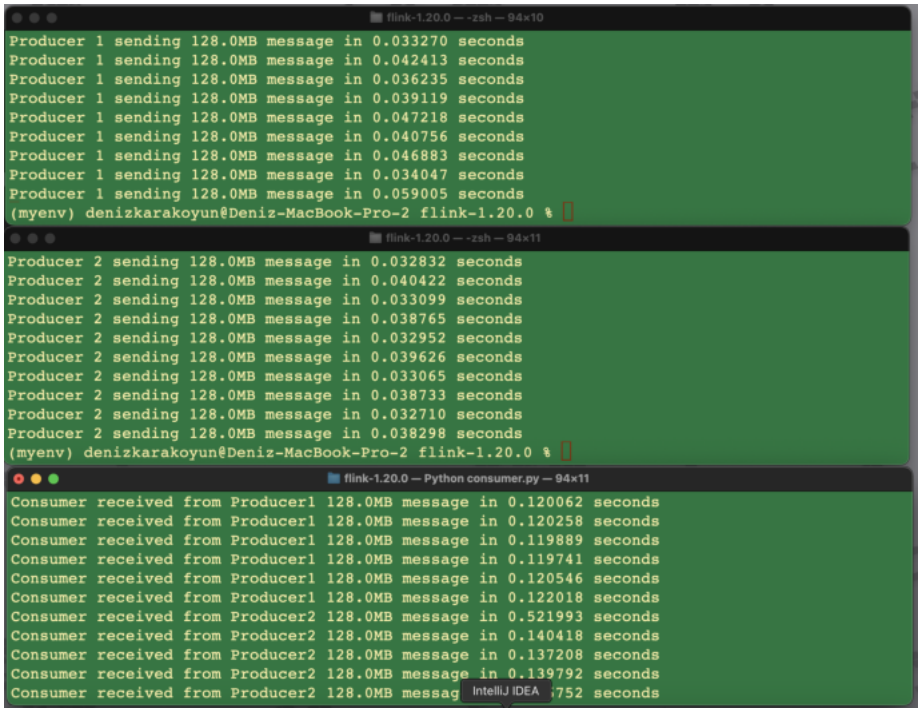


Figure 51: Timeline of 128MB Message

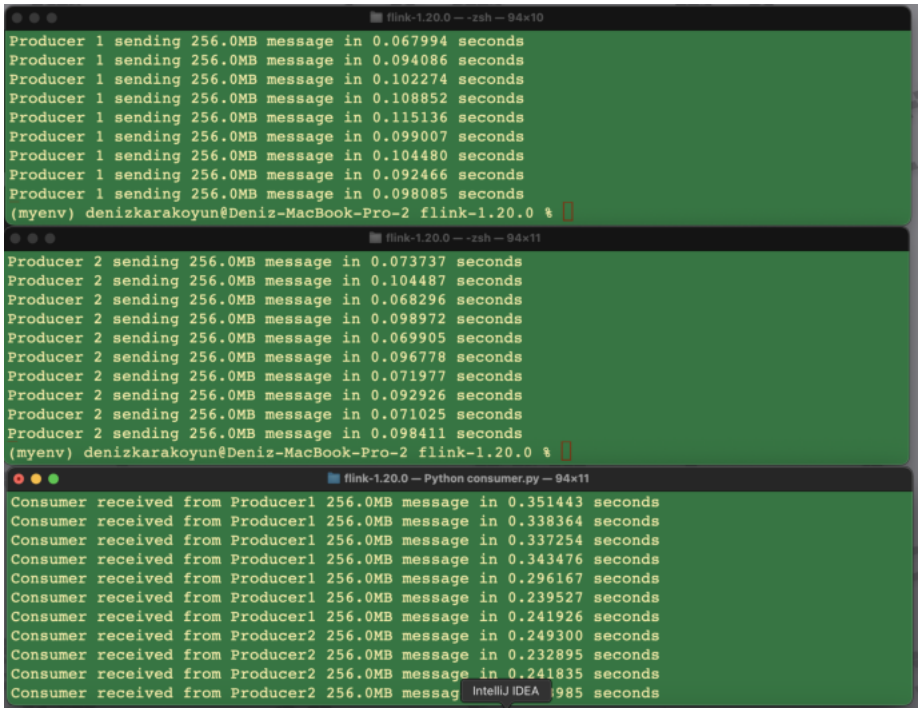


Figure 52: Timeline of 256MB Message

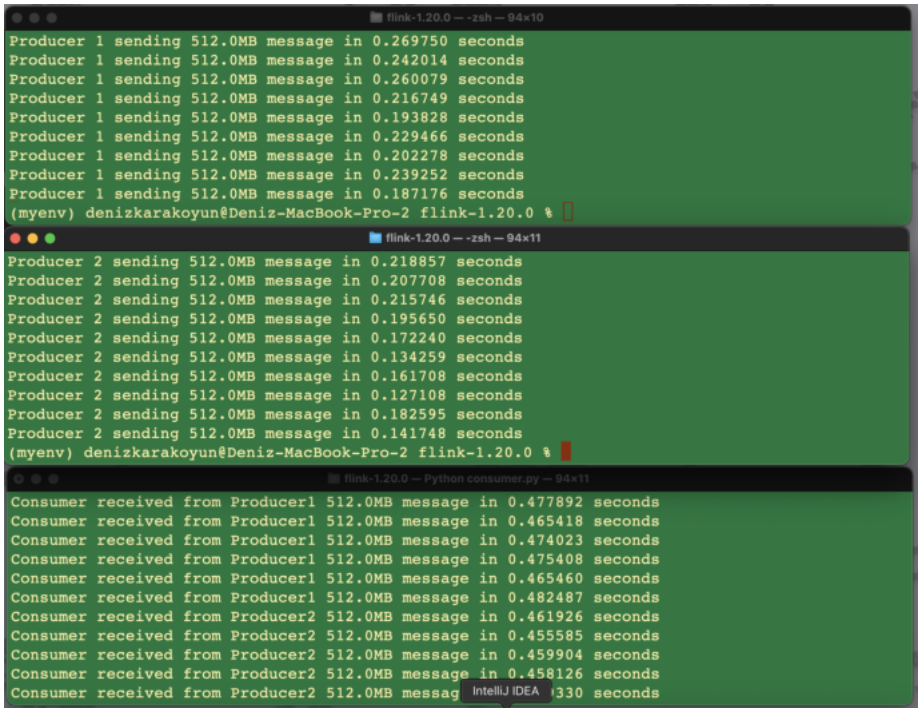


Figure 53: Timeline of 512MB Message

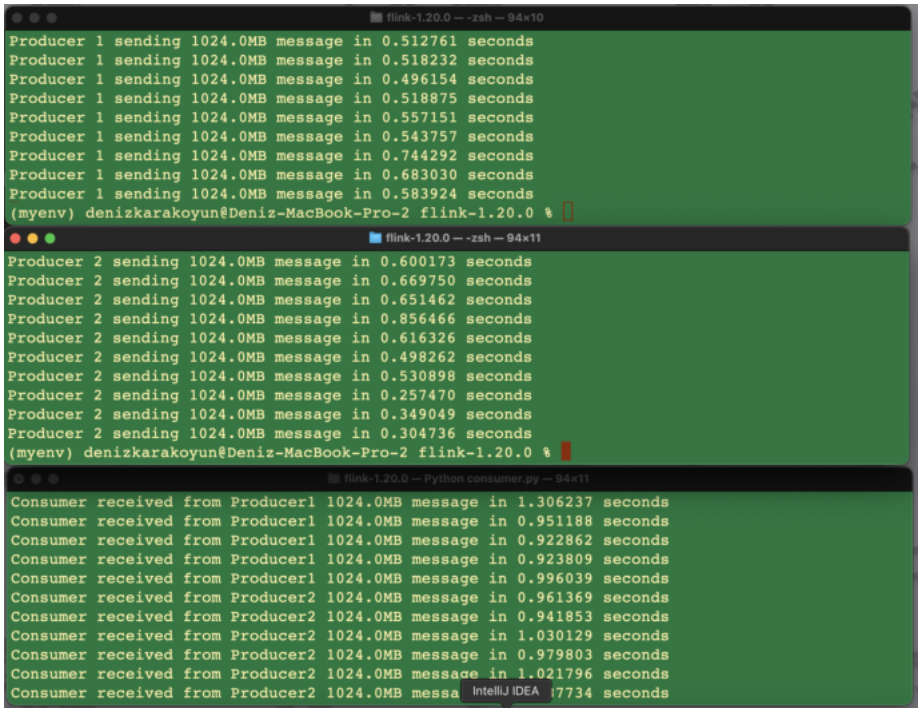


Figure 54: Timeline of 1024MB Message

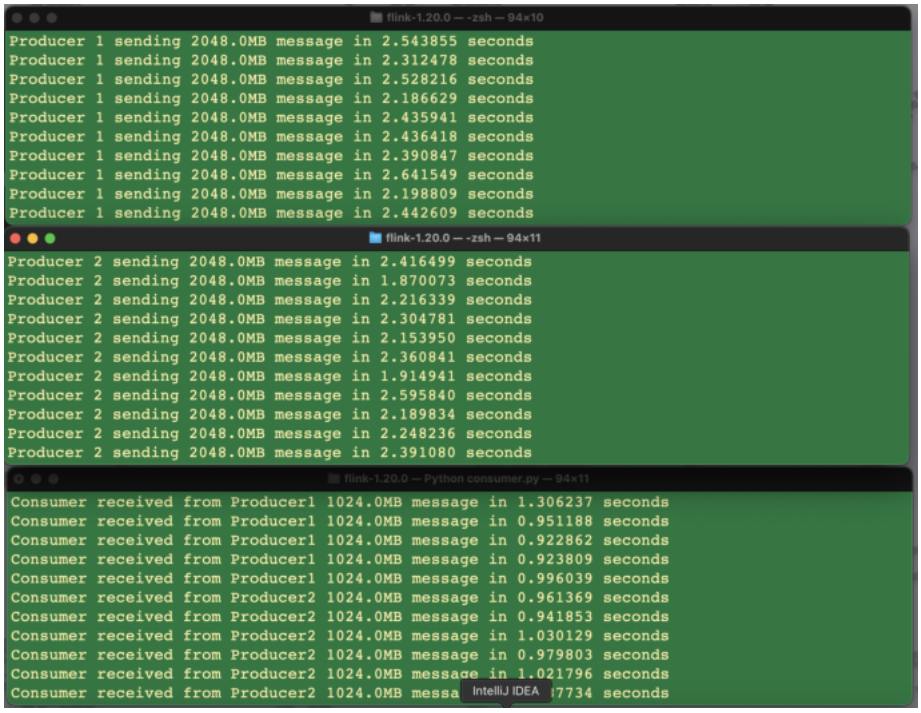


Figure 55: Timeline of 2048MB Message