RoMi-H
Enabled

Robotics Middleware for Healthcare (RoMi-H)

Traffic Management and Negotiation
Sharing of Assets by Robots

3rd December 2021

Centre for
Healthcare Assistive
& Robotics Technology
CHART

Challenges in Multi-fleet Deployment in Healthcare

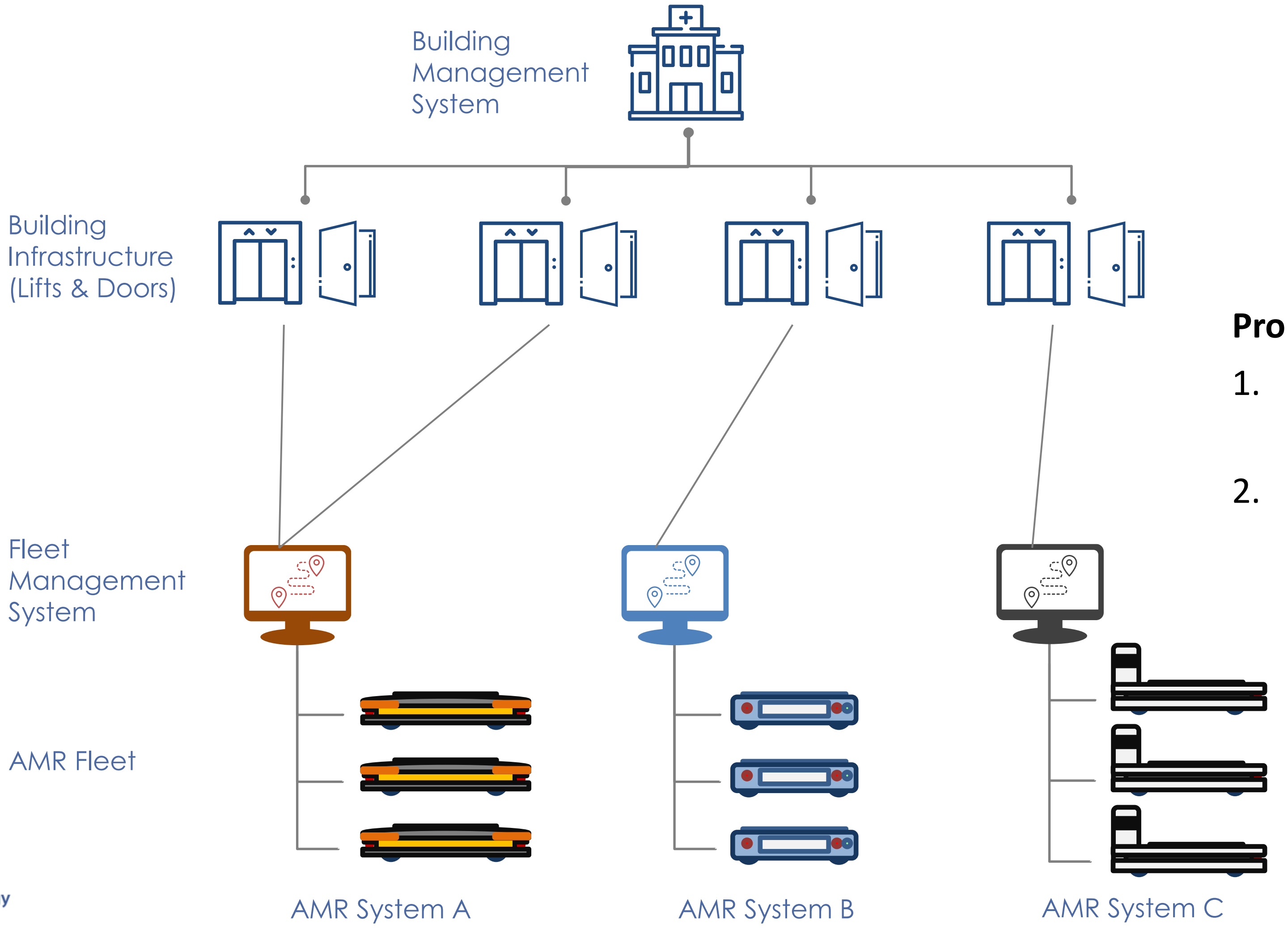
Lack of Interoperability between multiple proprietary systems

- Lack of robot-to-robot communication (handshaking to avoid collision and info on robot routing and status)
- Similarly for IoT solutions, non-consensus of communication protocols complicates and hinders data aggregation

Physical and infrastructural constrains

- Need for mobile robots to interface with lifts and doors
- Dedicated lifts and routes assigned to a single fleet of robots
- Multiple charging and docking sites for robots
- Large footprint with heavy infrastructure requirements

Current State of Multi-fleet deployment



Building Management System

Building Infrastructure (Lifts & Doors)

Fleet Management System

AMR Fleet

AMR System A

AMR System B

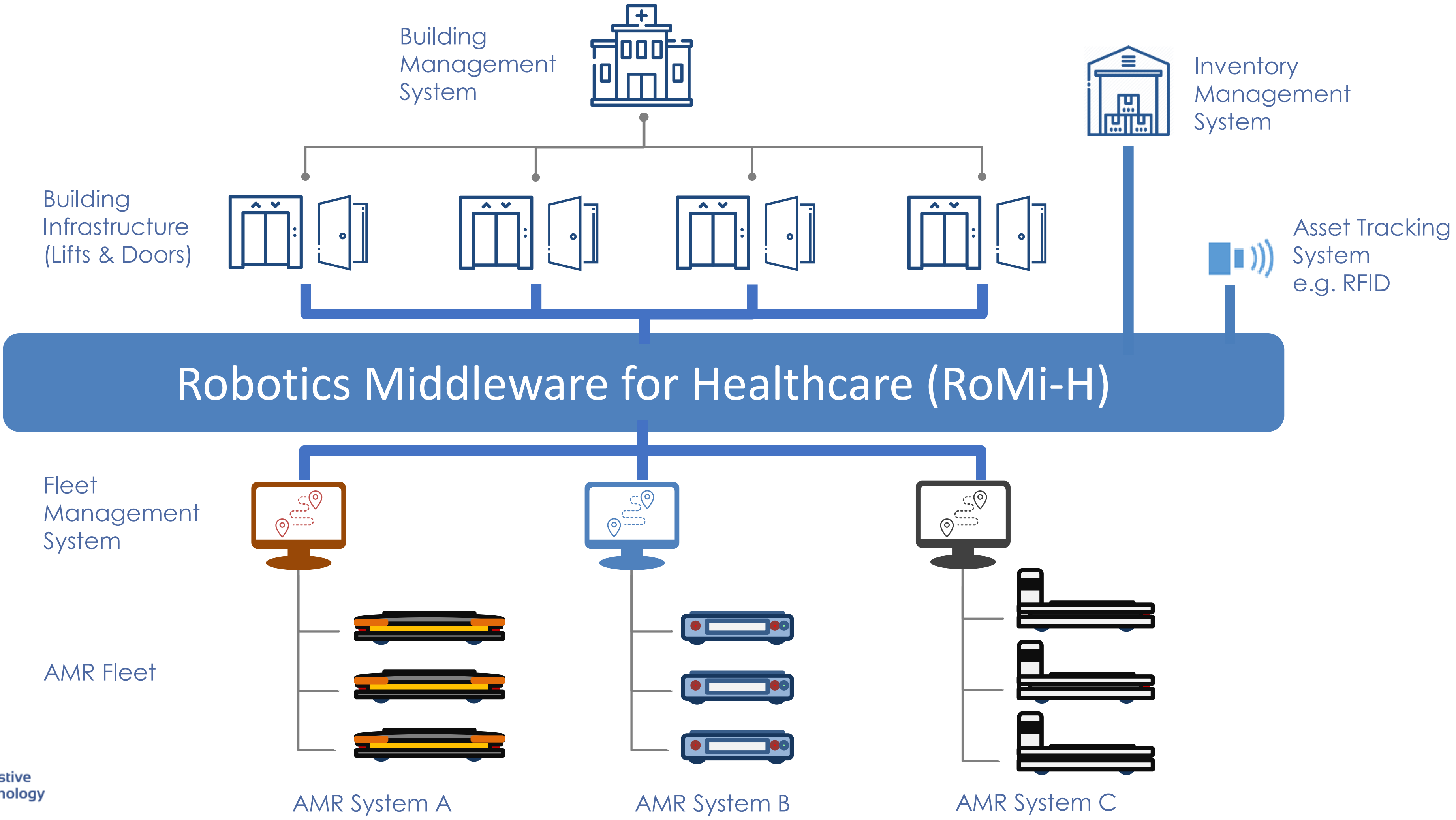
AMR System C

Problems faced

1. Duplication in integration efforts and increased integration cost
2. Lack of ability for resource optimisation (Usage of lifts, and corridors)

CHART Centre for Healthcare Assistive & Robotics Technology

RoMi-H for Multi-fleet deployment

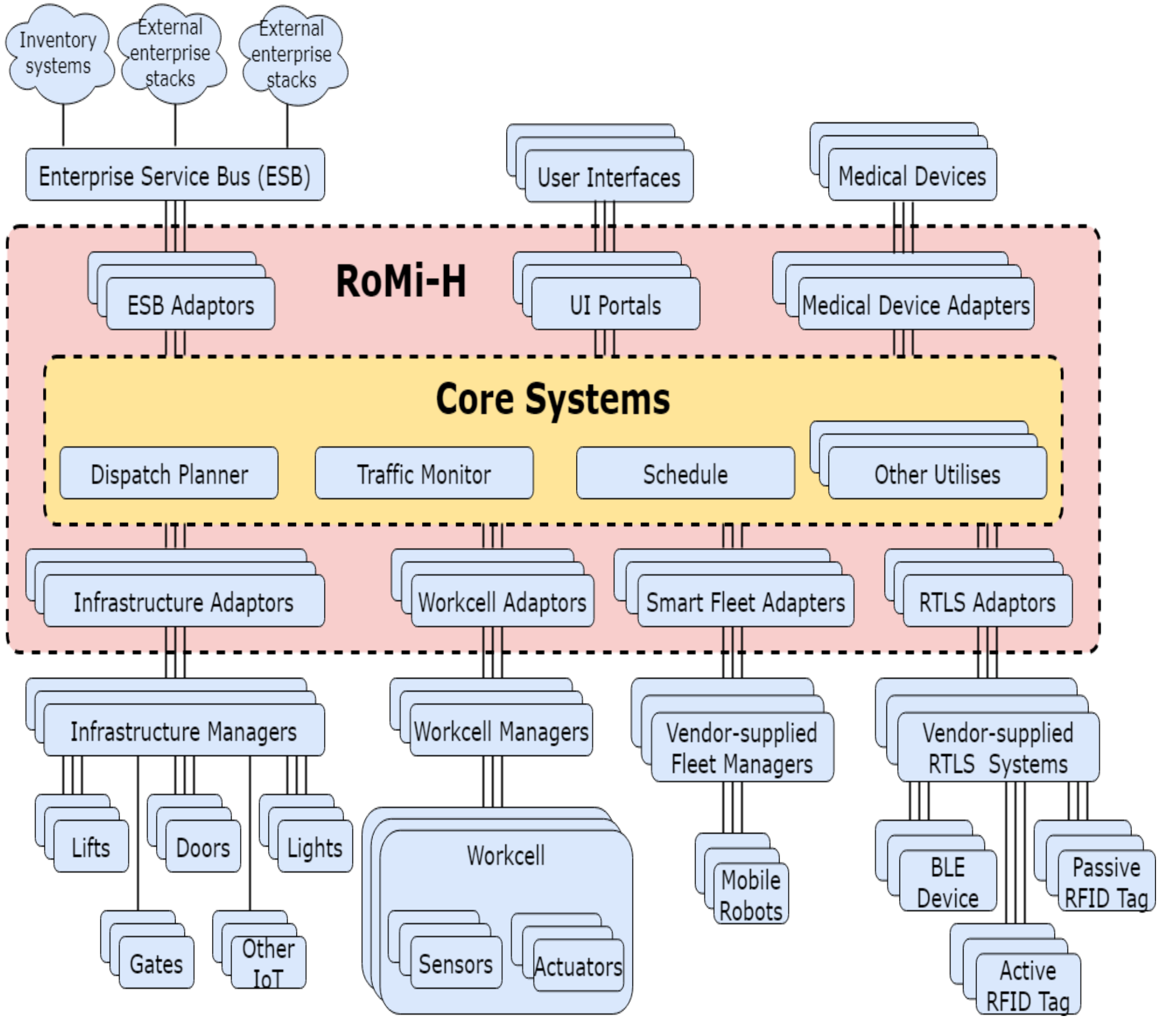


Centre for
Healthcare Assistive
& Robotics Technology

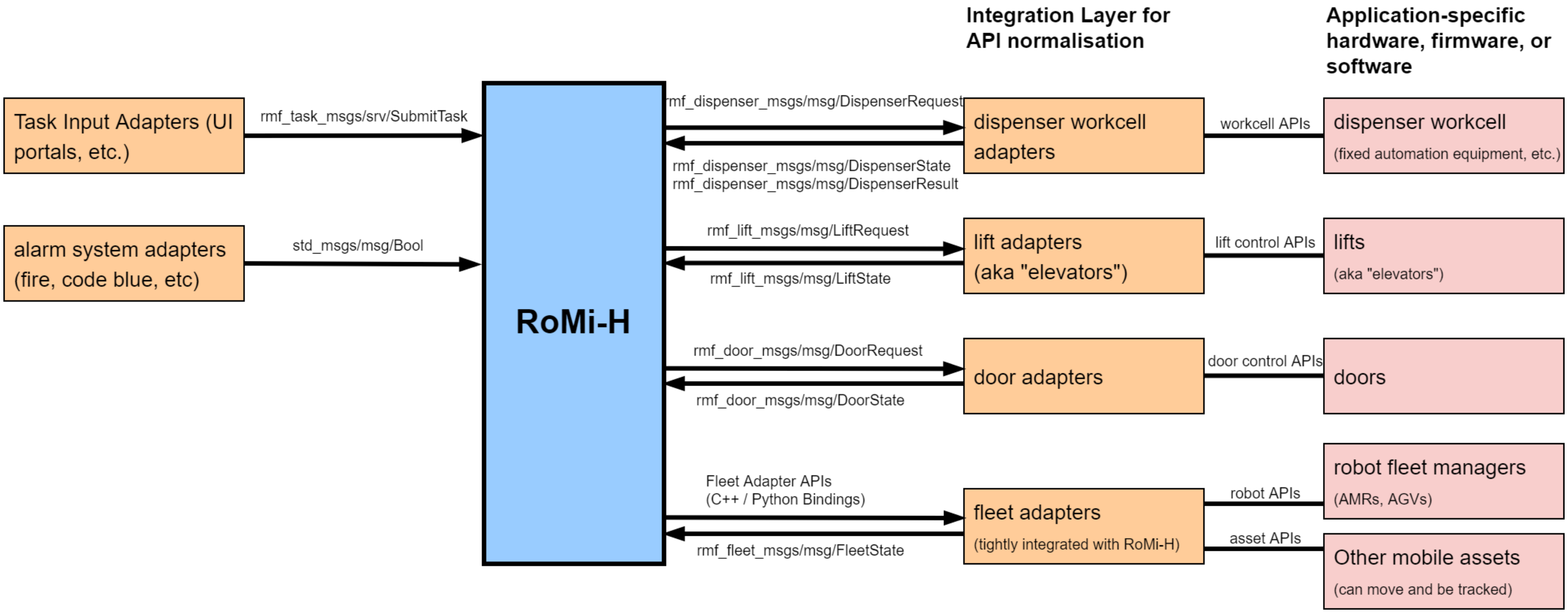
CHART

Robotics Middleware for Healthcare (RoMi-H)

- RoMi-H is a Robotic Middleware Framework comprising of a collection of libraries and tools that facilitate interoperability among:
  - Heterogeneous robot fleets with different OSes
  - Smart building & infrastructure (including Lift & Doors)
  - Automation Systems (e.g. dispenser, pick & place robots)
- Provides visibility of status of inter-connected systems
- Adds intelligence to the overall interconnected system through resource allocation and de-conflict shared resources



RoMi-H Standardised Interfaces and Messages



Integration Layer for API normalisation

Application-specific hardware, firmware, or software

Task Input Adapters (UI portals, etc.)

rmf_task_msgs/srv/SubmitTask

alarm system adapters (fire, code blue, etc)

std_msgs/msg/Bool

RoMi-H

rmf_dispenser_msgs/msg/DispenserRequest

rmf_dispenser_msgs/msg/DispenserState
rmf_dispenser_msgs/msg/DispenserResult

dispenser workcell adapters

workcell APIs

dispenser workcell
(fixed automation equipment, etc.)

rmf_lift_msgs/msg/LiftRequest

rmf_lift_msgs/msg/LiftState

lift adapters
(aka "elevators")

lift control APIs

lifts
(aka "elevators")

rmf_door_msgs/msg/DoorRequest

rmf_door_msgs/msg/DoorState

door adapters

door control APIs

doors

Fleet Adapter APIs
(C++ / Python Bindings)

rmf_fleet_msgs/msg/FleetState

fleet adapters
(tightly integrated with RoMi-H)

robot APIs

robot fleet managers
(AMRs, AGVs)

asset APIs

Other mobile assets
(can move and be tracked)

RoMi-H Compliance Level

<table>
<tr><th>High</th><th>Medium</th></tr>
<tr><td colspan="2">Able to provide robot’s location</td></tr>
<tr><td colspan="2">Able to be told to pause/resume current mission</td></tr>
<tr><td>Able to accept external issued destination goals and waypoints</td><td>Only able to provide destination goals and waypoints</td></tr>
</table>

Fleet Adapter API Key Classes

Critical Classes	**Description**
Adapter	Initialises and maintains communication with the other core RMF systems. Use this to register one or more fleets and receive a FleetUpdateHandle for each fleet.
FleetUpdateHandle	Allows you to configure a fleet by adding robots and specifying settings for the fleet (e.g. specifying what types of deliveries the fleet can perform). New robots can be added to the fleet at any time.
RobotUpdateHandle	Use this to update the position of a robot and to notify the adapter if the robot's progress gets interrupted.
RobotCommandHandle	This is a pure abstract interface class. The functions of this class must be implemented to call upon the API of the specific fleet manager that is being adapted.
EasyTrafficLight	This is a simplified API for medium compliance fleets to receive moving and waiting instructions from RoMi-H. This is also used to update the current position and path of a robot.

Integrating Robot APIs into Fleet Adapters

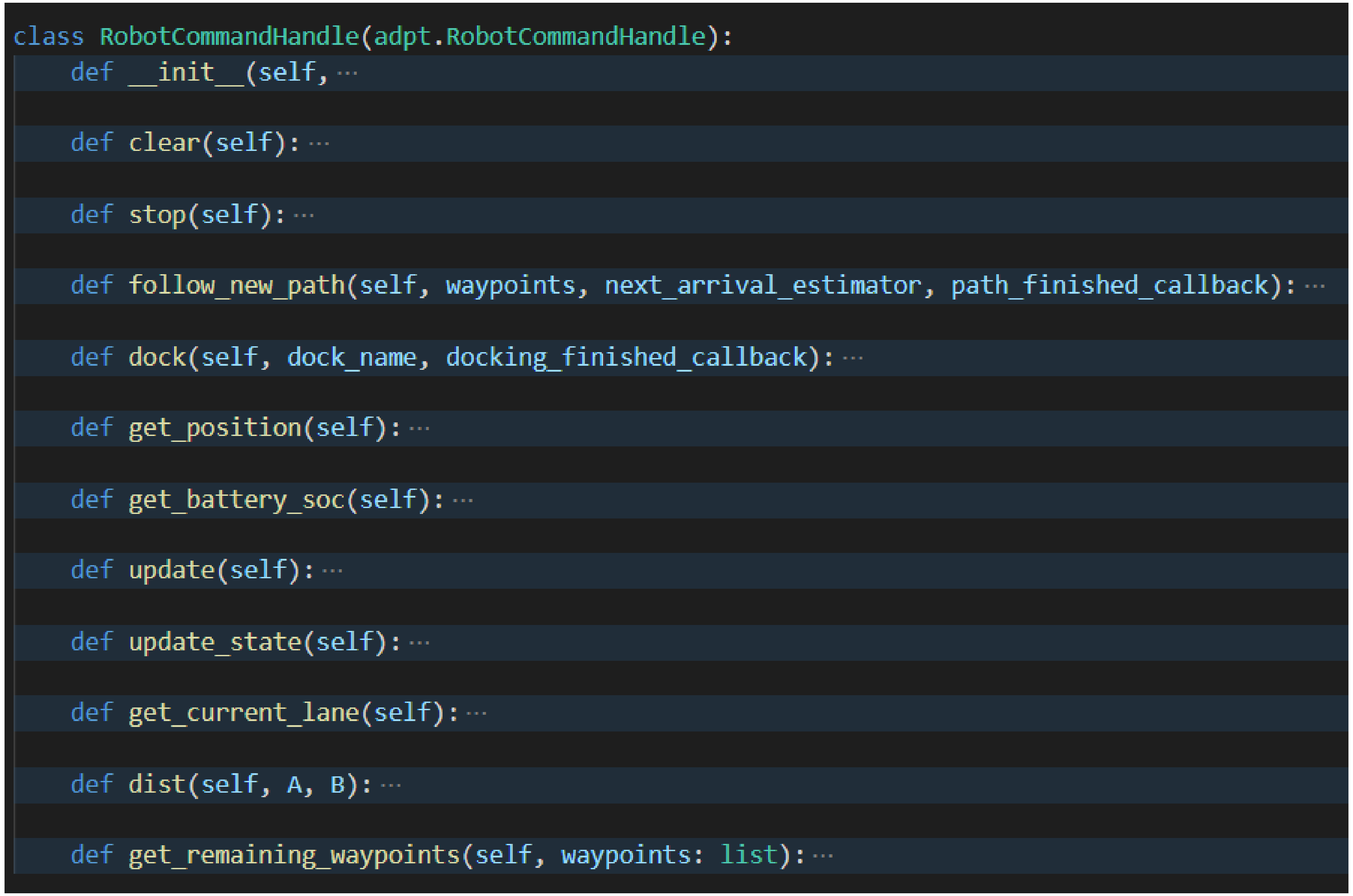
High Compliance	
RobotCommandHandle::follow_new_path()	The Robot API to command a robot to a specific location is to be placed in this function.
RobotCommandHandle::stop()	The Robot API to command a robot to stop all actions/missions immediately is to be placed in this function.
RobotUpdateHandle::update_position() RobotUpdateHandle::update_battery_soc()	These two function can help to update the required robot states. Functions can be implemented in a timer callback, Robot APIs are to be used to provide the necessary information. Timer can be implemented as part of the RobotCommandHandle
Medium Compliance	
pause() & resume() functions	Respective Robot API commands are to be included in the respective functions. The functions are required arguments inputs for the EasyTrafficLight class.

High Compliance Example (Python)

```python
class RobotAPI:
    # The constructor below accepts parameters typically required to submit
    # http requests. Users should modify the constructor as per the
    # requirements of their robot's API
    def __init__(self, prefix: str, user: str, password: str):
        self.prefix = prefix
        self.user = user
        self.password = password
        self.connected = False
        # Test connectivity
        connected = self.check_connection()
        if connected:
            print("Successfully able to query API server")
            self.connected = True
        else:
            print("Unable to query API server")

    def check_connection(self): ···

    def position(self): ···

    def navigate(self, pose, map_name: str): ···

    def start_process(self, process: str, map_name: str): ···

    def stop(self): ···

    def navigation_remaining_duration(self): ···

    def navigation_completed(self): ···

    def process_completed(self): ···

    def battery_soc(self): ···
```

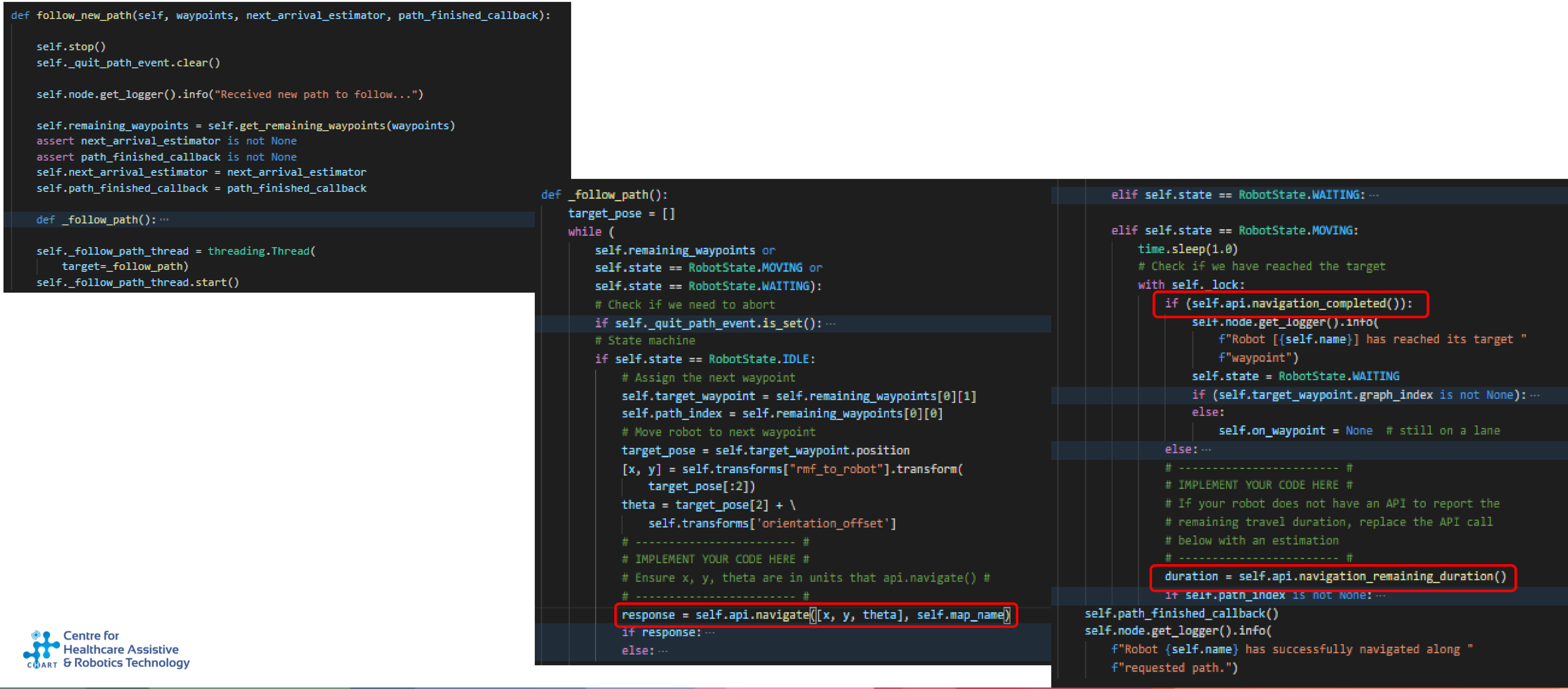
check_connection()	Return True if connection to the robot API server is successful
position()	Return [x, y, theta] expressed in the robot's coordinate frame or ‘None’ if any errors are encountered
navigate()	Request the robot to navigate to pose: [x, y, theta] where x, y and theta are in the robot's coordinate convention. This function should return True if the robot has accepted the request, else False
start_process()	Request the robot to begin a process. This is specific to the robot and the use case. For example, load/unload a cart for Deliverybot or begin cleaning a zone for a cleaning robot. Return True if the robot has accepted the request, else False
stop()	Command the robot to stop. Return True if robot has successfully stopped, else False
navigation_remaining_duration()	Return the number of seconds remaining for the robot to reach its destination Return True if the robot has successfully completed its previous navigation request, else False
navigation_completed()	Return True if the robot has successfully completed its previous navigation request, else False.
process_completed()	Return True if the robot has successfully completed its previous process request, else False
battery_soc()	Return the state of the charge of the robot as a value between 0.0 and 1.0, else return ‘None’ if any errors are encountered

High Compliance Example (Python)



```python
class RobotCommandHandle(adpt.RobotCommandHandle):
    def __init__(self, ···

    def clear(self): ···

    def stop(self): ···

    def follow_new_path(self, waypoints, next_arrival_estimator, path_finished_callback): ···

    def dock(self, dock_name, docking_finished_callback): ···

    def get_position(self): ···

    def get_battery_soc(self): ···

    def update(self): ···

    def update_state(self): ···

    def get_current_lane(self): ···

    def dist(self, A, B): ···

    def get_remaining_waypoints(self, waypoints: list): ···
```

High Compliance Example (Python)

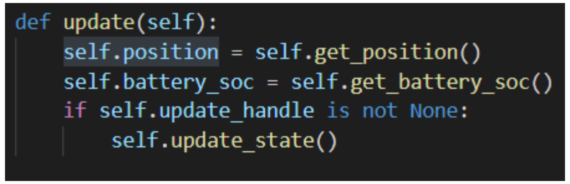


```python
def follow_new_path(self, waypoints, next_arrival_estimator, path_finished_callback):

    self.stop()
    self._quit_path_event.clear()

    self.node.get_logger().info("Received new path to follow...")

    self.remaining_waypoints = self.get_remaining_waypoints(waypoints)
    assert next_arrival_estimator is not None
    assert path_finished_callback is not None
    self.next_arrival_estimator = next_arrival_estimator
    self.path_finished_callback = path_finished_callback

    def _follow_path():···

    self._follow_path_thread = threading.Thread(
        target=_follow_path)
    self._follow_path_thread.start()
```

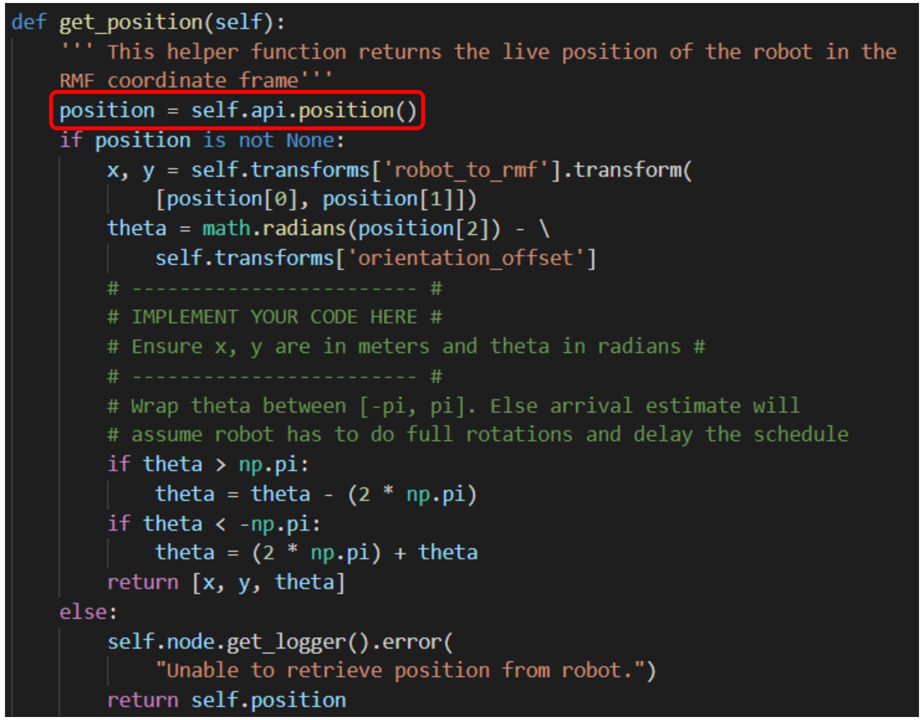
```python
def _follow_path():
    target_pose = []
    while (
        self.remaining_waypoints or
        self.state == RobotState.MOVING or
        self.state == RobotState.WAITING):
        # Check if we need to abort
        if self._quit_path_event.is_set():···
        # State machine
        if self.state == RobotState.IDLE:
            # Assign the next waypoint
            self.target_waypoint = self.remaining_waypoints[0][1]
            self.path_index = self.remaining_waypoints[0][0]
            # Move robot to next waypoint
            target_pose = self.target_waypoint.position
            [x, y] = self.transforms["rmf_to_robot"].transform(
                target_pose[:2])
            theta = target_pose[2] + \
                self.transforms['orientation_offset']
            # ------------------------ #
            # IMPLEMENT YOUR CODE HERE #
            # Ensure x, y, theta are in units that api.navigate() #
            # ------------------------ #
            response = self.api.navigate([x, y, theta], self.map_name)
            if response:···
            else:···
```

```python
        elif self.state == RobotState.WAITING:···

        elif self.state == RobotState.MOVING:
            time.sleep(1.0)
            # Check if we have reached the target
            with self._lock:
                if (self.api.navigation_completed()):
                    self.node.get_logger().info(
                        f"Robot [{self.name}] has reached its target "
                        f"waypoint")
                    self.state = RobotState.WAITING
                    if (self.target_waypoint.graph_index is not None):···
                    else:
                        self.on_waypoint = None  # still on a lane
                else:···
                # ------------------------ #
                # IMPLEMENT YOUR CODE HERE #
                # If your robot does not have an API to report the
                # remaining travel duration, replace the API call
                # below with an estimation
                # ------------------------ #
                duration = self.api.navigation_remaining_duration()
                if self.path_index is not None:···
    self.path_finished_callback()
    self.node.get_logger().info(
        f"Robot {self.name} has successfully navigated along "
        f"requested path.")
```

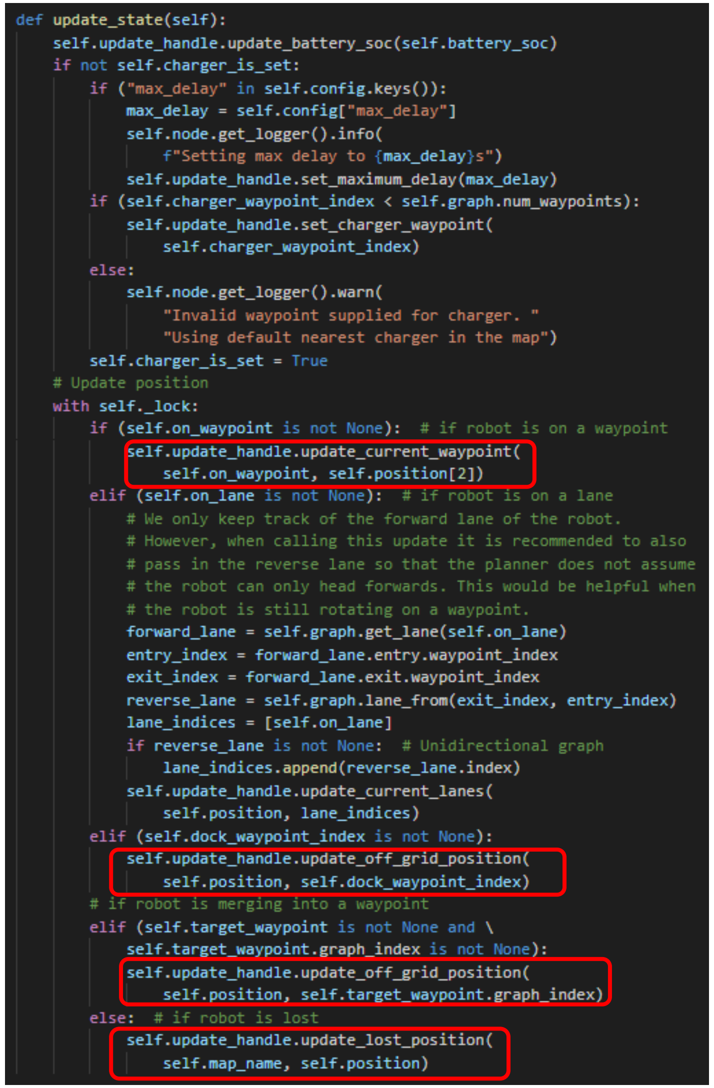
High Compliance Example (Python)



```python
def update(self):
    self.position = self.get_position()
    self.battery_soc = self.get_battery_soc()
    if self.update_handle is not None:
        self.update_state()
```



```python
def get_position(self):
    ''' This helper function returns the live position of the robot in the
    RMF coordinate frame'''
    position = self.api.position()
    if position is not None:
        x, y = self.transforms['robot_to_rmf'].transform(
            [position[0], position[1]])
        theta = math.radians(position[2]) - \
            self.transforms['orientation_offset']
        # ------------------------ #
        # IMPLEMENT YOUR CODE HERE #
        # Ensure x, y are in meters and theta in radians #
        # ------------------------ #
        # Wrap theta between [-pi, pi]. Else arrival estimate will
        # assume robot has to do full rotations and delay the schedule
        if theta > np.pi:
            theta = theta - (2 * np.pi)
        if theta < -np.pi:
            theta = (2 * np.pi) + theta
        return [x, y, theta]
    else:
        self.node.get_logger().error(
            "Unable to retrieve position from robot.")
        return self.position
```



```python
def update_state(self):
    self.update_handle.update_battery_soc(self.battery_soc)
    if not self.charger_is_set:
        if ("max_delay" in self.config.keys()):
            max_delay = self.config["max_delay"]
            self.node.get_logger().info(
                f"Setting max delay to {max_delay}s")
            self.update_handle.set_maximum_delay(max_delay)
        if (self.charger_waypoint_index < self.graph.num_waypoints):
            self.update_handle.set_charger_waypoint(
                self.charger_waypoint_index)
        else:
            self.node.get_logger().warn(
                "Invalid waypoint supplied for charger. "
                "Using default nearest charger in the map")
        self.charger_is_set = True
    # Update position
    with self._lock:
        if (self.on_waypoint is not None):  # if robot is on a waypoint
            self.update_handle.update_current_waypoint(
                self.on_waypoint, self.position[2])
        elif (self.on_lane is not None):  # if robot is on a lane
            # We only keep track of the forward lane of the robot.
            # However, when calling this update it is recommended to also
            # pass in the reverse lane so that the planner does not assume
            # the robot can only head forwards. This would be helpful when
            # the robot is still rotating on a waypoint.
            forward_lane = self.graph.get_lane(self.on_lane)
            entry_index = forward_lane.entry.waypoint_index
            exit_index = forward_lane.exit.waypoint_index
            reverse_lane = self.graph.lane_from(exit_index, entry_index)
            lane_indices = [self.on_lane]
            if reverse_lane is not None:  # Unidirectional graph
                lane_indices.append(reverse_lane.index)
            self.update_handle.update_current_lanes(
                self.position, lane_indices)
        elif (self.dock_waypoint_index is not None):
            self.update_handle.update_off_grid_position(
                self.position, self.dock_waypoint_index)
        # if robot is merging into a waypoint
        elif (self.target_waypoint is not None and \
            self.target_waypoint.graph_index is not None):
            self.update_handle.update_off_grid_position(
                self.position, self.target_waypoint.graph_index)
        else:  # if robot is lost
            self.update_handle.update_lost_position(
                self.map_name, self.position)
```

RoMi-H Traffic Management and Negotiation

Smart Fleet Adapter A

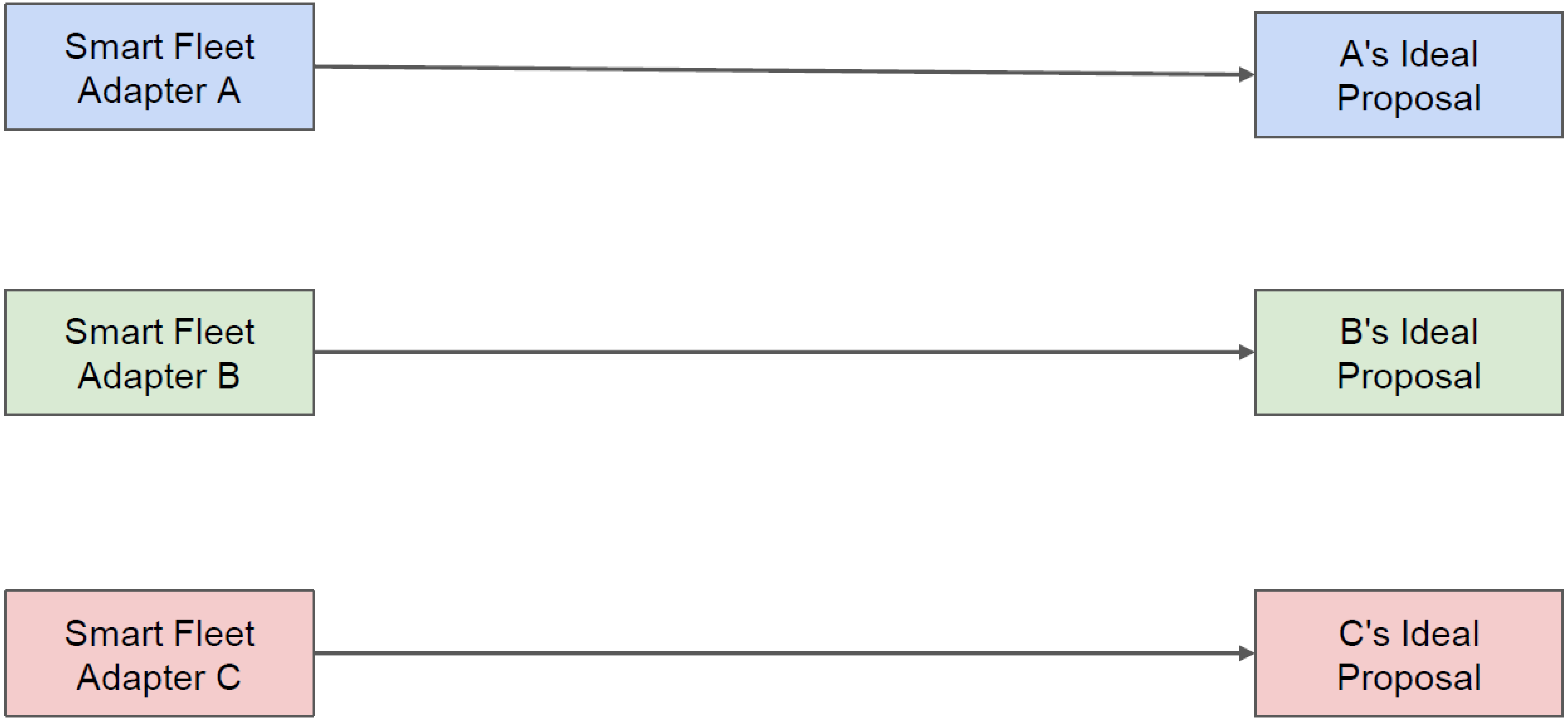
Smart Fleet Adapter B

Smart Fleet Adapter C

Assumptions

- Each fleet does not know what the others are capable of
- Each fleet can communicate a plan that is feasible for itself
- Each fleet can see the other’s plans and attempt to plan around it

RoMi-H Traffic Management and Negotiation



Smart Fleet Adapter A

A's Ideal Proposal

Smart Fleet Adapter B

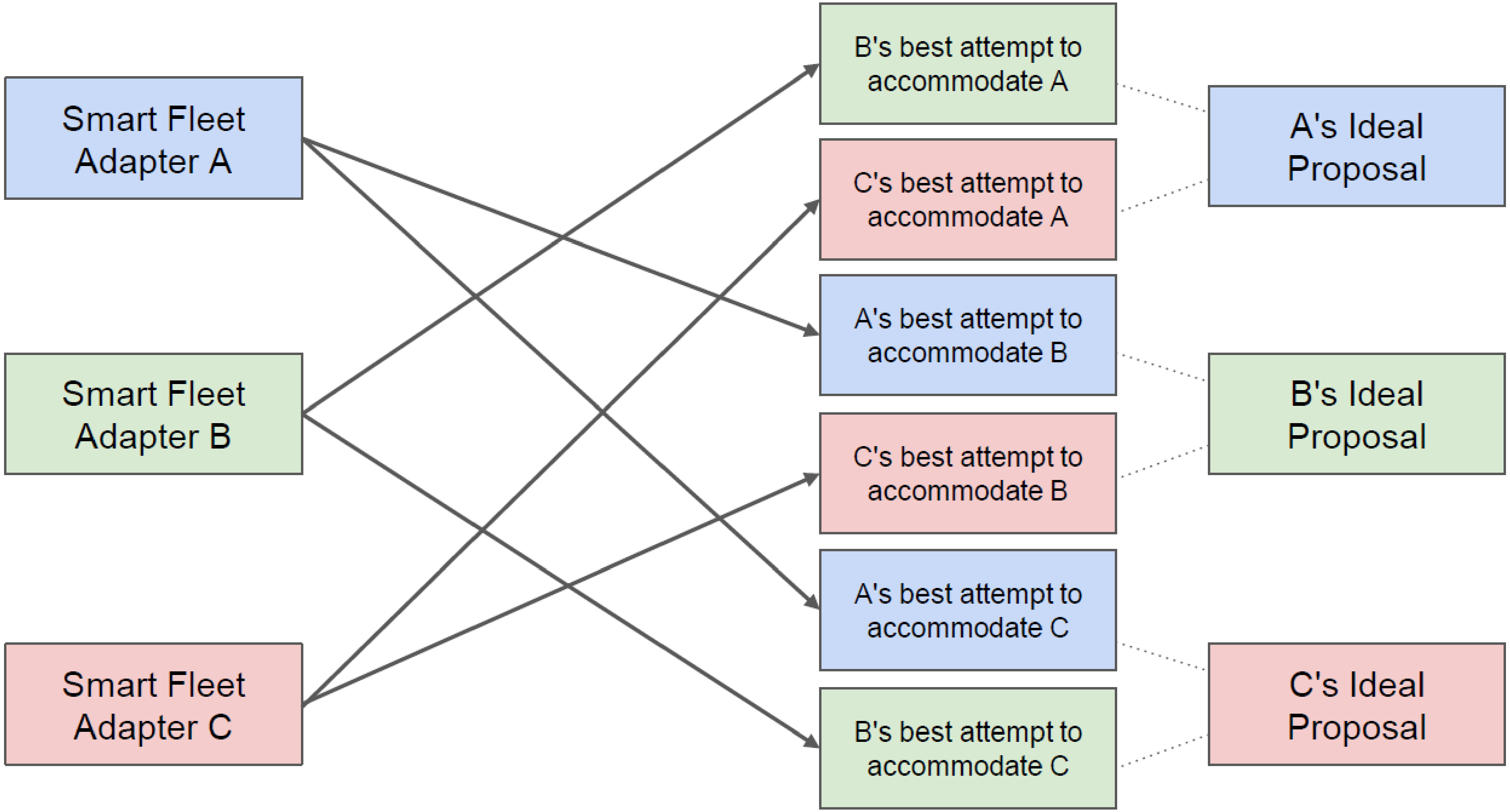
B's Ideal Proposal

Smart Fleet Adapter C

C's Ideal Proposal

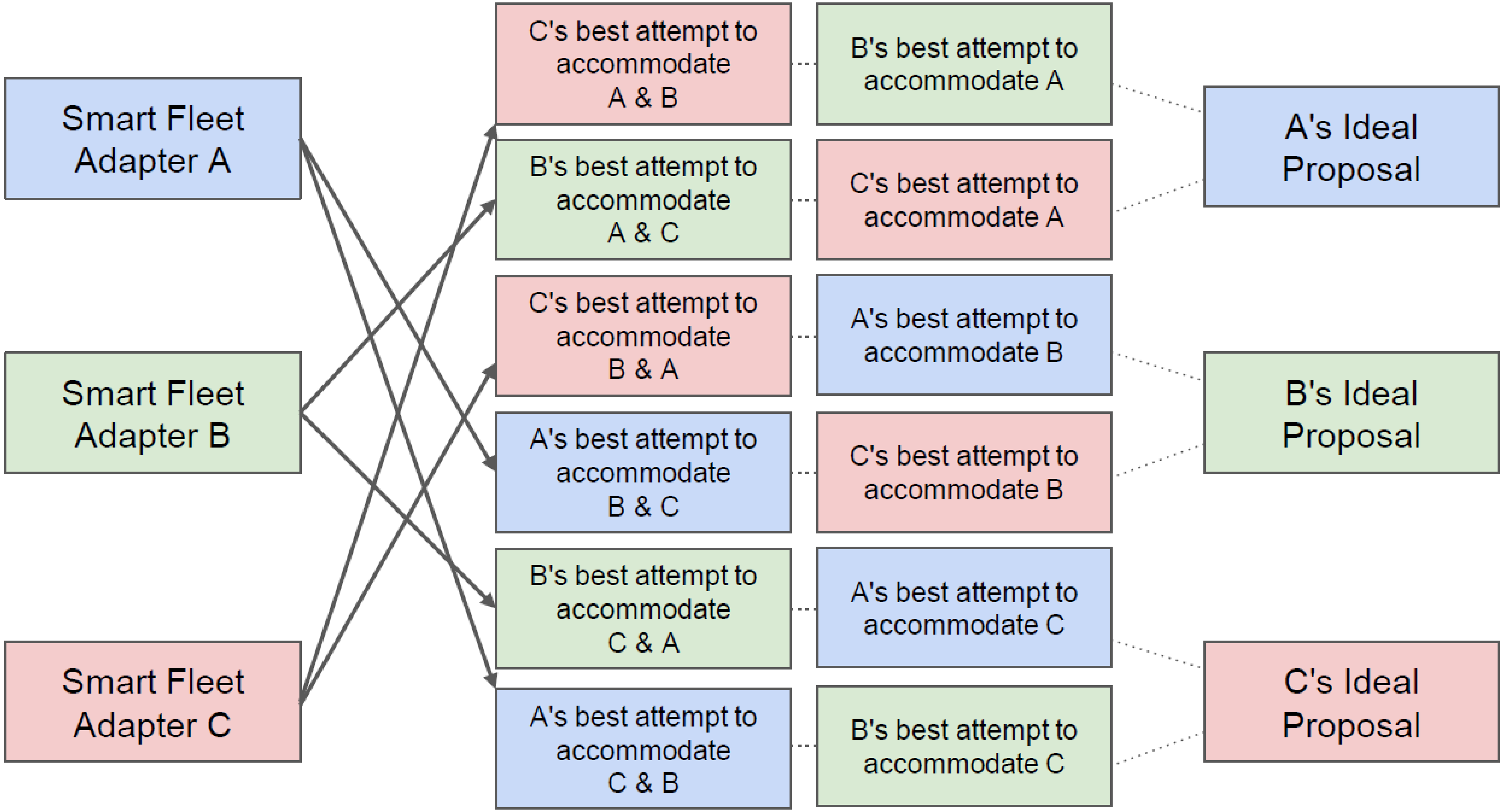
Each Fleet will propose the itinerary they would like to follow

RoMi-H Traffic Management and Negotiation



Each Fleet will respond to the ideal itineraries of the others with an itinerary that is feasible for itself while accommodating the other

RoMi-H Traffic Management and Negotiation



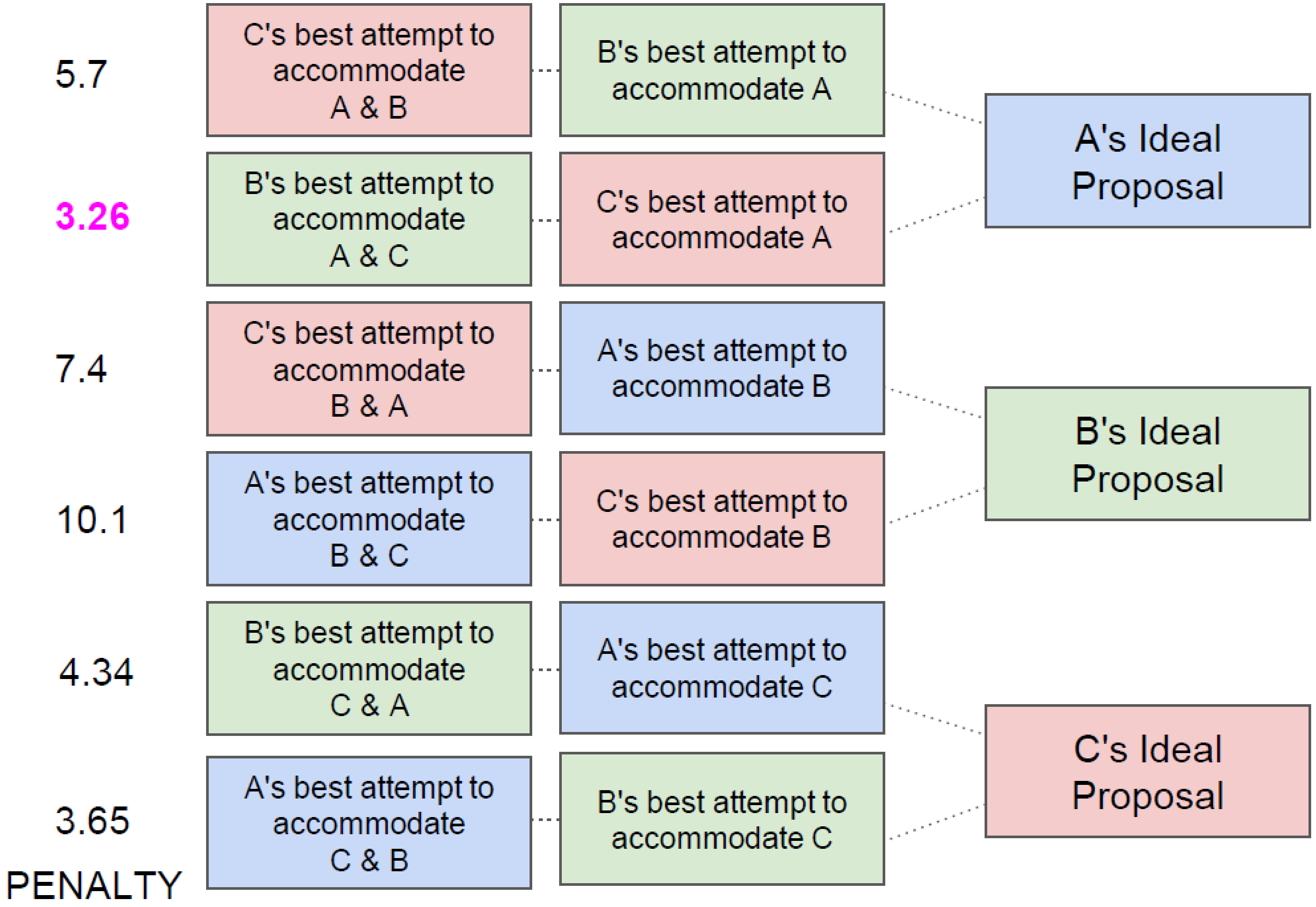
Each Fleet will then respond to each combination of the other's proposed itineraries with an itinerary that would be feasible for itself

RoMi-H Traffic Management and Negotiation

A third-party judge measures the penalty of each set of proposals.

The plan with the lowest penalty will be chosen.

The penalty may be measured by the sum of the delays in completing all of the tasks. The sum may be weighted by the importance of each task.



PENALTY		
5.7	C's best attempt to accommodate A & B	B's best attempt to accommodate A
3.26	B's best attempt to accommodate A & C	C's best attempt to accommodate A
7.4	C's best attempt to accommodate B & A	A's best attempt to accommodate B
10.1	A's best attempt to accommodate B & C	C's best attempt to accommodate B
4.34	B's best attempt to accommodate C & A	A's best attempt to accommodate C
3.65	A's best attempt to accommodate C & B	B's best attempt to accommodate C

A's Ideal Proposal

B's Ideal Proposal

C's Ideal Proposal

Video of RoMi-H Traffic Management and Negotiation at Expo Hall 10

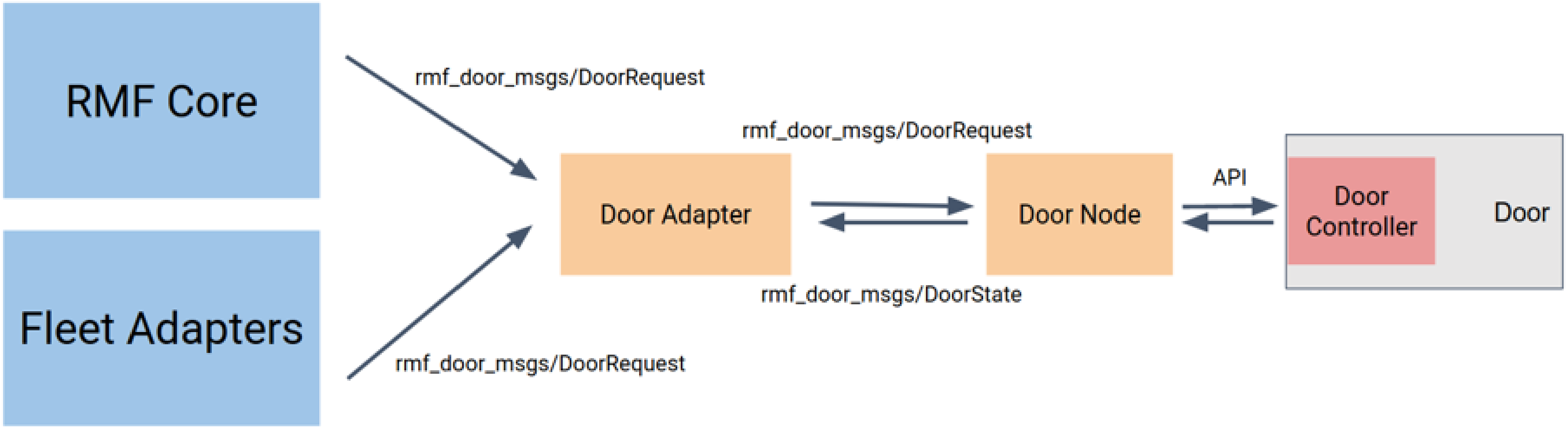
Centre for
Healthcare Assistive
& Robotics Technology

Challenges on Sharing Assets

- Non-unified communication protocol (AMR & different brand of lift/door)
- Need to re-develop adapter with lifts and doors for new brand of AMR
- Need to dedicate space for different types of chargers for different brand of AMR

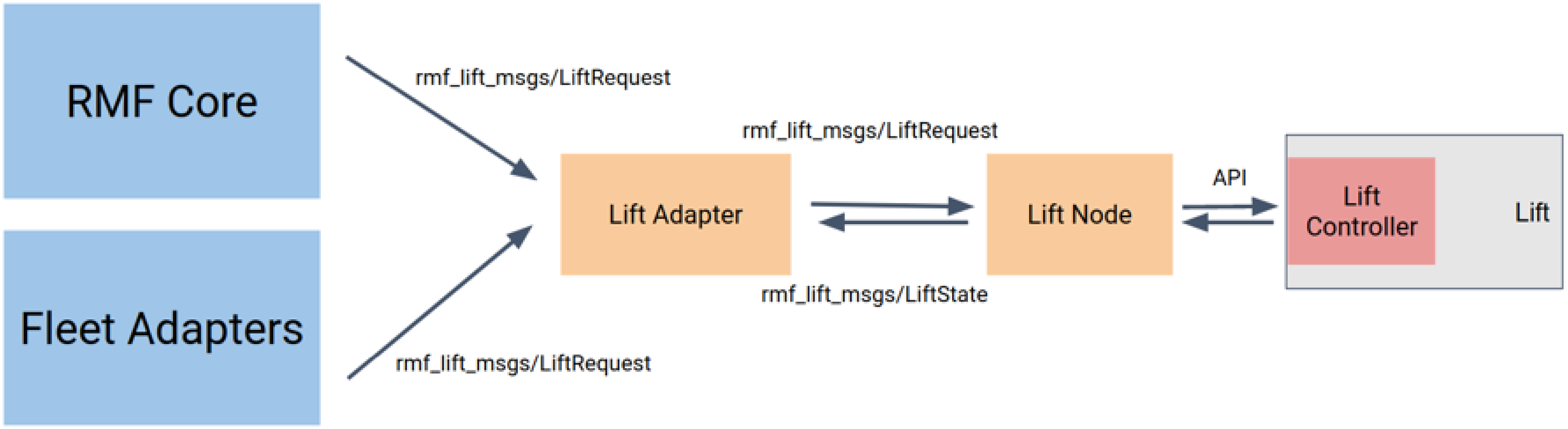
Common Infrastructure 1 ---- Door

- ‘Door Adapter’ acts like a state supervisor
- ‘Door Node’ acts like a translator, to translate RMF command to door controller command



Common Infrastructure 2 ---- Lift

- ‘Lift Adapter’ acts like a state supervisor
- ‘Lift Node’ acts like a translator, to translate RMF command to lift controller command



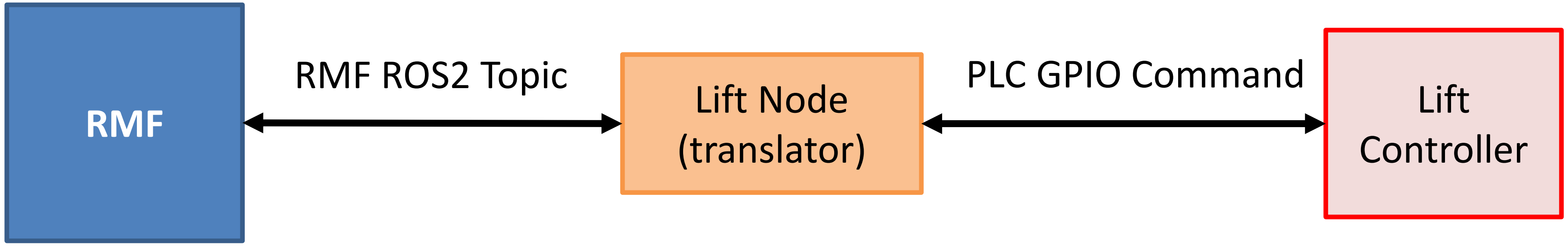
Message Exchange (RMF & Lift)

Message Types	ROS2 Topic	Description
rmf_lift_msgs/LiftState	/lift_states	State of the lift published by the door node
rmf_lift_msgs/LiftRequest	/lift_requests	Direct requests subscribed by the lift node and published by the lift adapter
rmf_lift_msgs/LiftRequest	/adapter_lift_requests	Requests to be sent to the lift adapter/supervisor to request safe operation of lifts

How ‘Lift Node’ works?

- Translate messages (RMF ←→ Low level Lift controller)

- Tasks:
  1. To obtain ‘lift state’ from lift controller and publish ROS2 topic ‘/lift_state’ to RMF
  2. To receive ROS2 topic ‘lift request’ from RMF and controls the signals of the GPIO pins on the MCU (send to lift controller)



RMF

RMF ROS2 Topic

Lift Node (translator)

PLC GPIO Command

Lift Controller

Lift Node Example (C++)

```
void DryContactLiftController::publish_lift_state_callback(){
  lift_state_message_->lift_time = ros_clock_.now();

  #ifdef BCM2835

  // Update lift's door state
  if ( bcm2835_gpio_lev(DOOR_STATUS) == 0)
    lift_state_message_->door_state = lift_state_message_->DOOR_OPEN;
  else if ( bcm2835_gpio_lev(DOOR_STATUS) == 1)
    lift_state_message_->door_state = lift_state_message_->DOOR_CLOSED;
  else // impossible
    lift_state_message_->door_state = lift_state_message_->DOOR_OPEN;

  #endif

  lift_state_message_->current_floor = get_current_floor();

  lift_state_publisher_->publish(*lift_state_message_);
}
```

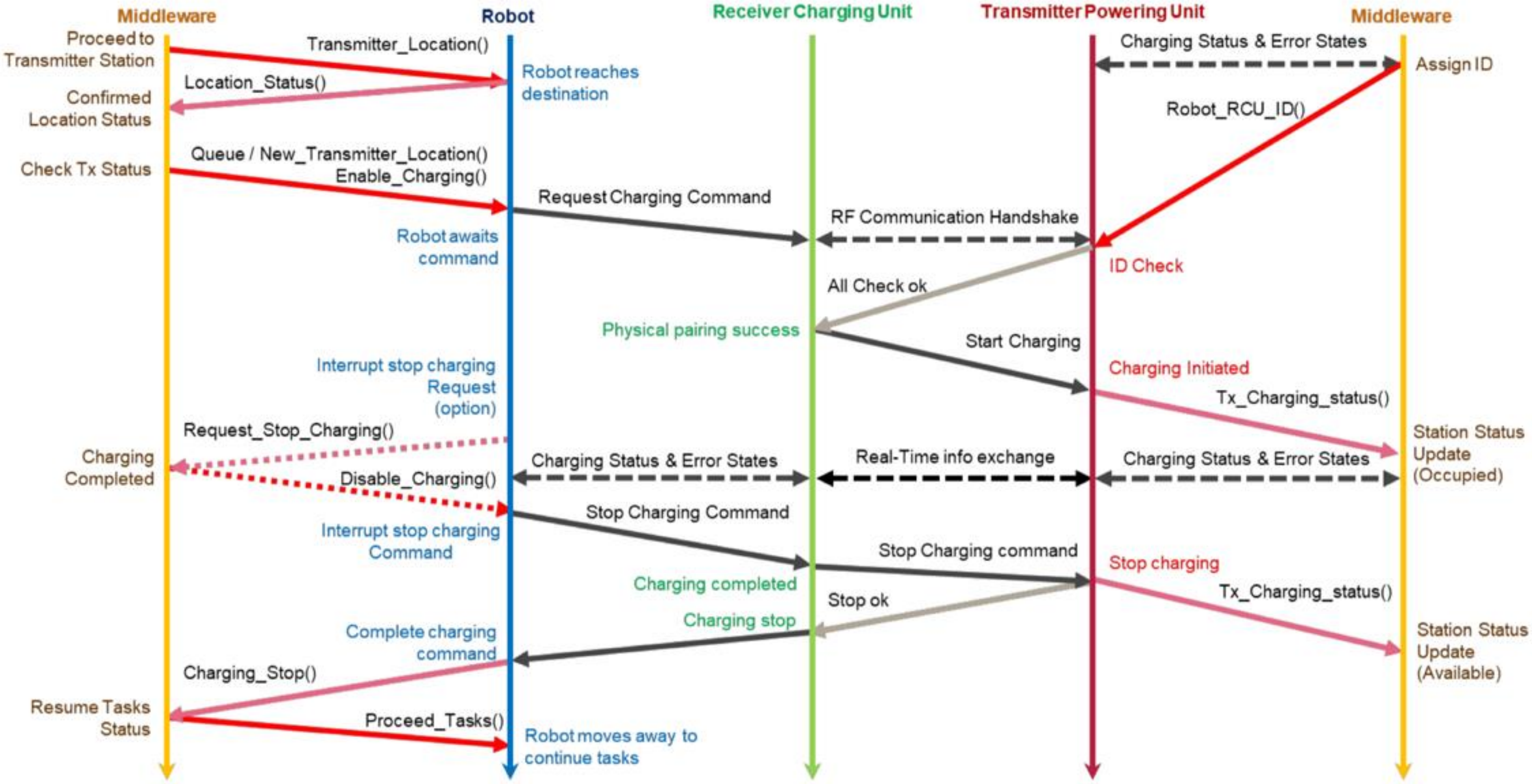
Reusability of Door/Lift Node (translator)

- One-time effort on developing ‘translator’ for same brand of door/lift
- Effortless for new brand of AMR to utilise the integrated shared infrastructure (eg. door & lift)

Common Infrastructure 3 ---- Charger

- Universal Wireless Charger
- To install an Universal Wireless Charging Receiver (RCU) on each AMR
- Different brand of AMR are able to charge at the same Universal Wireless Charging Station (TPU)
- Charging Station equips with height adjustable transmitter pad

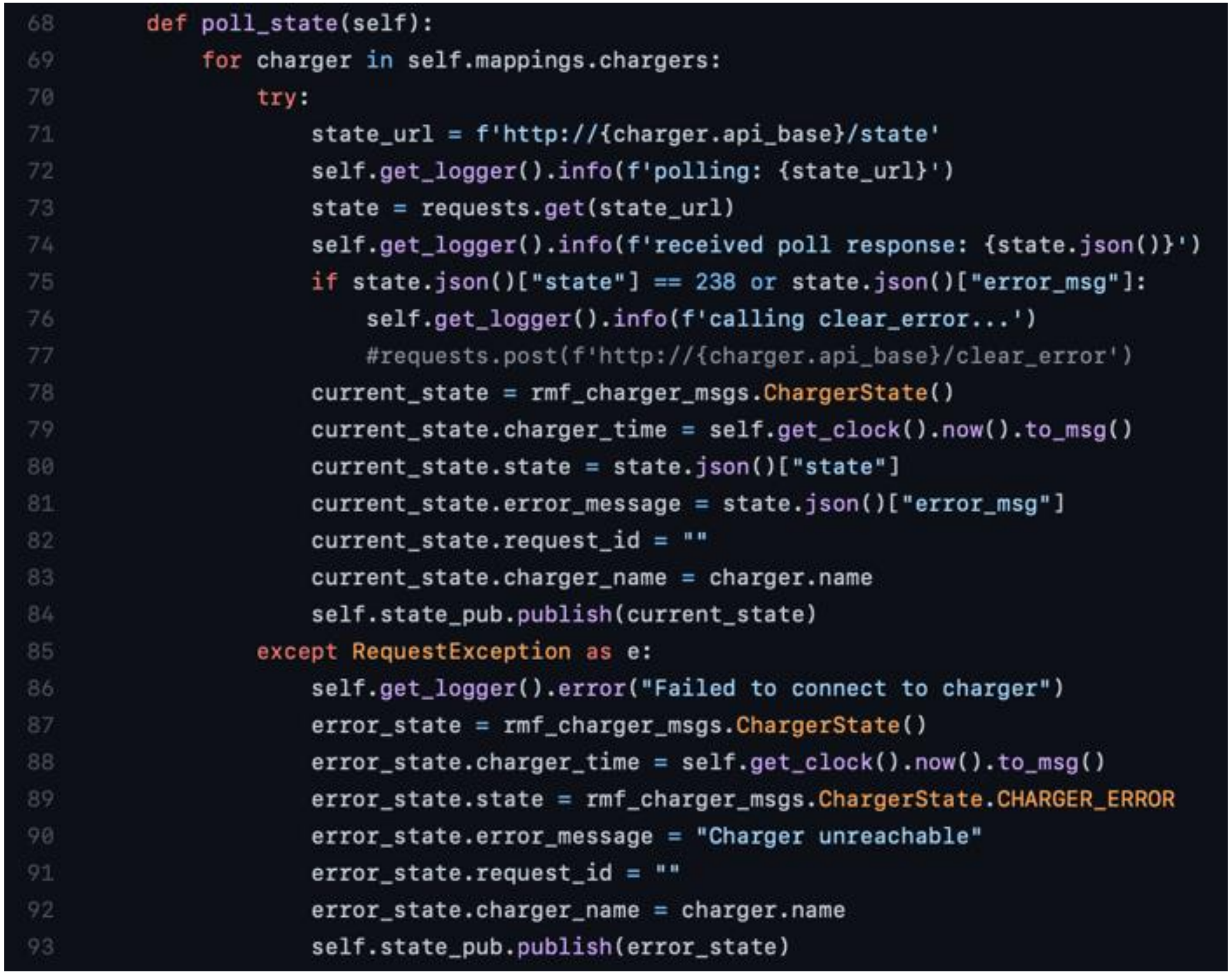
Message Exchange (RMF, Charger, Receiver)



Middleware
Robot
Receiver Charging Unit
Transmitter Powering Unit
Middleware

Proceed to Transmitter Station
Transmitter_Location()
Charging Status & Error States
Assign ID
Location_Status()
Robot reaches destination
Confirmed Location Status
Robot_RCU_ID()
Check Tx Status
Queue / New_Transmitter_Location()
Enable_Charging()
Request Charging Command
RF Communication Handshake
Robot awaits command
ID Check
All Check ok
Physical pairing success
Start Charging
Interrupt stop charging Request (option)
Charging Initiated
Tx_Charging_status()
Request_Stop_Charging()
Station Status Update (Occupied)
Charging Completed
Charging Status & Error States
Real-Time info exchange
Charging Status & Error States
Disable_Charging()
Stop Charging Command
Interrupt stop charging Command
Stop Charging command
Stop charging
Charging completed
Tx_Charging_status()
Stop ok
Charging stop
Complete charging command
Station Status Update (Available)
Charging_Stop()
Resume Tasks Status
Proceed_Tasks()
Robot moves away to continue tasks

Charger Adapter Example (Python)



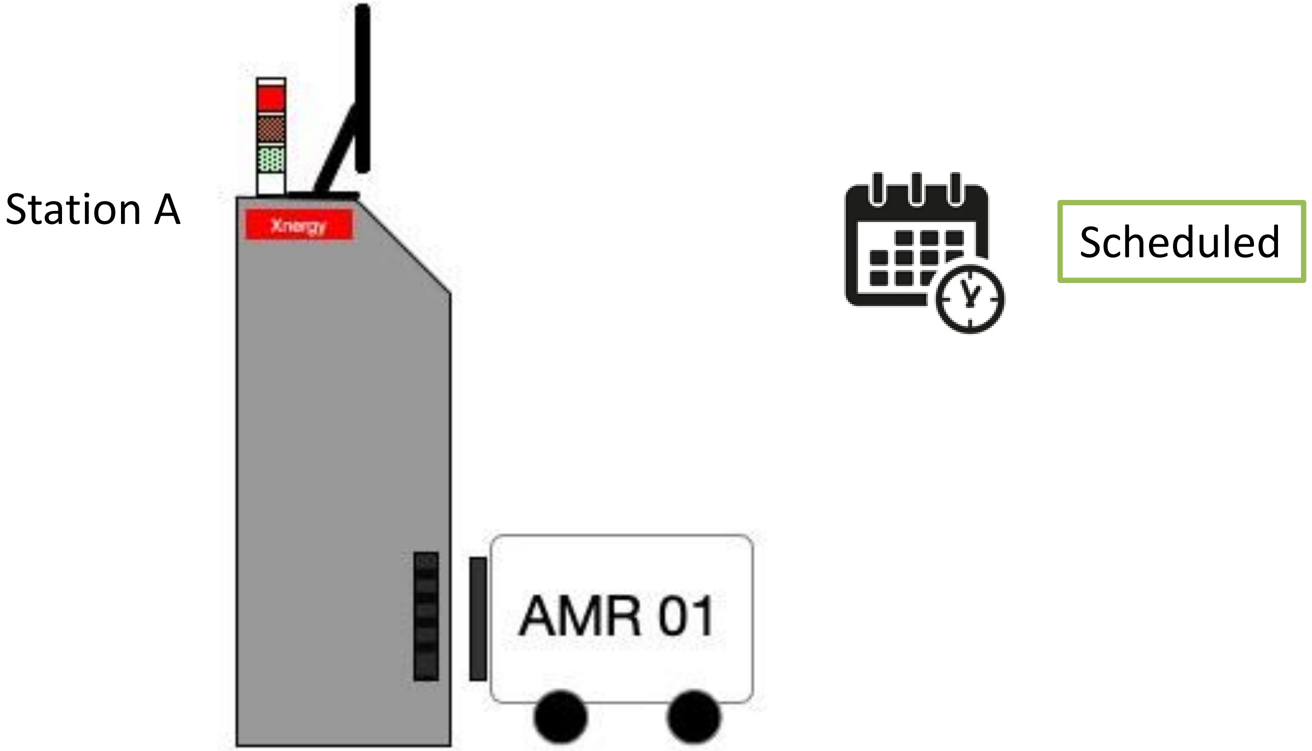
```python
    def poll_state(self):
        for charger in self.mappings.chargers:
            try:
                state_url = f'http://{charger.api_base}/state'
                self.get_logger().info(f'polling: {state_url}')
                state = requests.get(state_url)
                self.get_logger().info(f'received poll response: {state.json()}')
                if state.json()["state"] == 238 or state.json()["error_msg"]:
                    self.get_logger().info(f'calling clear_error...')
                    #requests.post(f'http://{charger.api_base}/clear_error')
                current_state = rmf_charger_msgs.ChargerState()
                current_state.charger_time = self.get_clock().now().to_msg()
                current_state.state = state.json()["state"]
                current_state.error_message = state.json()["error_msg"]
                current_state.request_id = ""
                current_state.charger_name = charger.name
                self.state_pub.publish(current_state)
            except RequestException as e:
                self.get_logger().error("Failed to connect to charger")
                error_state = rmf_charger_msgs.ChargerState()
                error_state.charger_time = self.get_clock().now().to_msg()
                error_state.state = rmf_charger_msgs.ChargerState.CHARGER_ERROR
                error_state.error_message = "Charger unreachable"
                error_state.request_id = ""
                error_state.charger_name = charger.name
                self.state_pub.publish(error_state)
```

5 Charging Scenarios with RMF

1. Scheduled Charging
2. Ad-hoc Self-requested Charging
3. Stacked up Charging Requests
4. Wrong Charging Station
5. Charging Failure

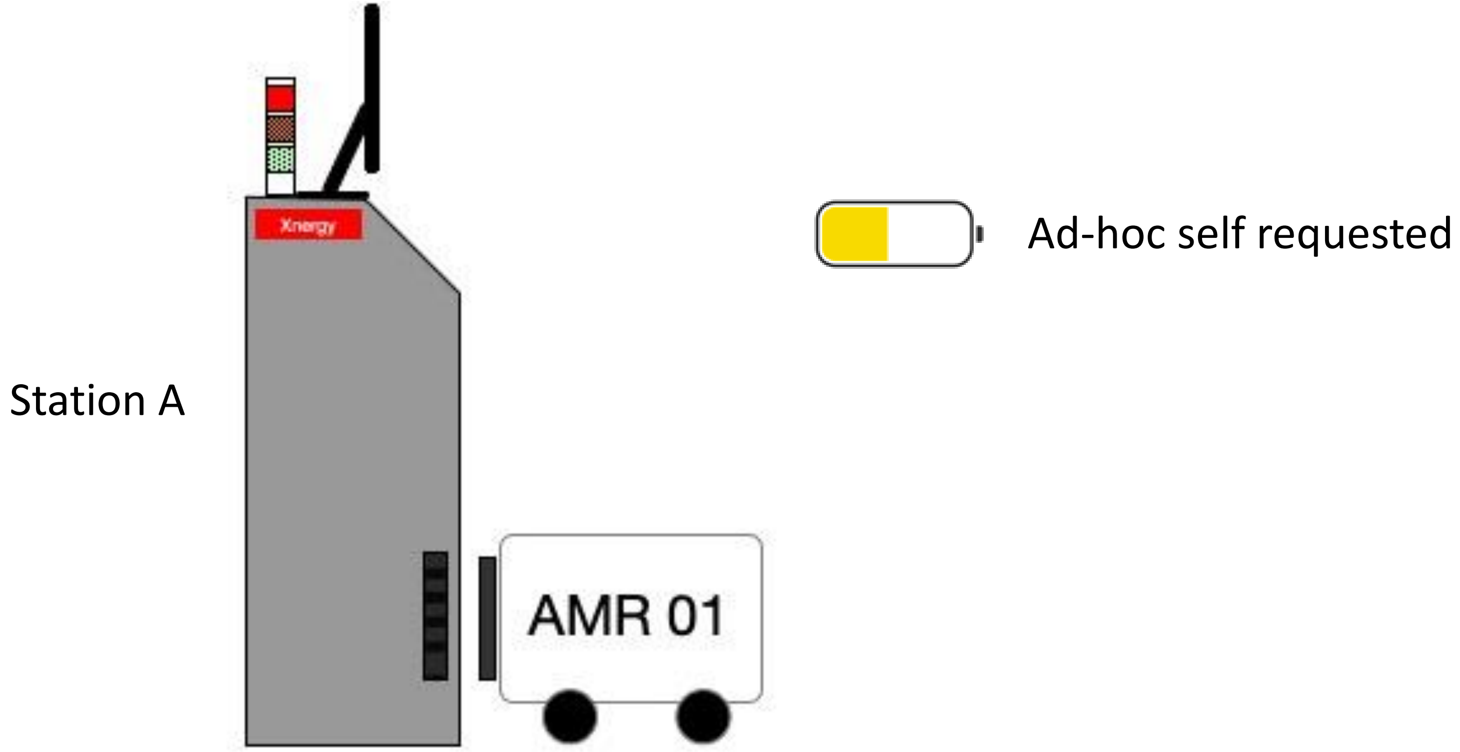
Scenario 1: Scheduled Charging

- **Charging task**: AMR01 is scheduled to charge at Station A.
- **Site Situation**: AMR01 arrived for charging. Charging Station A is available.
- **Action**: Handshake and initiate charging.



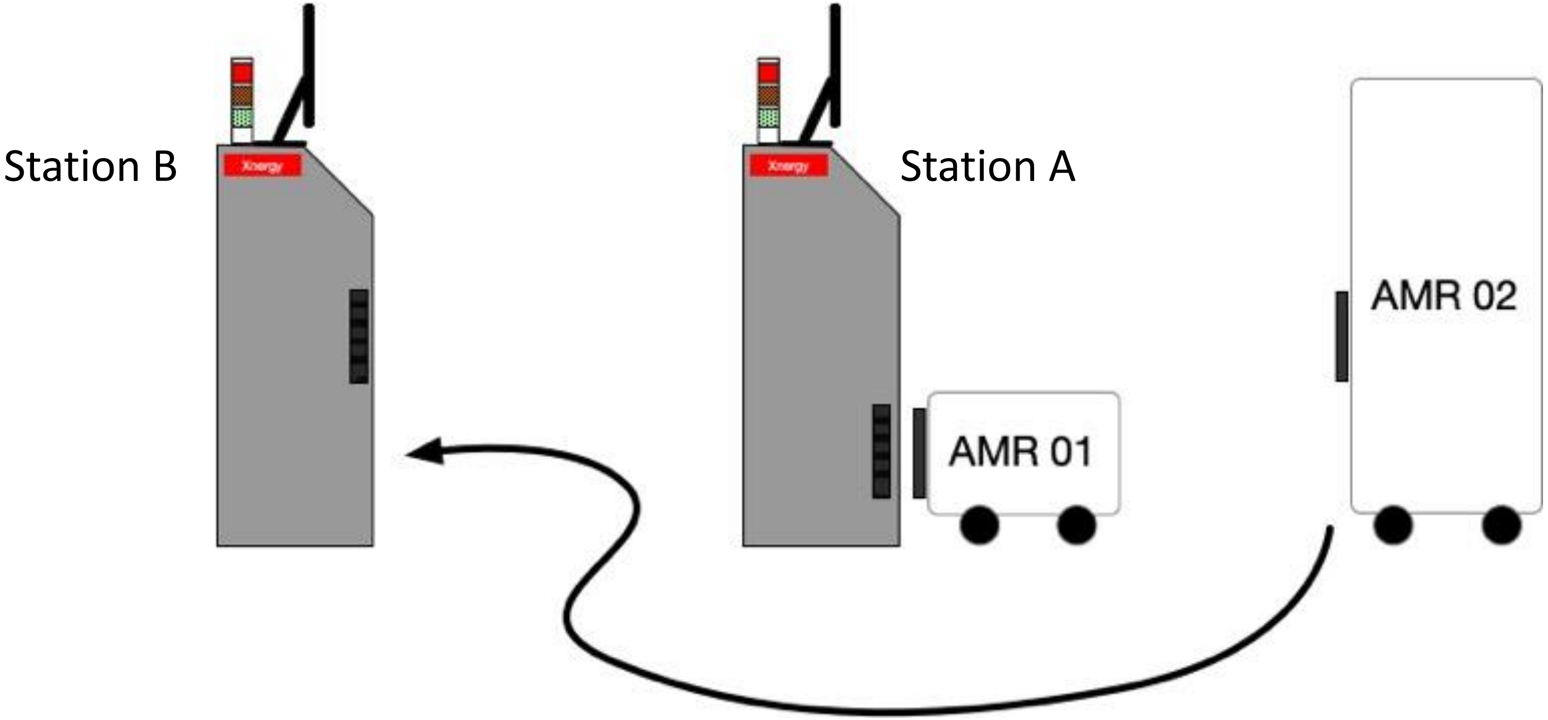
Scenario 2: Ad-hoc self-requested charging

- **Charging task**: AMR01 battery is low, requested for immediate charging and assigned to Station A.
- **Site Situation**: AMR01 arrived for charging. Charging Station A is available.
- **Action**: Handshake and initiate charging.



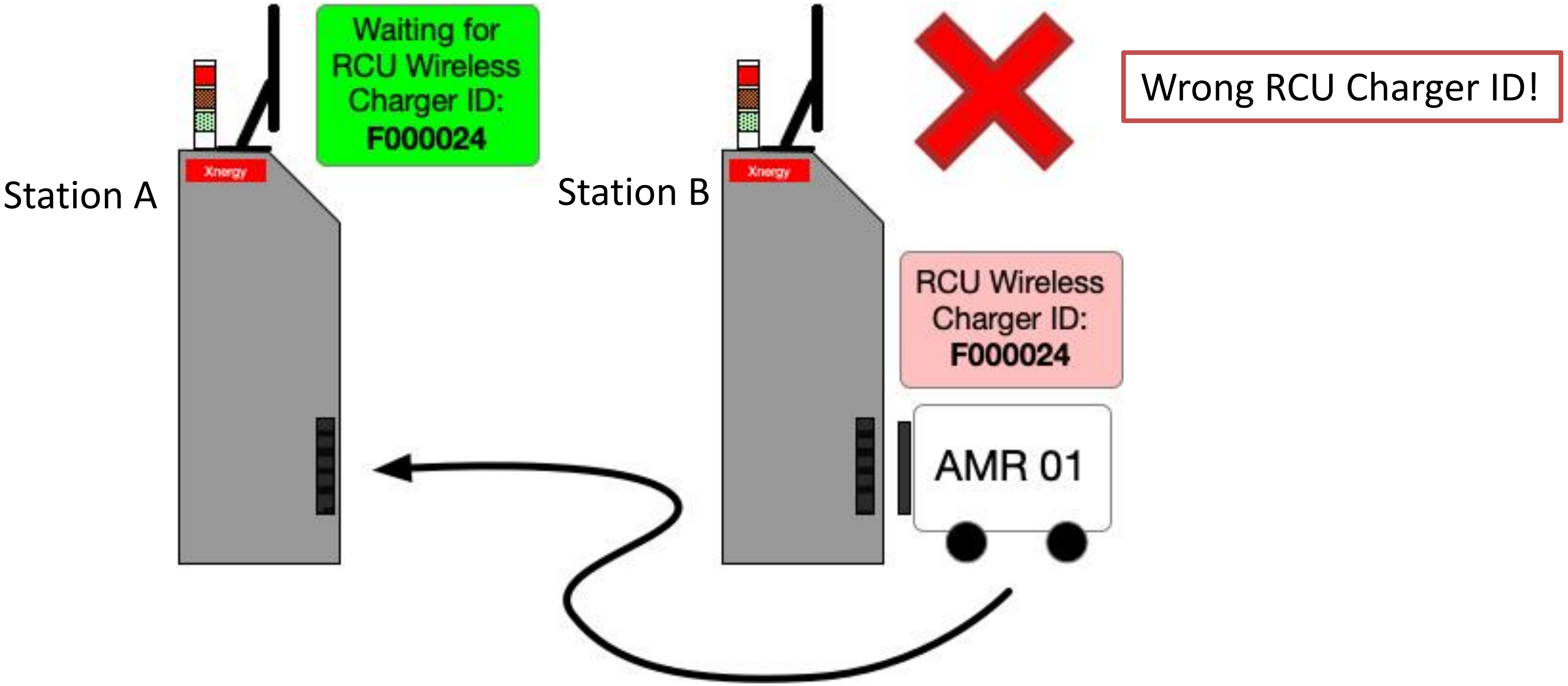
Scenario 3: Stacked up charging requests

- Charging task: AMR01 is scheduled to have finished charging at Station A. AMR02 is assigned to Station A for charging.
- Site Situation: AMR02 arrived for charging. Dock is still occupied by AMR01.
- Action: AMR02 reassigned to Station B (if available); OR AMR02 notified to queue up and wait for available station.



Scenario 4: Wrong charging station

- **Charging task**: AMR01 is scheduled to charge at Station A.
- **Site Situation**: AMR01 approached Station B for charging due to navigation error. AMR01 is trying to initiate charging with Station B.
- **Action**: The transmitter in Station B identifies AMR01. Station B notifies the wrong docking of AMR01. AMR01 moves to Station A for charging.



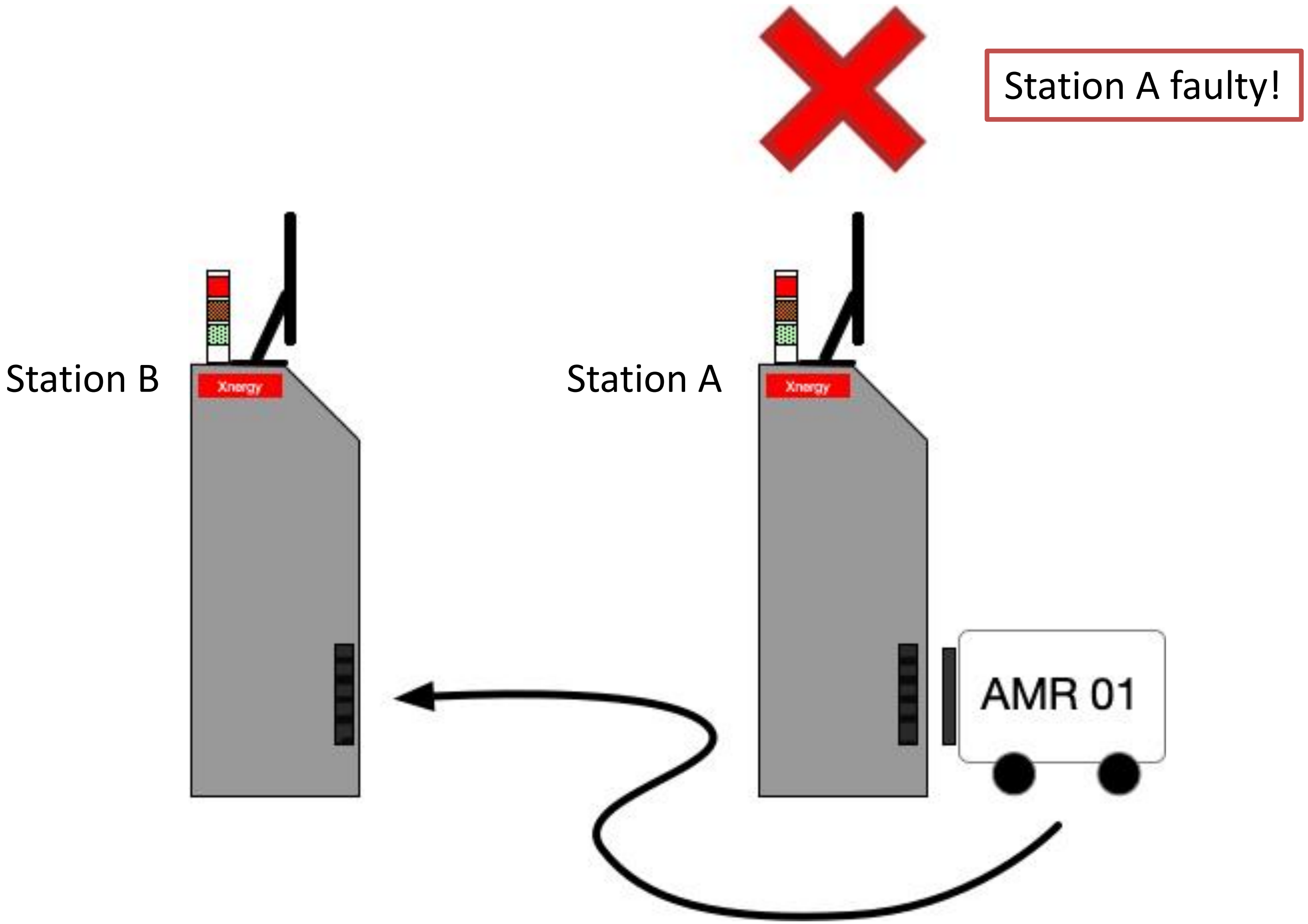
Scenario 5: Charging failure

- **Charging task**: AMR01 is scheduled to charge at Station A.

- **Site Situation**: AMR01 arrived for charging. Charging Station A is available. However, due to technical issues, the charging process cannot be initiated.

- **Action**: Station A inform RMF of the failure. Station A and AMR01 enter "protection mode" to prevent damages.
  - a. Self-check identified Station A faulty, AMR01 reassigned to Station B.
  - b. Self-check identified Station A good, AMR01 faulty. Technical support team activated for corrective maintenance.

Scenario 5: Charging failure


Station A faulty!

Video

1. 5 AMR charging at same charging station demo
2. RoMio SCM charging demo

Information:

[illegible]	Magni – CHART
[illegible]	F000 0024
[illegible]	24V (29V max)
[illegible]	4.3 A
[illegible]	Lead Acid
[illegible]	High