# Documentation

## Client

* Connects to the Load Balancer via TCP on port 5059
* Sends and retrieves data to the Load Balancer via the Sender Thread
* Waits for responses via the Receiver Thread

**Threads**:

* Main thread
* Data receiver thread
* Data sender thread
* Input thread

## Load Balancer

* Listens for client connections via the Client Listener Thread
* When a new client has connected borrows a Client Data Receiver Thread from the Client Thread Pool
* Publishes the clients request to a blocking Request Queue
* Worker-Client Request Dispatcher Thread subscribes to the Request Queue and when a new request appears it gets the next Worker (Round Robin algorithm) from the Worker List and sends the data to the Worker
* When a new Worker connects the Worker Listener thread registers it by adding it to the Worker List and borrows a Worker Listener Thread from the Worker Thread Pool
* When a new Worker is registered all other Workers are notified so they can open a TCP connection to the newly connected Worker, one of the notified workers also receives a request for full data export to the newly connected Worker
* When a Worker disconnects the Worker Manager thread unregisters it by removing it from the Worker List
* The Worker Data Receiver thread waits for Worker notifications and when it receives a notification from a Worker it puts it to a blocking Response Queue where the Client-Worker Response Dispatcher Thread delivers the response to the Client

**Threads**:

* Main thread
* Client listener thread
* N \* Client data receiver thread
* Worker-client request dispatcher thread
* Worker listener thread
* N \* Worker data receiver thread
* Client-worker response dispatcher thread
* Input thread

**Data structures**:

* Blocking queue for Client requests and responses (like a circular buffer but instead of overwriting data when it overflows, it blocks)
* Circular doubly linked list for Workers with a Round Robin pointer (so we can easily add/remove Workers)
* Thread pools for Client and Worker data receiver threads

## Worker

* Connects to the Load Balancer and listens for data storage requests via the Receiver Thread
* When data is received on the Receiver Thread it inserts the data into a HashMap
* The Receiver Thread sends a notification to the Load Balancer when the data is stored and also broadcasts the data to its peers via the Peer Manager
* The Receiver Thread also gets notified about new peers by the Load Balancer and optionally the message may contain a flag that requests that the worker sends all of its data to the new peer
* A Peer Listener Thread listens for messages from its peers they can be full data exports when the worker is fresh or data sync while the worker is ready and running

**Threads**:

* Main thread
* Receiver thread
* Export thread
* Peer Listener thread
* Input thread

**Data structures**:

* A HashMap to store client data (simple key-value store)
* A static array to store data about peers
* A simple queue for storing export data requests

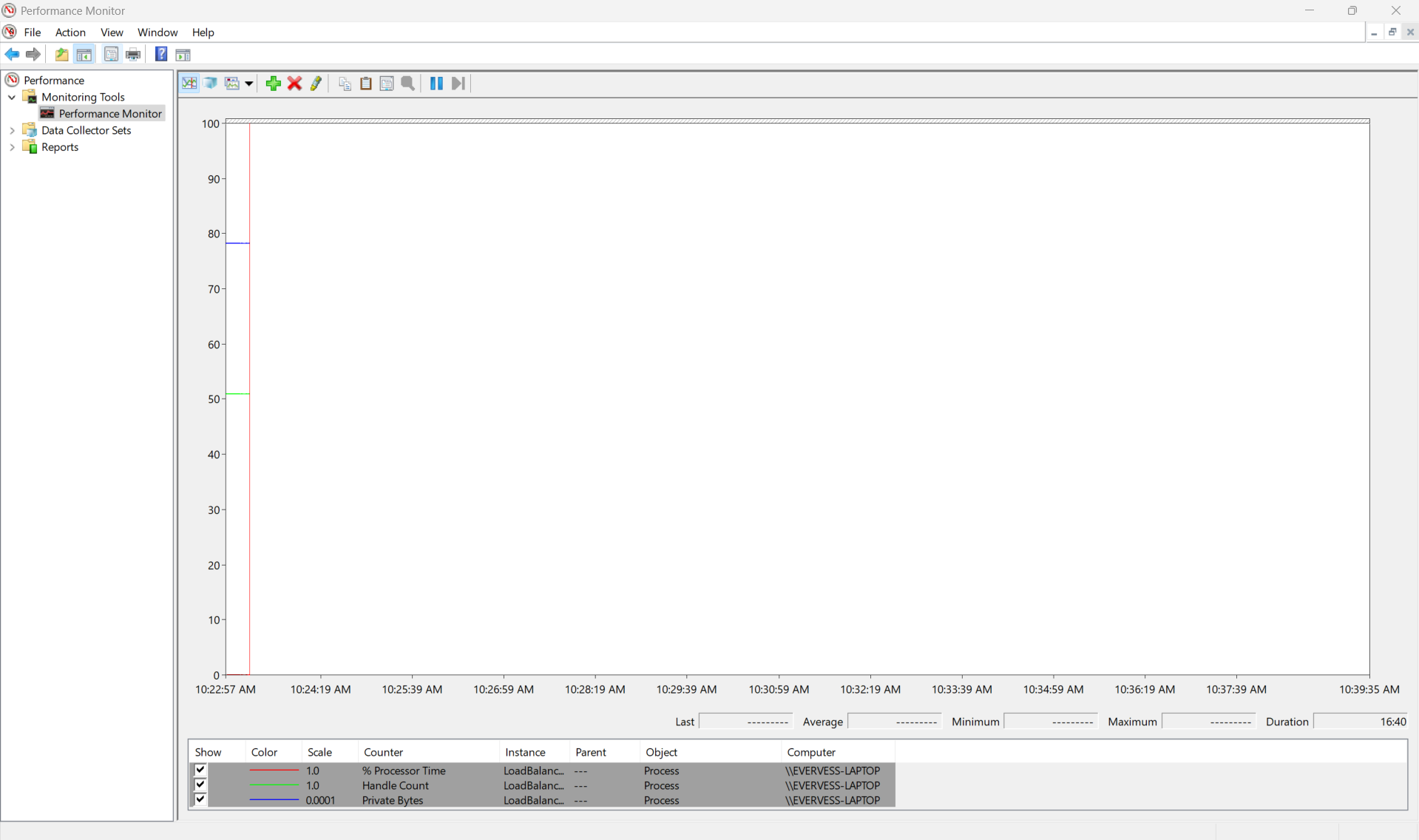
# Testing

## Step 1: Just the Load Balancer is started and waiting for a key press

Heap:



Process Monitor:

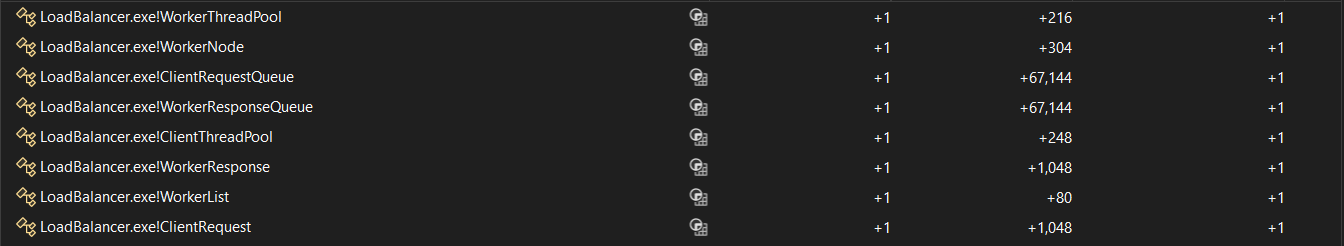


# Step 2: After the memory allocations are done and threads are started

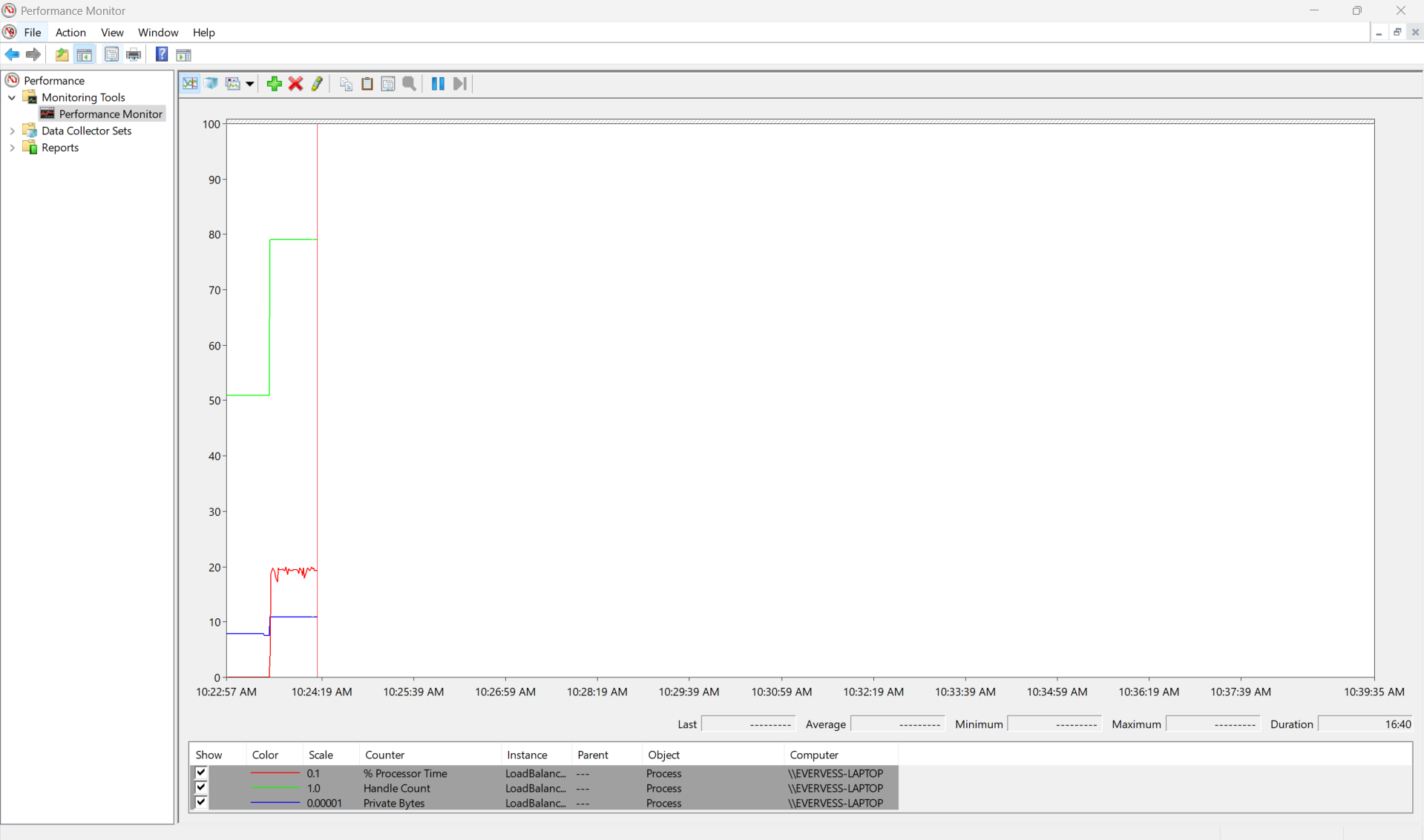
Heap:



Allocations:

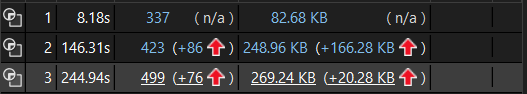


Process Monitor:

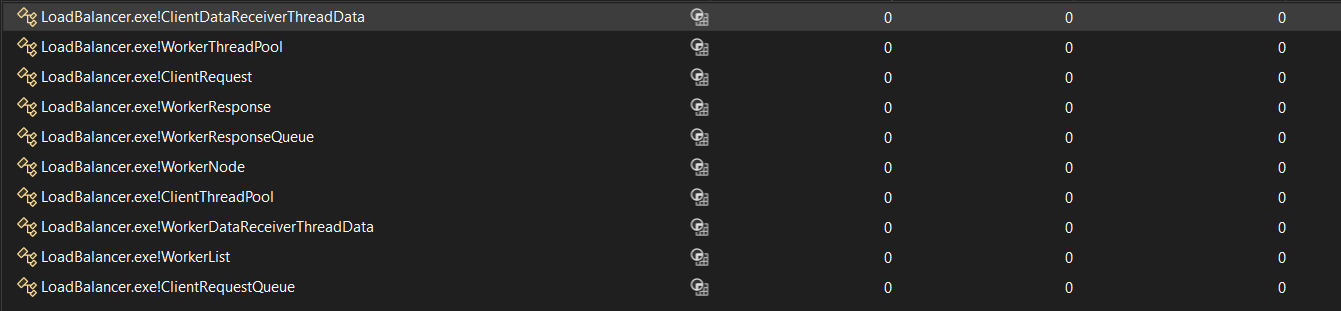


## Step 3: 2 workers are started and the Load Balancer is waiting for clients

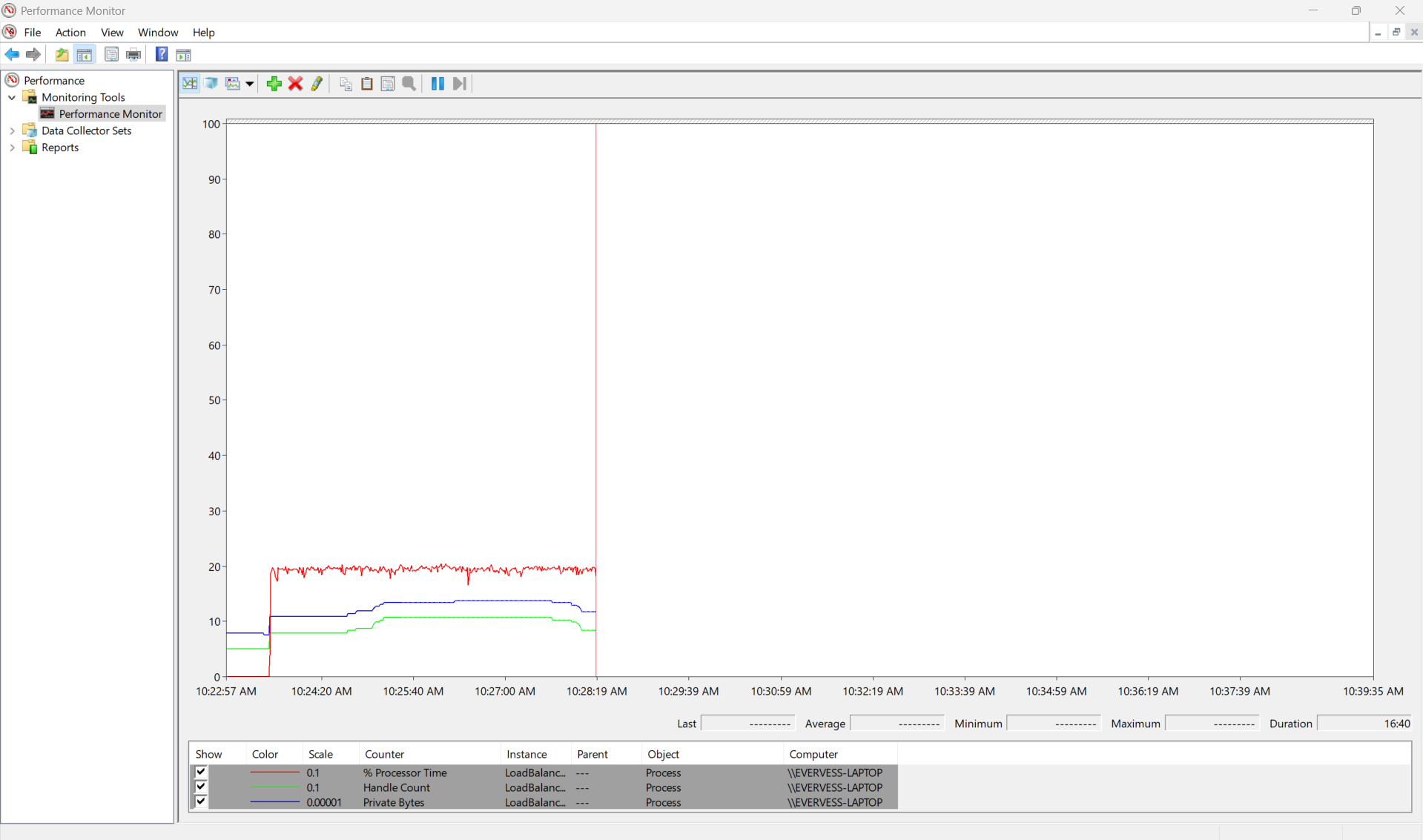
Heap:



Allocations:

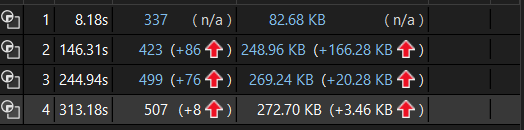


Process Monitor:

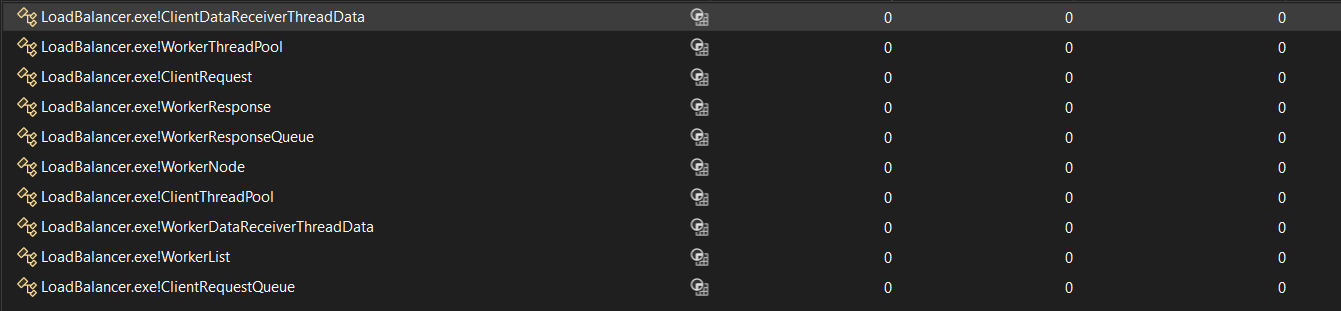


## Step 4: 3 clients are connected and the Load Balancer is under stress distributing requests

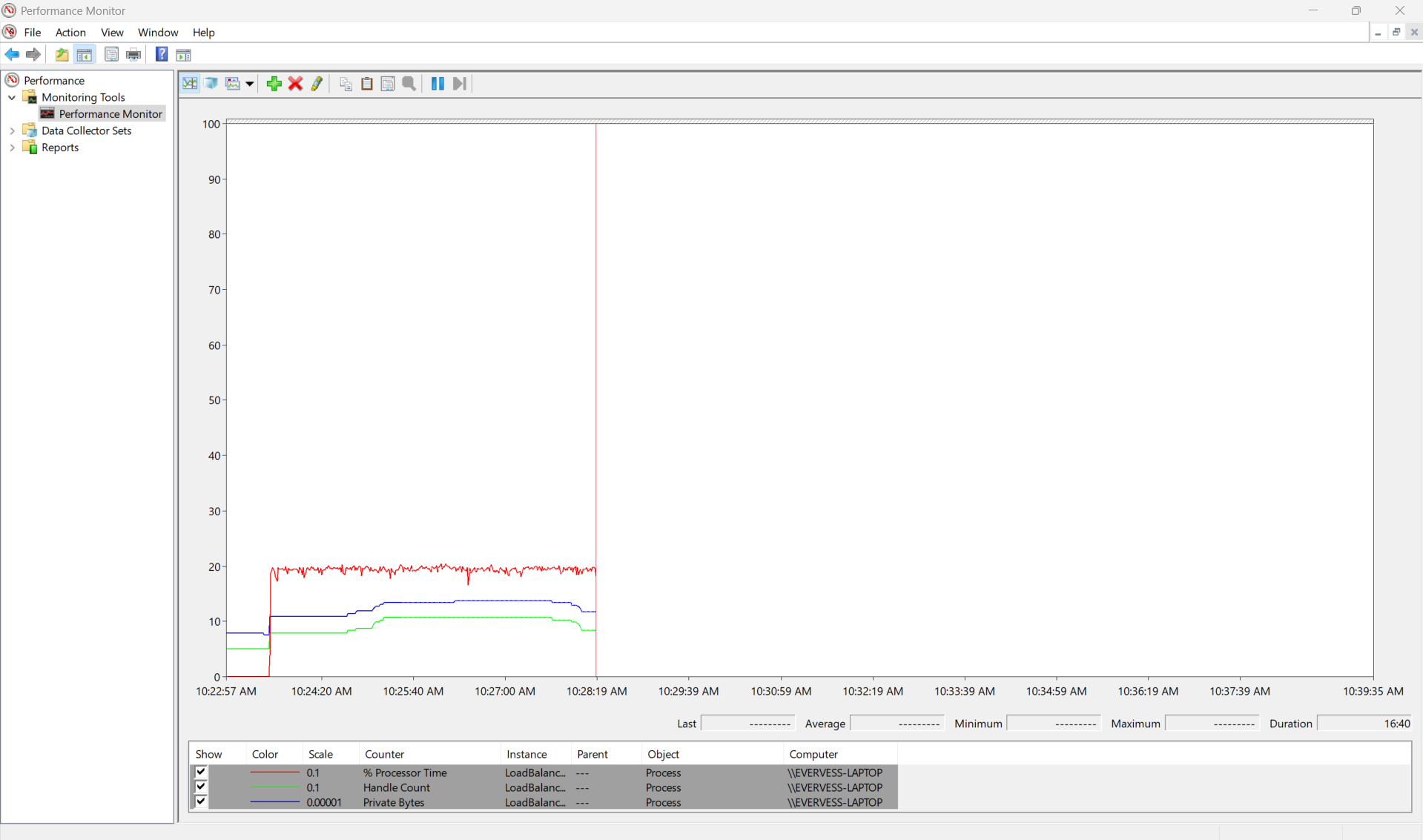
Heap:



Allocations:

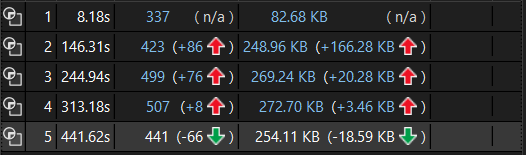


Process Monitor:

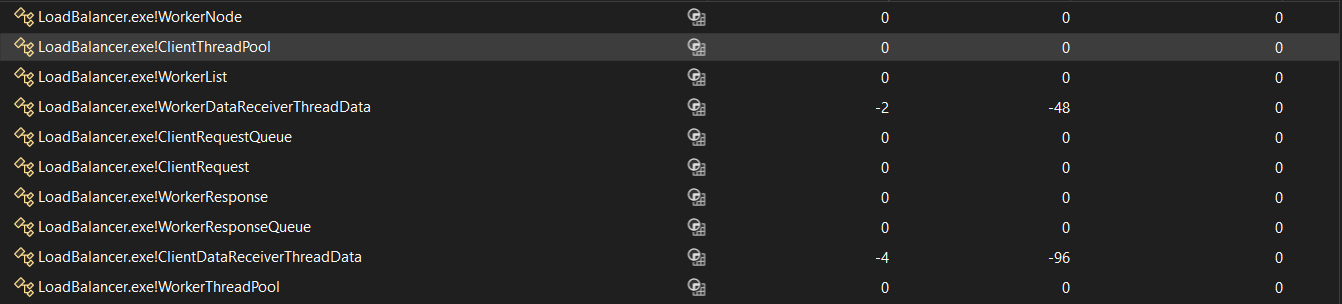


## Step 5: The clients are disconnected and the Load Balancer is passive

Heap:

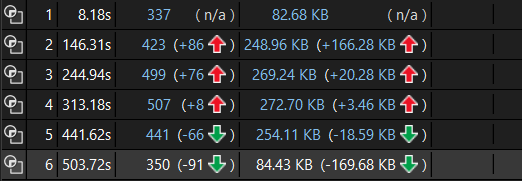


Allocations:



## Step 6: All workers are disconnected and the Load Balancer is stopped, memory deallocated and threads finished, waiting for a key press just before the final exit

Heap:



Allocations:



Process Monitor:



# Protocol

All messages start with a header (packed to 3 bytes):

* type (1 byte, MessageType)
* length (2 bytes, payload size, not including header)

| **Message Type** | **Payload Fields** |
| --- | --- |
| MSG\_PUT | keyLen (2), key (keyLen), valueLen (2), value (valueLen) |
| MSG\_PUT\_RESPONSE | result (1), keyLen (2), key (keyLen) |
| MSG\_GET | keyLen (2), key (keyLen) |
| MSG\_GET\_RESPONSE | result (1), keyLen (2), key (keyLen), valueLen (2), value (valueLen, if > 0) |
| MSG\_STORE\_REQUEST | clientId (4), keyLen (2), key (keyLen), valueLen (2), value (valueLen) |
| MSG\_STORE\_RESPONSE | result (1), clientId (4), keyLen (2), key (keyLen) |
| MSG\_RETRIEVE\_REQUEST | clientId (4), keyLen (2), key (keyLen) |
| MSG\_RETRIEVE\_RESPONSE | result (1), clientId (4), keyLen (2), key (keyLen), valueLen (2), value (valueLen, if > 0) |
| MSG\_WORKER\_REGISTRY\_START | totalWorkers (4) |
| MSG\_WORKER\_ENTRY | workerId (4), addrLen (2), address (addrLen), port (2), shouldExportData (1) |
| MSG\_WORKER\_REGISTRY\_END | (no payload) |
| MSG\_DATA\_EXPORT\_START | totalEntries (4) |
| MSG\_DATA\_ENTRY | keyLen (2), key (keyLen), valueLen (2), value (valueLen) |
| MSG\_DATA\_EXPORT\_END | (no payload) |
| MSG\_WORKER\_READY | workerId (4), peerPort (2) |
| MSG\_WORKER\_NOT\_READY | workerId (4) |
| MSG\_PEER\_NOTIFY | keyLen (2), key (keyLen), valueLen (2), value (valueLen) |
| MSG\_SHUTDOWN | (no payload) |
| MSG\_ERROR | errorCode (1), messageLen (2), message (messageLen, if > 0) |