MultilatSensorNet

Multilateration and Tracking in 3D Space Using a Distributed Network of Simulated Distance Sensors

Author: Matteo Gianferrari

Contents

1.Introduction

MultilatSensorNet aims to build a distributed, scalable network of simulated distance sensors to estimate and track an object's position in 3D space via multilateration. The system comprises four main components—Node, Network, Target, and Client—that work together to measure distances, compute multilateration with least squares, and provide real-time object tracking.

MultilatSensorNet provides an environment in which:

1. A Target moves in 3D space following a defined trajectory.
2. Multiple Node + Sensor pairs measure the Euclidean distance to the Target.
3. A centralized Network component aggregates these distances to compute the Target's position using multilateration with least squares.
4. A Client application requests the global Target position from the Network, using a Tracker and KalmanFilter to track the movement over time.

By utilizing a distributed node-sensor approach, the system aims to be scalable, robust, and adaptable to various use cases.

1.1.Features

Below are reported the main features of MultilatSensorNet:

- Distributed Sensing (multiple Nodes measuring distances in 3D space, communicating results to a central Network).
- Scalable Multilateration (use of a least squares algorithm to estimate the position of the Target from multiple distance measurements).
- Kalman Filter Integration (the Client tracks the Target in real time, smoothing measurements via a Kalman Filter).
- Modular Architecture (each component—Node, Network, Target, and Client—can be extended or replaced, enabling flexible experimentation).

1.2.System Components

Below are reported the main components of MultilatSensorNet:

- Node (Node + Sensor Packages)
    1. Simulated distance sensor that measures the Euclidean distance to the Target in 3D space.
    2. Handles network requests and communications with the Network.
- Network (Network + Estimator Packages)
    1. Central component managing Nodes registration, distance retrieval, and communication requests.

    2. Performs multilateration using least squares to compute the Target’s position.
    3. Communicates computed positions to the Client.

- Target
    1. An object that moves in 3D space according to a defined trajectory.
    2. Provides distance information to each Node upon request.
- Client
    1. Requests the global Target position from the Network.
    2. Maintains a Tracker with a Kalman Filter for smooth real-time tracking.

2.Software Requirements Specification

This chapter outlines the Software Requirements Specification (SRS) for the system, detailing both functional and non-functional requirements. It provides a comprehensive description of the expected features, behaviors, and constraints of the system to ensure proper implementation and performance.

2.1.System SRS

This section defines the high-level requirements for the system as a whole. It specifies the structural composition and constraints of the system, including instances required to achieve the desired functionality.

2.1.1.Functional Requirements

Requirement ID	Requirement Title	Requirement Description
SRS-SYSTEM-001	Client Instance	The system must be composed of only one client instance.
SRS-SYSTEM-002	Network Instance	The system must be composed of only one network instance.
SRS-SYSTEM-003	Target Instance	The system must be composed of only one target instance.
SRS-SYSTEM-004	Node Instances	The system must be composed of at least three node instances.

2.2.Client Component SRS

This section describes the requirements specific to the Client component of the system. It includes functional requirements, non-functional requirements, and dependencies, ensuring that the client meets performance and usability expectations.

2.2.1.Functional Requirements

Requirement ID	Requirement Title	Requirement Description
SRS-CLIENT-001	Client Initialization	The client must possess an unique ID.
SRS-CLIENT-002	Data Output	The client must allow logging the predicted positions in a CSV format.
SRS-CLIENT-003	Network Communication	The client must connect to the distributed network via gRPC for position retrieval requests.

SRS-CLIENT-004	Network Communication	The client must send a start request to the distributed network to activate it.
SRS-CLIENT-005	Real-Time Tracking	The client must retrieve the target's position at a configurable frequency in Hertz (Hz).
SRS-CLIENT-006	Real-Time Tracking	The client must predict the target future position.

2.2.2.Non-Functional Requirements

Requirement ID	Requirement Title	Requirement Description
SRS-CLIENT-007	Fault Tolerance	The client must handle connection failures or unexpected shutdowns gracefully.
SRS-CLIENT-008	Usability	Command-line arguments must enable verbose logging for debugging purposes.
SRS-CLIENT-009	Usability	Logs must include detailed status updates for debugging.

2.2.3.Interfaces and Dependencies

External Interfaces:

- gRPC-based communication for interacting with the distributed network.
- CSV file output for storing predicted target positions.

Dependencies:

- grpc for network communication.
- numpy for mathematical operations.
- time for scheduling.
- datetime for timestamping.

2.3.Target Component SRS

This section describes the requirements specific to the Target component of the system. It includes functional requirements, non-functional requirements, and dependencies, ensuring that the target meets performance and usability expectations.

2.3.1.Functional Requirements

Requirement ID	Requirement Title	Requirement Description
SRS-TARGET-001	Target Initialization	The target must possess an unique network address.
SRS-TARGET-002	Trajectory	The target must load a predefined trajectory from a

	Management	JSON file containing waypoints with x, y, and z coordinates.
SRS-TARGET-003	Trajectory Management	The target must support looping the trajectory indefinitely if specified.
SRS-TARGET-004	Trajectory Management	The trajectory must be updated at a configurable frequency in Hertz (Hz).
SRS-TARGET-005	Position Updates	The target must update its position based on trajectory waypoints.
SRS-TARGET-006	Network Communication	The target must provide a gRPC service to handle position requests from nodes.
SRS-TARGET-007	Network Communication	The gRPC service must handle multiple concurrent requests.
SRS-TARGET-008	Network Communication	The gRPC service must send the current target position as a response to nodes.

2.3.2.Non-Functional Requirements

Requirement ID	Requirement Title	Requirement Description
SRS-TARGET-009	Scalability	The target must support scalable addition of requesting nodes without performance degradation.
SRS-TARGET-010	Thread-Safety	The target must ensure fairness among threads, avoiding starvation or deadlocks.
SRS-TARGET-011	Fault Tolerance	The target must handle invalid or missing waypoints in the trajectory file gracefully, throwing exceptions.
SRS-TARGET-012	Fault Tolerance	The gRPC service must handle sudden shutdowns and terminate safely.
SRS-TARGET-013	Usability	Command-line arguments must support enabling verbose logging for debugging purposes.
SRS-TARGET-014	Usability	Logs must provide detailed status updates for debugging, including position updates and server status.

2.3.3.Interfaces and Dependencies

External Interfaces:

- gRPC-based communication for target position queries.
- JSON file input for defining trajectories.

Dependencies:

- numpy for mathematical operations.
- grpc for network communication.
- threading for thread management.
- concurrent.futures for thread management.

2.4.Node Component SRS

This section describes the requirements specific to the Node component of the system. It includes functional requirements, non-functional requirements, and dependencies, ensuring that the nodes meet performance and usability expectations.

2.4.1.Functional Requirements

Requirement ID	Requirement Title	Requirement Description
SRS-NODE-001	Node Initialization	The node must possess an unique ID.
SRS-NODE-002	Node Initialization	The node must possess a position in a 3D space.
SRS-NODE-003	Node Initialization	The node must possess unique network addresses.
SRS-NODE-004	Sensor Integration	The node must include a sensor to measure distances to the target in 3D space.
SRS-NODE-005	Distance Computation	The sensor must compute Euclidean distances.
SRS-NODE-006	Measurement Accuracy	The sensor must have configurable accuracy for its measurements.
SRS-NODE-007	Measurement Frequency	The sensor must have configurable update frequency.
SRS-NODE-008	Network Registration	The node must register itself with the distributed network via gRPC.
SRS-NODE-009	Communication Handling	The node must handle incoming requests from the distributed network for distance measurements.

2.4.2.Non-Functional Requirements

Requirement ID	Requirement Title	Requirement Description
SRS-NODE-010	Scalability	The node must support scalability to operate as part of a large distributed network.
SRS-NODE-011	Thread-Safety	The node must ensure fairness among threads, avoiding starvation or deadlocks.
SRS-NODE-012	Usability	Command-line arguments must enable verbose logging for debugging purposes.

SRS-NODE-013	Logging	Logs must include details of measurements, communications, and errors for debugging.

2.4.3.Interfaces and Dependencies

External Interfaces:

- gRPC-based communication for interacting with the distributed network.
- ZeroMQ for asynchronous messaging with the distributed network for data aggregation.

Dependencies:

- grpc for network communication.
- numpy for mathematical computations.
- zmq for ZeroMQ messaging.
- threading for concurrency management.

2.5.Network Component SRS

This section describes the requirements specific to the Network component of the system. It includes functional requirements, non-functional requirements, and dependencies, ensuring that the network meets performance and usability expectations.

2.5.1.Functional Requirements

Requirement ID	Requirement Title	Requirement Description
SRS-NETWORK-001	Network Initialization	The network must possess an unique network address.
SRS-NETWORK-002	Node Management	The network must keep track of the connected nodes, including their positions and addresses.
SRS-NETWORK-003	Position Computation	The network must compute the target position using multilateration based on distance measurements.
SRS-NETWORK-004	Distance Retrieval	The network must retrieve distance measurements from nodes.
SRS-NETWORK-005	Network Communication	The network must provide a gRPC service to handle network operation requests from a client.
SRS-NETWORK-006	Node Addition	The network must provide a gRPC service to handle node addition requests from nodes.
SRS-NETWORK-007	Concurrent Requests Handling	The gRPC service must handle multiple concurrent requests.
SRS-NETWORK-008	Operation Status	The gRPC service must send the status of operations

	Responses	as responses to both clients and nodes.
SRS-NETWORK-009	Position Reporting	The gRPC service must send the current target global position as a response to a client when asked.
SRS-NETWORK-010	Node Count Reporting	The gRPC service must send the number of connected nodes as a response to a client when the network is started.

2.5.2.Non-Functional Requirements

Requirement ID	Requirement Title	Requirement Description
SRS-NETWORK-011	Scalability	The network must scale to handle multiple nodes without performance degradation.
SRS-NETWORK-012	Thread-Safety	The target must ensure fairness among threads, avoiding starvation or deadlocks.
SRS-NETWORK-013	Fault Tolerance	The network must handle connection failures or unexpected shutdowns gracefully.
SRS-NETWORK-014	Usability	Command-line arguments must enable verbose logging for debugging purposes.
SRS-NETWORK-015	Logging	Logs must include detailed status updates for debugging, including network states and communication processes.

2.5.3.Interfaces and Dependencies

External Interfaces:

- gRPC-based communication for interacting with nodes and clients.
- ZeroMQ for asynchronous messaging with nodes for data aggregation.

Dependencies:

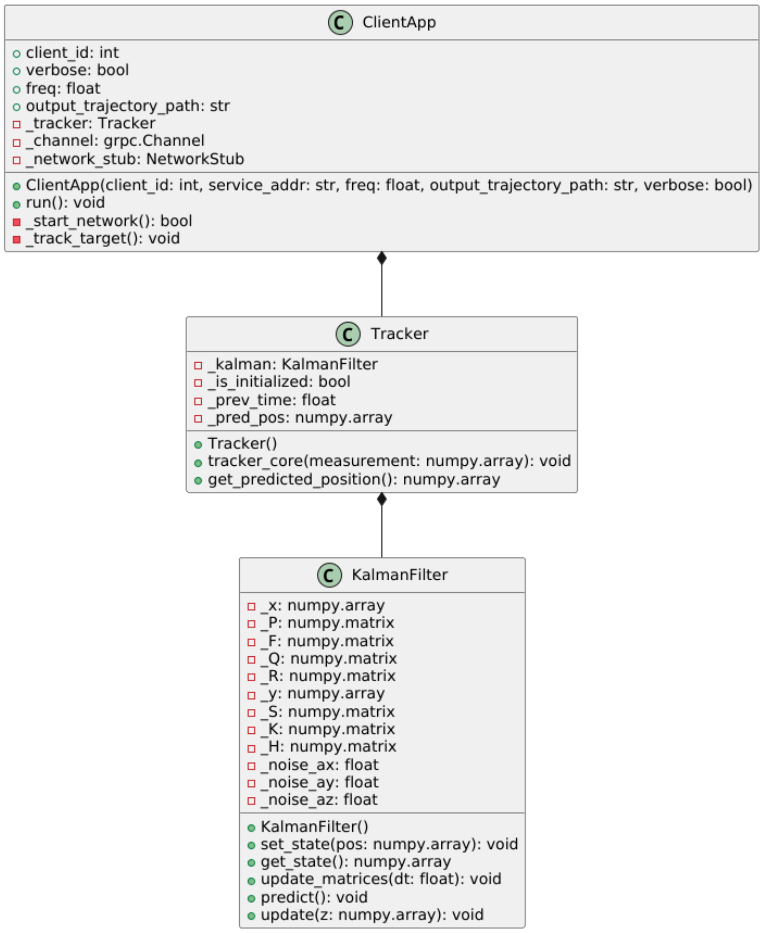
- grpc for network communication.
- numpy for mathematical operations.
- scipy for numerical optimization.
- zmq for ZeroMQ messaging.
- threading for concurrency management.

3.Class Diagrams

This chapter presents the UML class diagrams for the components of the system. These diagrams provide a visual representation of the structure and relationships between classes, highlighting their attributes, methods, and associations. The diagrams serve as a blueprint for the implementation of the system, ensuring clarity in design.

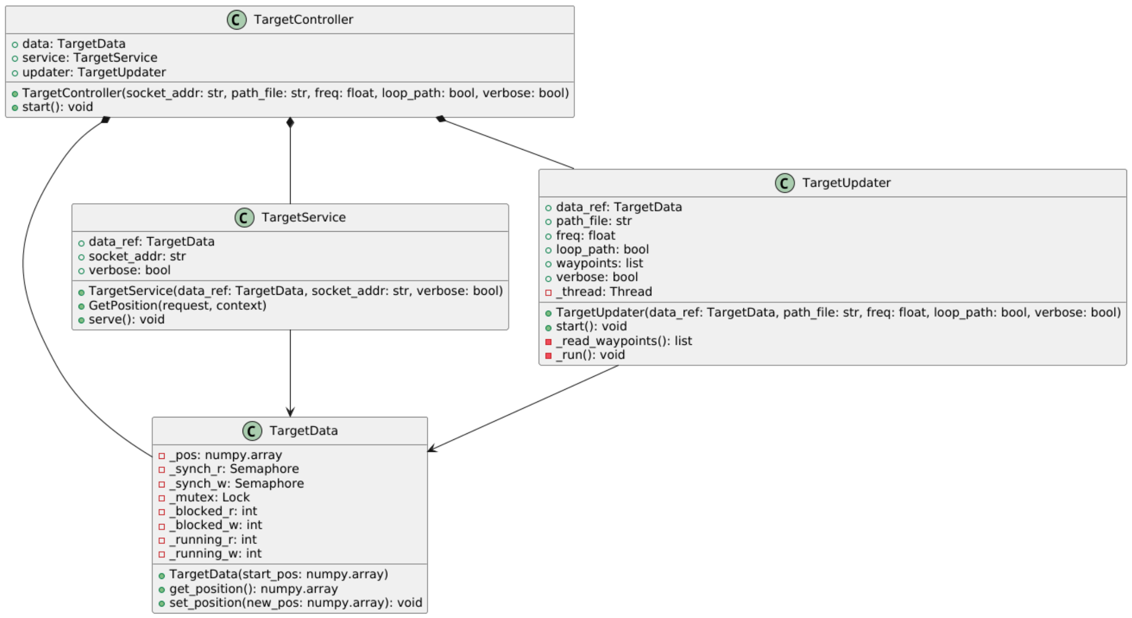
3.1.Client Component

This section focuses on the Client component, detailing its structure through a UML class diagram. The diagram illustrates the key classes within the client, their attributes, and methods.



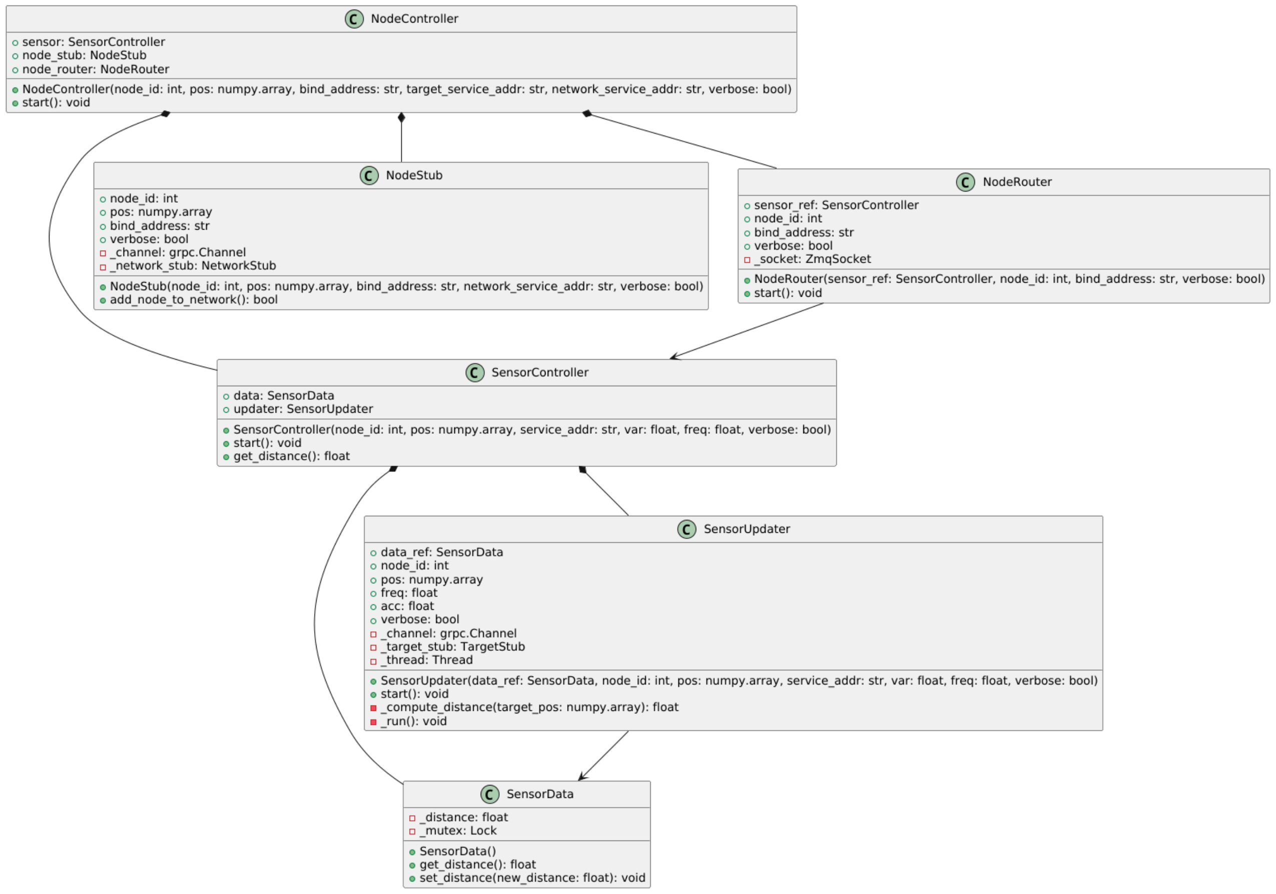
3.2.Target Component

This section focuses on the Target component, detailing its structure through a UML class diagram. The diagram illustrates the key classes within the target, their attributes, and methods.



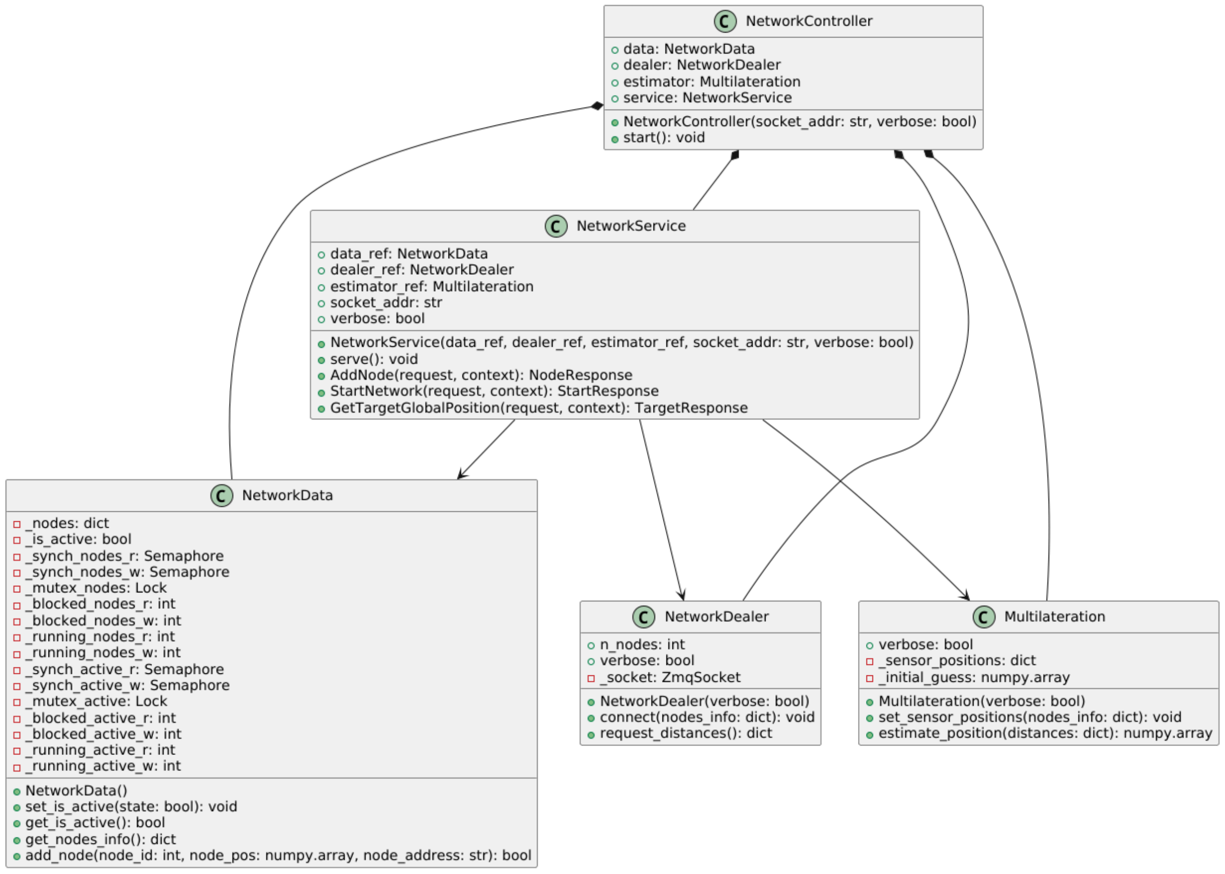
3.3.Node Component

This section focuses on the Node component, detailing its structure through a UML class diagram. The diagram illustrates the key classes within the node, their attributes, and methods.


NodeController
sensor: SensorController
node_stub: NodeStub
node_router: NodeRouter
NodeController(node_id: int, pos: numpy.array, bind_address: str, target_service_addr: str, network_service_addr: str, verbose: bool)
start(): void
NodeStub
node_id: int
pos: numpy.array
bind_address: str
verbose: bool
_channel: grpc.Channel
_network_stub: NetworkStub
NodeStub(node_id: int, pos: numpy.array, bind_address: str, network_service_addr: str, verbose: bool)
add_node_to_network(): bool
NodeRouter
sensor_ref: SensorController
node_id: int
bind_address: str
verbose: bool
_socket: ZmqSocket
NodeRouter(sensor_ref: SensorController, node_id: int, bind_address: str, verbose: bool)
start(): void
SensorController
data: SensorData
updater: SensorUpdater
SensorController(node_id: int, pos: numpy.array, service_addr: str, var: float, freq: float, verbose: bool)
start(): void
get_distance(): float
SensorUpdater
data_ref: SensorData
node_id: int
pos: numpy.array
freq: float
acc: float
verbose: bool
_channel: grpc.Channel
_target_stub: TargetStub
_thread: Thread
SensorUpdater(data_ref: SensorData, node_id: int, pos: numpy.array, service_addr: str, var: float, freq: float, verbose: bool)
start(): void
_compute_distance(target_pos: numpy.array): float
_run(): void
SensorData
_distance: float
_mutex: Lock
SensorData()
get_distance(): float
set_distance(new_distance: float): void

3.4.Network Component

This section focuses on the Network component, detailing its structure through a UML class diagram. The diagram illustrates the key classes within the network, their attributes, and methods.


NetworkController
data: NetworkData
dealer: NetworkDealer
estimator: Multilateration
service: NetworkService
NetworkController(socket_addr: str, verbose: bool)
start(): void
NetworkService
data_ref: NetworkData
dealer_ref: NetworkDealer
estimator_ref: Multilateration
socket_addr: str
verbose: bool
NetworkService(data_ref, dealer_ref, estimator_ref, socket_addr: str, verbose: bool)
serve(): void
AddNode(request, context): NodeResponse
StartNetwork(request, context): StartResponse
GetTargetGlobalPosition(request, context): TargetResponse
NetworkData
_nodes: dict
_is_active: bool
_synch_nodes_r: Semaphore
_synch_nodes_w: Semaphore
_mutex_nodes: Lock
_blocked_nodes_r: int
_blocked_nodes_w: int
_running_nodes_r: int
_running_nodes_w: int
_synch_active_r: Semaphore
_synch_active_w: Semaphore
_mutex_active: Lock
_blocked_active_r: int
_blocked_active_w: int
_running_active_r: int
_running_active_w: int
NetworkData()
set_is_active(state: bool): void
get_is_active(): bool
get_nodes_info(): dict
add_node(node_id: int, node_pos: numpy.array, node_address: str): bool
NetworkDealer
n_nodes: int
verbose: bool
_socket: ZmqSocket
NetworkDealer(verbose: bool)
connect(nodes_info: dict): void
request_distances(): dict
Multilateration
verbose: bool
_sensor_positions: dict
_initial_guess: numpy.array
Multilateration(verbose: bool)
set_sensor_positions(nodes_info: dict): void
estimate_position(distances: dict): numpy.array

4.Communication Diagrams

This chapter presents UML Communication Diagrams, specifically focusing on the sequence diagrams that describe the protocols used within the system. These diagrams provide a visual representation of the interactions between components, illustrating the flow of messages required to execute specific procedures.

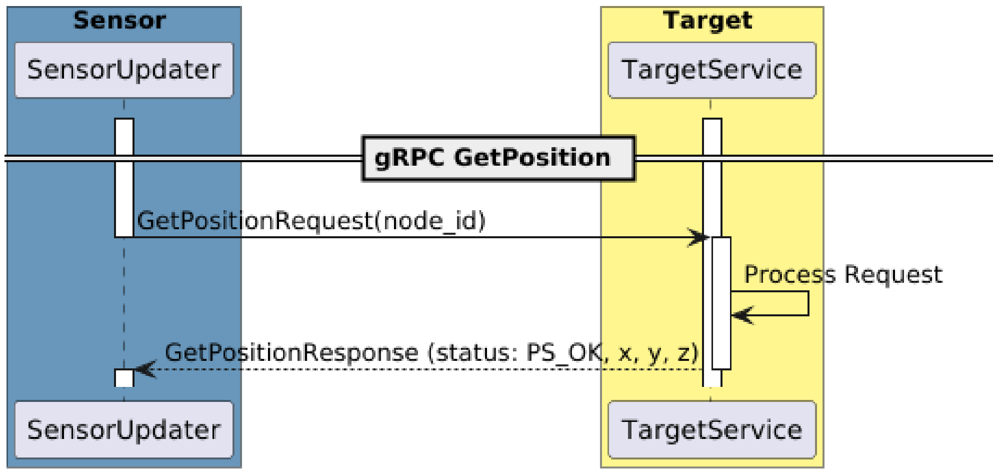
4.1.gRPC: GetPosition Procedure

This section details the gRPC GetPosition procedure using a UML sequence diagram to describe the protocol for retrieving the position of the target. The interaction involves two main components:

- SensorUpdater (Client) – Initiates the request to retrieve the target's position.
- TargetService (Server) – Processes the incoming request and responds with the requested data.

The sequence diagram follows these steps:

1. Initialization and Activation (both SensorUpdater and TargetService threads are active; TargetService runs in a thread, waiting for incoming requests from clients).
2. Request Transmission (SensorUpdater sends a GetPositionRequest message to the TargetService, including the requesting node's ID; the SensorUpdater thread is blocked while waiting for a response, ensuring synchronous communication).
3. Request Processing (TargetService assigns the request to a worker thread from its thread pool to handle asynchronous processing; the GetPosition procedure processes the request to retrieve the target's position).
4. Response Transmission (TargetService sends a GetPositionResponse message back to the SensorUpdater; the response includes a status code (PS_OK) and the target's coordinates (x, y, z)).
5. Completion (the SensorUpdater is reactivated, completing the protocol).



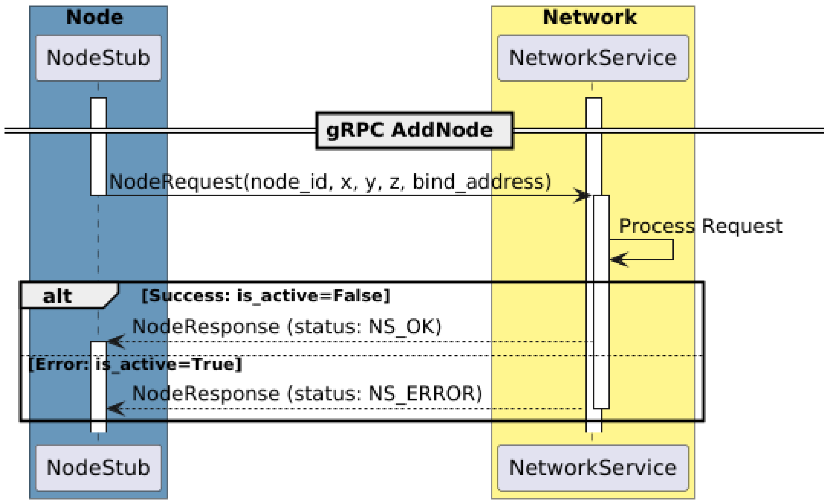
4.2.gRPC: AddNode Procedure

This section details the gRPC AddNode procedure using a UML sequence diagram to describe the protocol for adding a node to the distributed network. The interaction involves two main components:

- NodeStub (Client) – Initiates the request to add a node with its ID, position, and bind address.
- NetworkService (Server) – Processes the request and determines whether the node can be added based on the network's activation state.

The sequence diagram follows these steps:

1. Initialization and Activation (both NodeStub and NetworkService threads are active; NetworkService runs in a thread, waiting for incoming requests from clients).
2. Request Transmission (NodeStub sends an AddNode RPC request containing the node ID, position (x, y, z), and bind address; the NodeStub thread is blocked while waiting for a response, ensuring synchronous communication).
3. Request Processing (NetworkService assigns the request to a worker thread from its thread pool for asynchronous processing; the AddNode procedure processes the request to determine whether the node can be added to the network).
4. Conditional Response Handling:
    a. Success Case (is_active = False; if the distributed network is not active, the node is successfully added; NetworkService sends a NodeResponse message to NodeStub with a positive status (NS_OK); NodeStub is reactivated, completing the protocol).
    b. Error Case (is_active = True; if the distributed network is already active, the node cannot be added; NetworkService sends a NodeResponse message to NodeStub with a negative status (NS_ERROR); NodeStub is reactivated, completing the protocol).


Node
Network
NodeStub
NetworkService
gRPC AddNode
NodeRequest(node_id, x, y, z, bind_address)
Process Request
alt
[Success: is_active=False]
NodeResponse (status: NS_OK)
[Error: is_active=True]
NodeResponse (status: NS_ERROR)
NodeStub
NetworkService

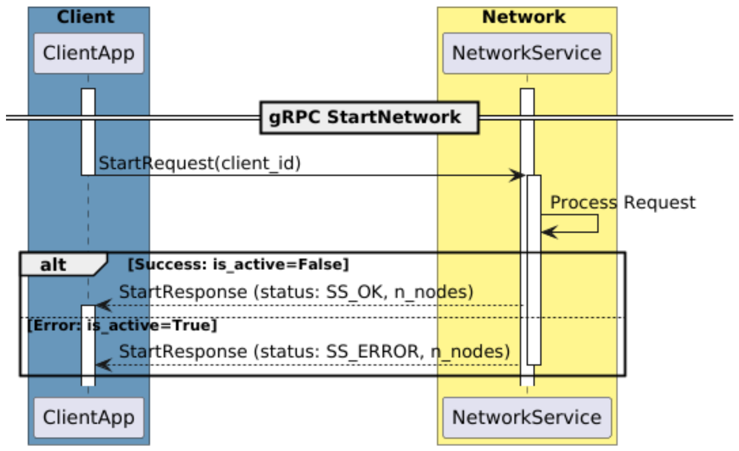
4.3.gRPC: StartNetwork Procedure

This section details the gRPC StartNetwork procedure using a UML sequence diagram to describe the protocol for starting the distributed network. The interaction involves two primary components:

- ClientApp (Client) – Initiates the request to start the network.
- NetworkService (Server) – Processes the request and determines whether the network can be started based on its activation state.

The sequence diagram follows these steps:

1. Initialization and Activation (both ClientApp and NetworkService threads are active; NetworkService runs in a thread, waiting for incoming requests from clients).
2. Request Transmission (ClientApp sends a StartNetwork RPC request with a StartRequest message containing the client ID; the ClientApp thread is blocked while waiting for a response, ensuring synchronous communication).
3. Request Processing (NetworkService assigns the request to a worker thread from its thread pool, enabling asynchronous handling of multiple incoming requests; the StartNetwork procedure processes the request to determine whether the network can be started).
4. Conditional Response Handling:
    a. Success Case (is_active = False; if the distributed network is not active, it is successfully started; NetworkService sends a StartResponse message back to ClientApp with a positive status code (SS_OK) and the number of nodes (n_nodes) currently present in the distributed network; ClientApp is reactivated, completing the protocol).
    b. Error Case (is_active = True; if the distributed network is already active, the request is denied; NetworkService sends a StartResponse message to ClientApp with a negative status code (SS_ERROR) and the number of nodes (n_nodes) currently present in the distributed network; ClientApp is reactivated, completing the protocol).



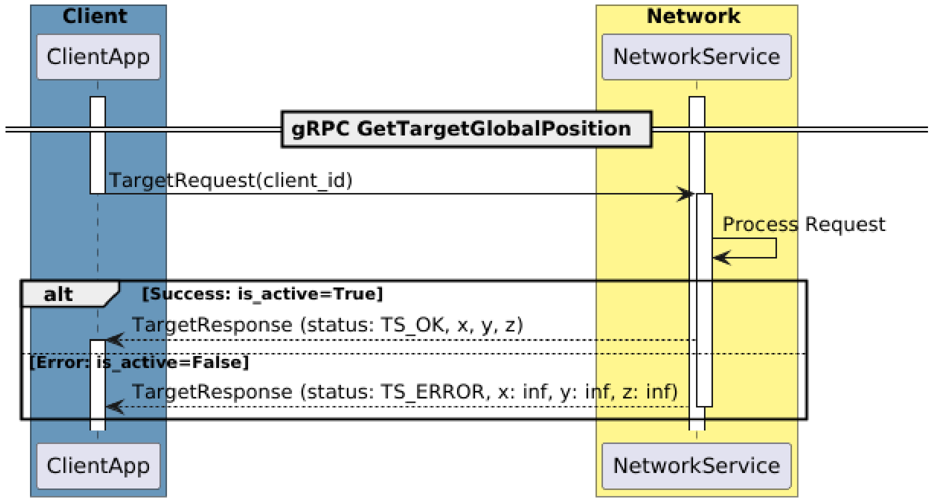
4.4.gRPC: GetTargetGlobalPosition Procedure

This section details the gRPC GetTargetGlobalPosition procedure using a UML sequence diagram to describe the protocol for retrieving the global position of the target. The interaction involves two primary components:

- ClientApp (Client) – Initiates the request to obtain the target's global position.
- NetworkService (Server) – Processes the request and determines whether the global position of the target can be computed based on the network's activation state.

The sequence diagram follows these steps:

1. Initialization and Activation (both ClientApp and NetworkService threads are active; NetworkService runs in a thread, waiting for incoming requests from clients).
2. Request Transmission (ClientApp sends a GetTargetGlobalPosition RPC request containing a TargetRequest message with the client ID; the ClientApp thread is blocked while awaiting a response, ensuring synchronous communication).
3. Request Processing (NetworkService assigns the request to a worker thread from its thread pool to handle asynchronous processing of multiple incoming requests; the GetTargetGlobalPosition procedure processes the request and determines whether the global position can be computed based on the network's activation status).
4. Conditional Response Handling:
    a. Success Case (is_active = True; if the distributed network is active, the global position of the target is successfully computed; NetworkService sends a TargetResponse message to ClientApp containing a positive status (TS_OK) and the target's global coordinates (x, y, z); ClientApp is reactivated, completing the protocol).
    b. Error Case (is_active = False; if the distributed network is not active, the global position cannot be computed; NetworkService sends a TargetResponse message to ClientApp containing a negative status (TS_ERROR) and the target's global coordinates set to infinite values (x: inf, y: inf, z: inf) to indicate an error state; ClientApp is reactivated, completing the protocol).


Client
ClientApp
Network
NetworkService
gRPC GetTargetGlobalPosition
TargetRequest(client_id)
Process Request
alt
[Success: is_active=True]
TargetResponse (status: TS_OK, x, y, z)
[Error: is_active=False]
TargetResponse (status: TS_ERROR, x: inf, y: inf, z: inf)
ClientApp
NetworkService

4.5.ZeroMQ: GetDistance

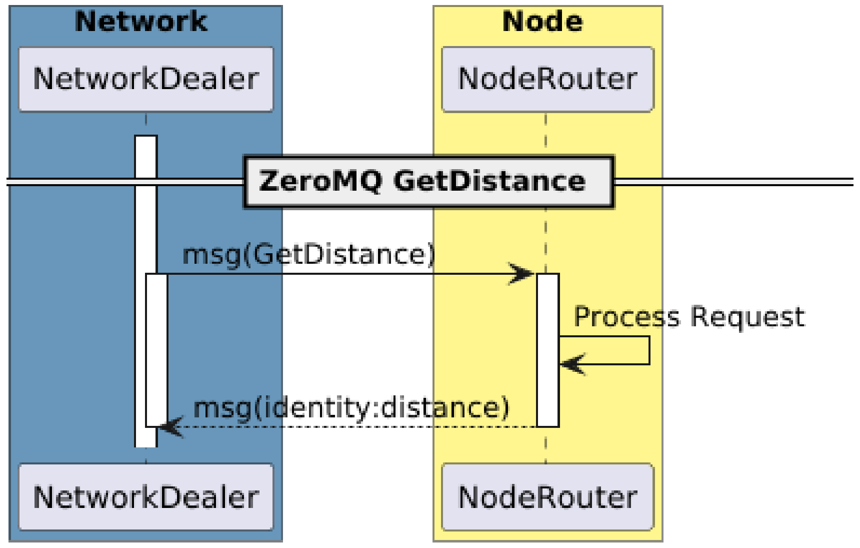
This section describes the ZeroMQ GetDistance procedure using a UML sequence diagram that outlines the protocol for retrieving the distances between nodes and the target, within the distributed network. The interaction involves two primary components:

- NetworkDealer (Network) – Initiates the request to obtain distances from all nodes.
- NodeRouter (Nodes) – Processes the request and returns the distance to the target.

The sequence diagram follows these steps:

1. Initialization and Activation (NetworkDealer thread is active and initiates the communication; NodeRouter thread remains blocked, waiting for incoming requests).
2. Request Transmission (NetworkDealer sends a GetDistance message to all nodes in the distributed network; messages are broadcasted to all connected NodeRouter).
3. Asynchronous Response Handling (NetworkDealer uses a ZeroMQ poller to asynchronously handle responses from multiple nodes; this ensures scalability by allowing parallel communication with all nodes without blocking the main thread).
4. Request Processing at Node (each NodeRouter receives the GetDistance message, unblocking its thread; the GetDistance procedure is executed to retrieve the distance between the node and the target).

5. Response Transmission (each NodeRouter sends a response back to NetworkDealer; the response includes the identity to identify the node sending the response and the distance between the node and the target; once the response is sent, NodeRouter blocks its thread waiting for the next incoming request).
6. Completion (NetworkDealer receives all responses asynchronously, collects the distances, completing the protocol).

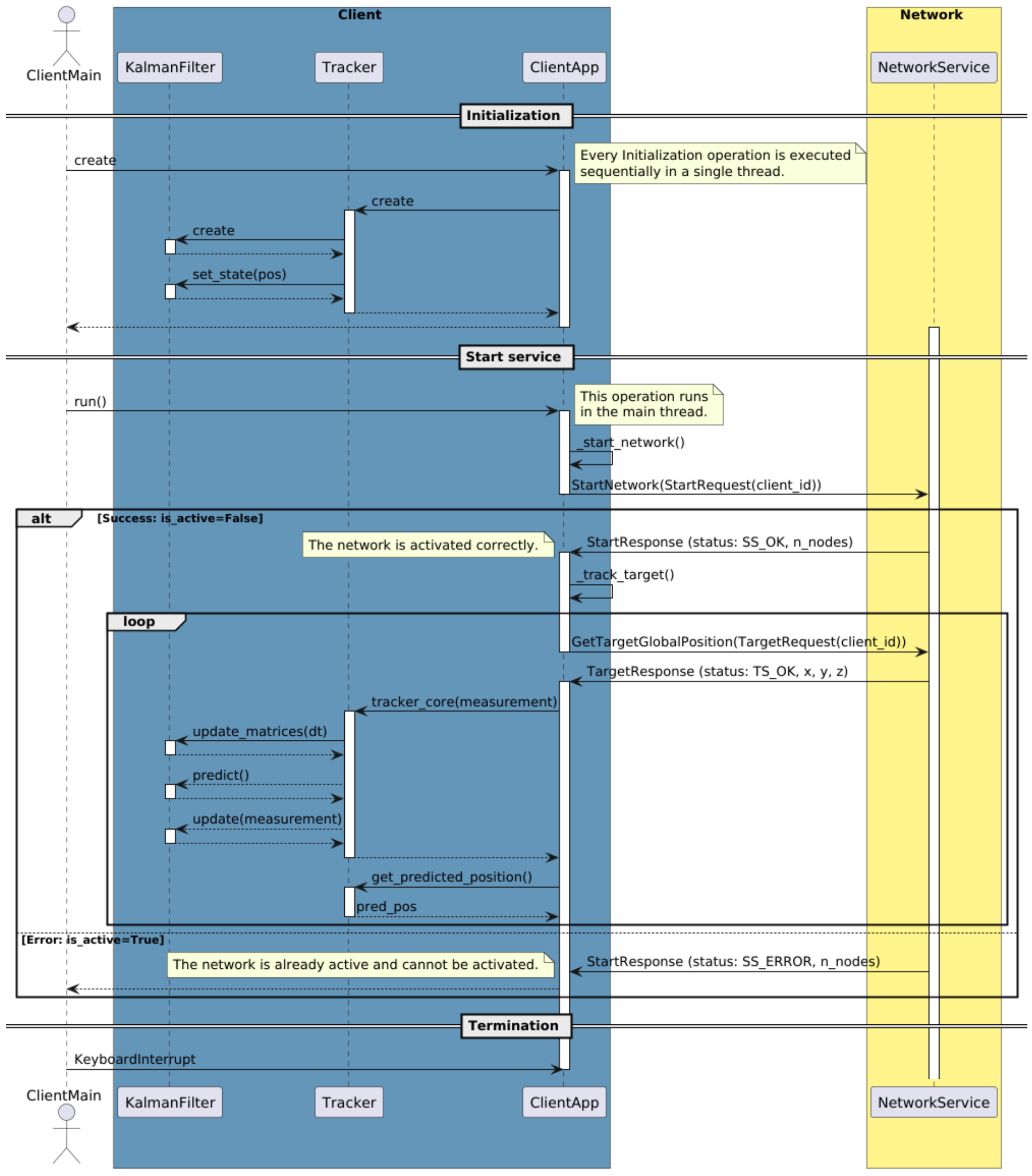


5.Sequence Diagrams

This chapter presents UML Sequence Diagrams that illustrate the dynamic behavior of the system's components. These diagrams provide a detailed view of interactions between objects, showing the sequence of messages exchanged and functions called to perform specific tasks or operations.

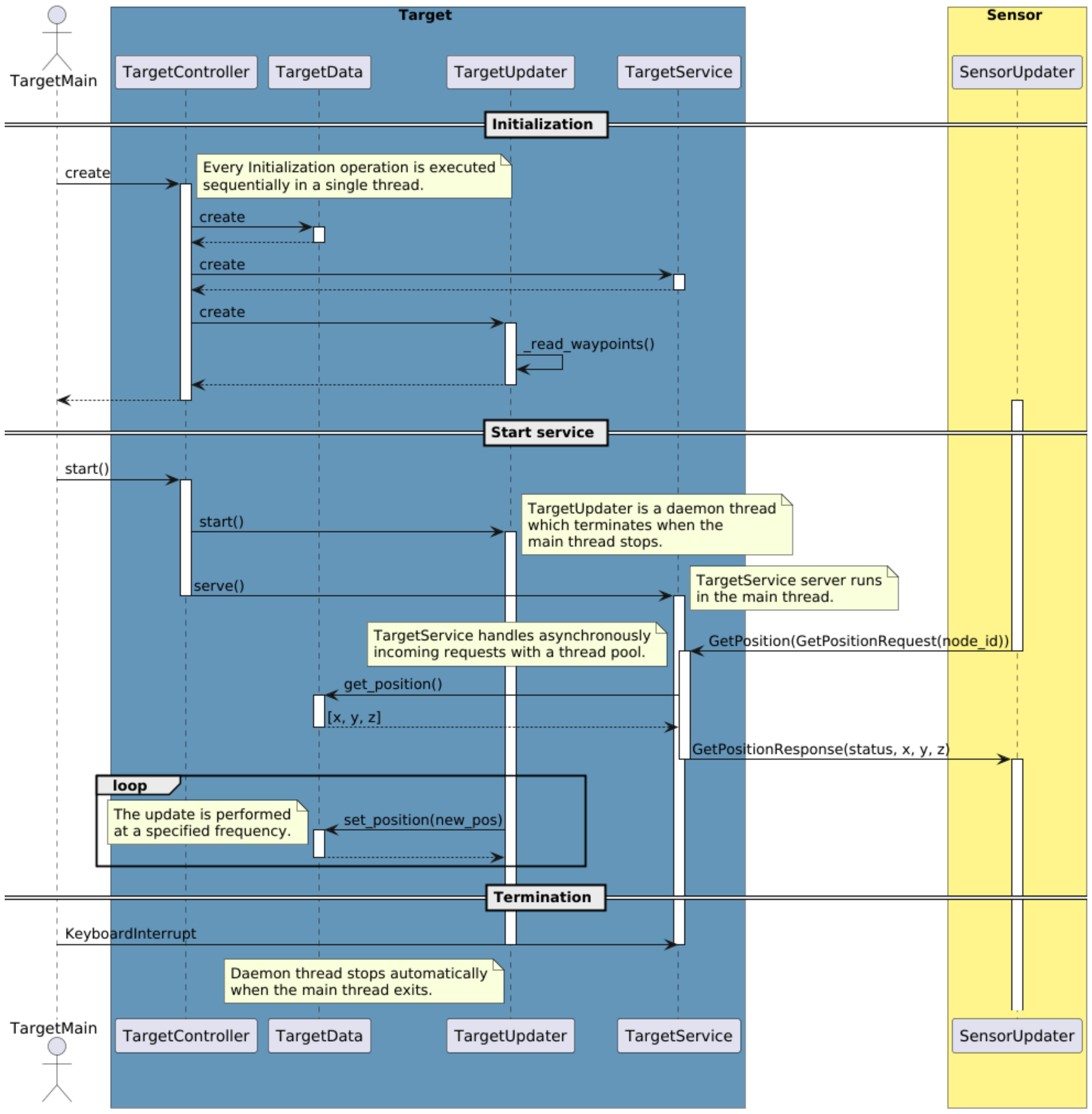
5.1.Client Component

This section focuses on the Client component, highlighting its interactions with other components in the system through sequence diagrams. For further details about the process, please refer to the related .puml file.


Client
Network
ClientMain
KalmanFilter
Tracker
ClientApp
NetworkService
Initialization
create
Every Initialization operation is executed sequentially in a single thread.
create
create
set_state(pos)
Start service
run()
This operation runs in the main thread.
_start_network()
StartNetwork(StartRequest(client_id))
alt
[Success: is_active=False]
The network is activated correctly.
StartResponse (status: SS_OK, n_nodes)
_track_target()
loop
GetTargetGlobalPosition(TargetRequest(client_id))
TargetResponse (status: TS_OK, x, y, z)
tracker_core(measurement)
update_matrices(dt)
predict()
update(measurement)
get_predicted_position()
pred_pos
[Error: is_active=True]
The network is already active and cannot be activated.
StartResponse (status: SS_ERROR, n_nodes)
Termination
KeyboardInterrupt

5.2.Target Component

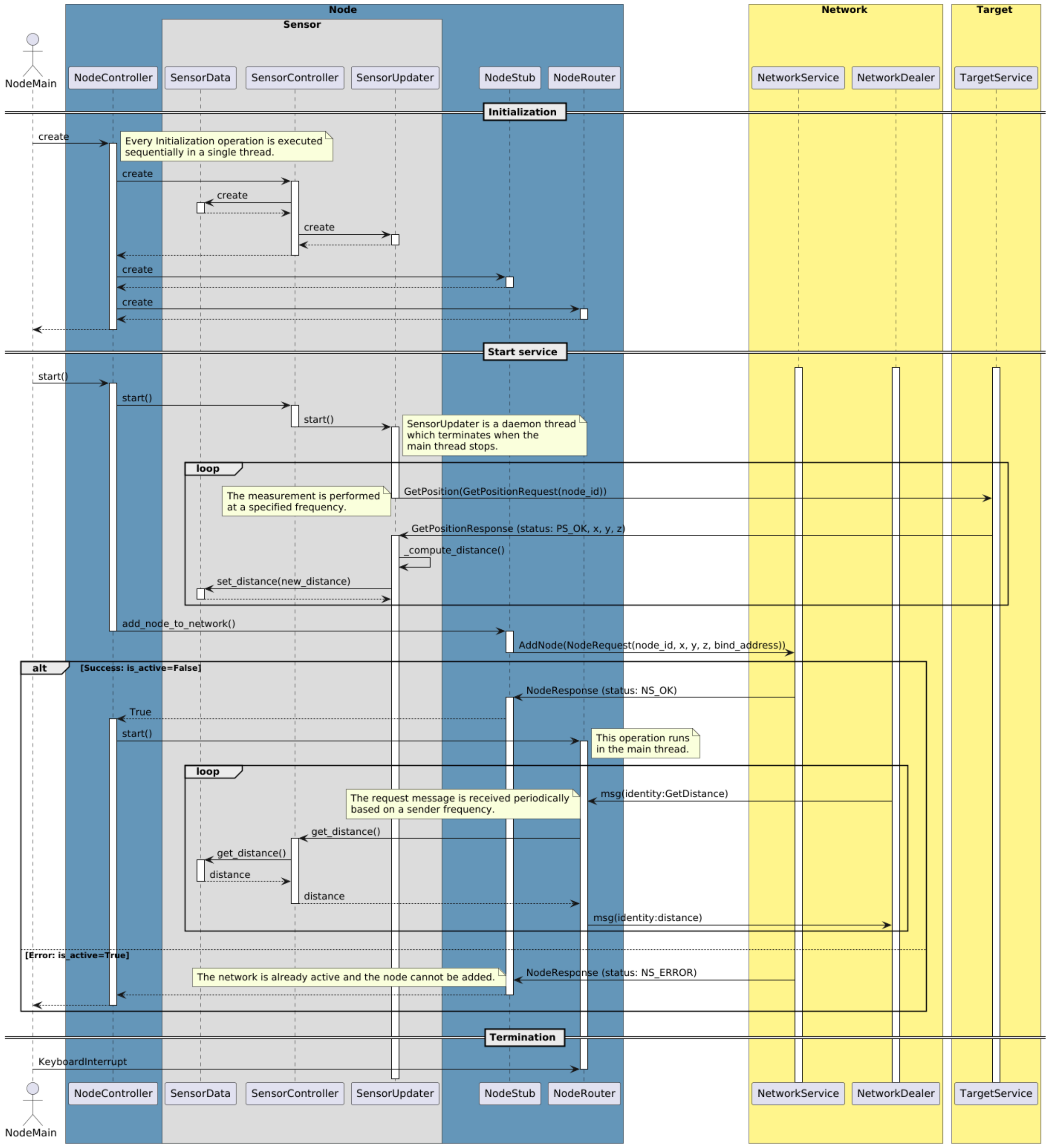
This section focuses on the Target component, highlighting its interactions with other components in the system through sequence diagrams. For further details about the process, please refer to the related .puml file.



Target
Sensor
TargetMain
TargetController
TargetData
TargetUpdater
TargetService
SensorUpdater
Initialization
create
Every Initialization operation is executed sequentially in a single thread.
create
create
create
_read_waypoints()
Start service
start()
start()
TargetUpdater is a daemon thread which terminates when the main thread stops.
serve()
TargetService server runs in the main thread.
GetPosition(GetPositionRequest(node_id))
TargetService handles asynchronously incoming requests with a thread pool.
get_position()
[x, y, z]
GetPositionResponse(status, x, y, z)
loop
The update is performed at a specified frequency.
set_position(new_pos)
Termination
KeyboardInterrupt
Daemon thread stops automatically when the main thread exits.

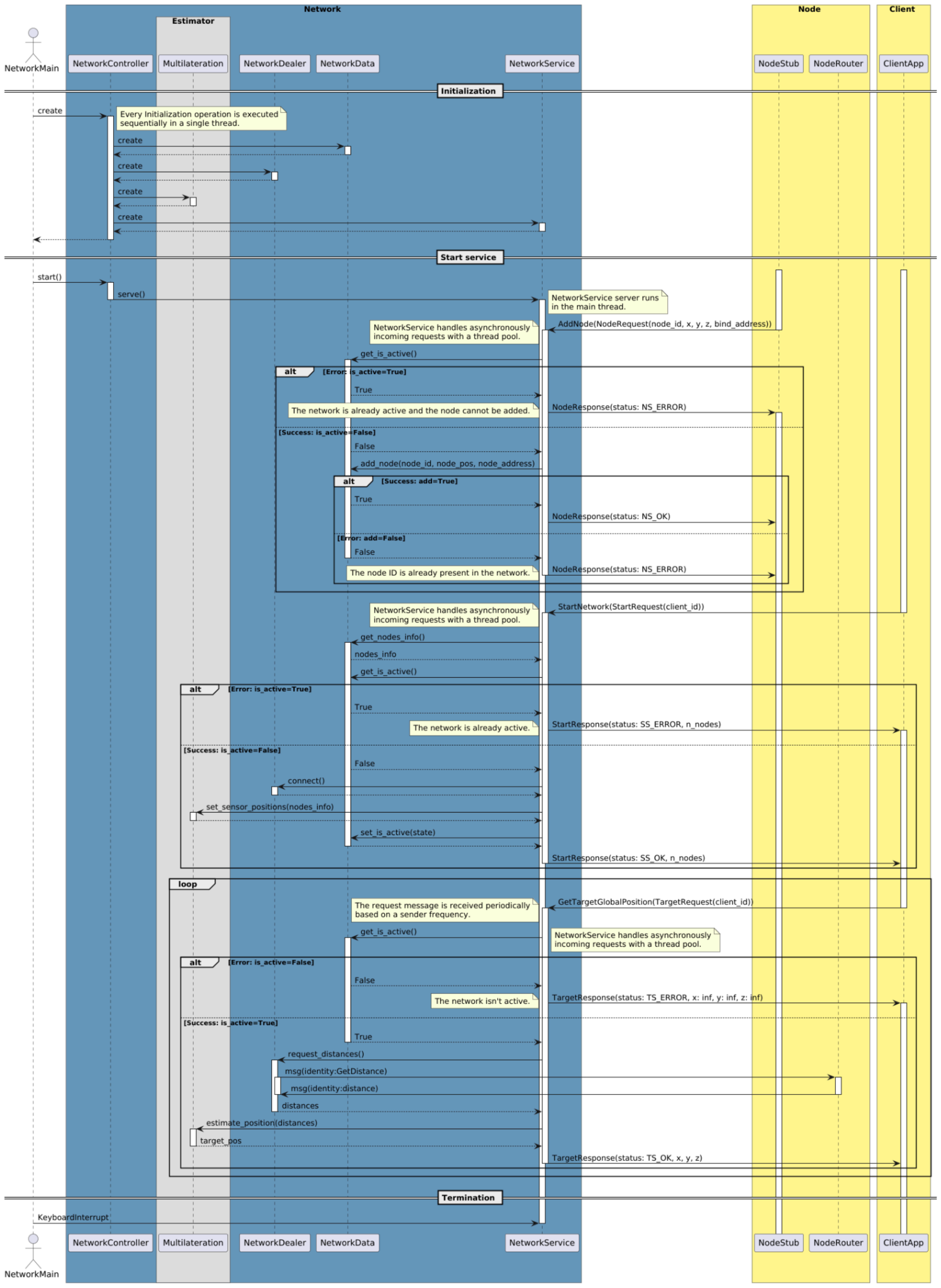
5.3.Node Component

This section focuses on the Node component, highlighting its interactions with other components in the system through sequence diagrams. For further details about the process, please refer to the related .puml file.


Node
Sensor
Network
Target
NodeMain
NodeController
SensorData
SensorController
SensorUpdater
NodeStub
NodeRouter
NetworkService
NetworkDealer
TargetService
Initialization
create
Every Initialization operation is executed sequentially in a single thread.
create
create
create
create
create
Start service
start()
start()
start()
SensorUpdater is a daemon thread which terminates when the main thread stops.
loop
The measurement is performed at a specified frequency.
GetPosition(GetPositionRequest(node_id))
GetPositionResponse (status: PS_OK, x, y, z)
_compute_distance()
set_distance(new_distance)
add_node_to_network()
AddNode(NodeRequest(node_id, x, y, z, bind_address))
alt
[Success: is_active=False]
NodeResponse (status: NS_OK)
True
start()
This operation runs in the main thread.
loop
The request message is received periodically based on a sender frequency.
msg(identity:GetDistance)
get_distance()
get_distance()
distance
distance
msg(identity:distance)
[Error: is_active=True]
The network is already active and the node cannot be added.
NodeResponse (status: NS_ERROR)
Termination
KeyboardInterrupt

5.4.Network Component

This section focuses on the Network component, highlighting its interactions with other components in the system through sequence diagrams. For further details about the process, please refer to the related .puml file.


Network
Estimator
Node
Client
NetworkMain
NetworkController
Multilateration
NetworkDealer
NetworkData
NetworkService
NodeStub
NodeRouter
ClientApp
Initialization
create
Every Initialization operation is executed sequentially in a single thread.
create
create
create
create
Start service
start()
serve()
NetworkService server runs in the main thread.
AddNode(NodeRequest(node_id, x, y, z, bind_address))
NetworkService handles asynchronously incoming requests with a thread pool.
get_is_active()
alt
[Error: is_active=True]
True
The network is already active and the node cannot be added.
NodeResponse(status: NS_ERROR)
[Success: is_active=False]
False
add_node(node_id, node_pos, node_address)
alt
[Success: add=True]
True
NodeResponse(status: NS_OK)
[Error: add=False]
False
The node ID is already present in the network.
NodeResponse(status: NS_ERROR)
StartNetwork(StartRequest(client_id))
NetworkService handles asynchronously incoming requests with a thread pool.
get_nodes_info()
nodes_info
get_is_active()
alt
[Error: is_active=True]
True
The network is already active.
StartResponse(status: SS_ERROR, n_nodes)
[Success: is_active=False]
False
connect()
set_sensor_positions(nodes_info)
set_is_active(state)
StartResponse(status: SS_OK, n_nodes)
loop
GetTargetGlobalPosition(TargetRequest(client_id))
The request message is received periodically based on a sender frequency.
get_is_active()
NetworkService handles asynchronously incoming requests with a thread pool.
alt
[Error: is_active=False]
False
The network isn't active.
TargetResponse(status: TS_ERROR, x: inf, y: inf, z: inf)
[Success: is_active=True]
True
request_distances()
msg(identity:GetDistance)
msg(identity:distance)
distances
estimate_position(distances)
target_pos
TargetResponse(status: TS_OK, x, y, z)
Termination
KeyboardInterrupt