Current Concept of ROS <-> AWS connection

Option #1:
Simple Service/Client interaction

- Maintains a server client relationship between ROS and AWS
- Prevents potentially unnecessary overhead on the ROS side, with continuous publishing of states
- Will block AWS operations until the service request is honored/completed
- Does not require any special implementation

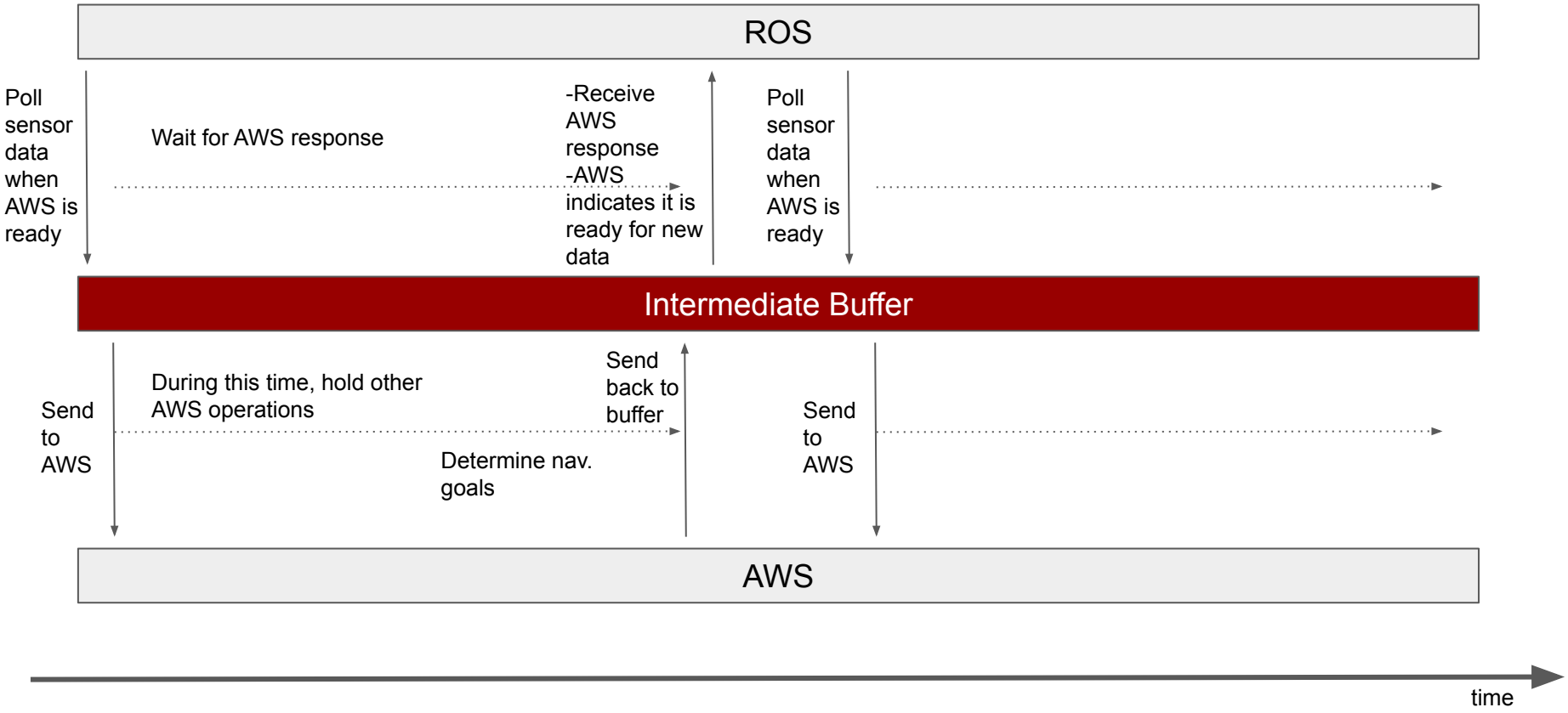
Reasons for using TCP/IP sockets:

- Using websockets is the easiest way to establish connection between the vehicle and AWS
- Protocol used by ROS 1
- Far more stable communication compared to UDP

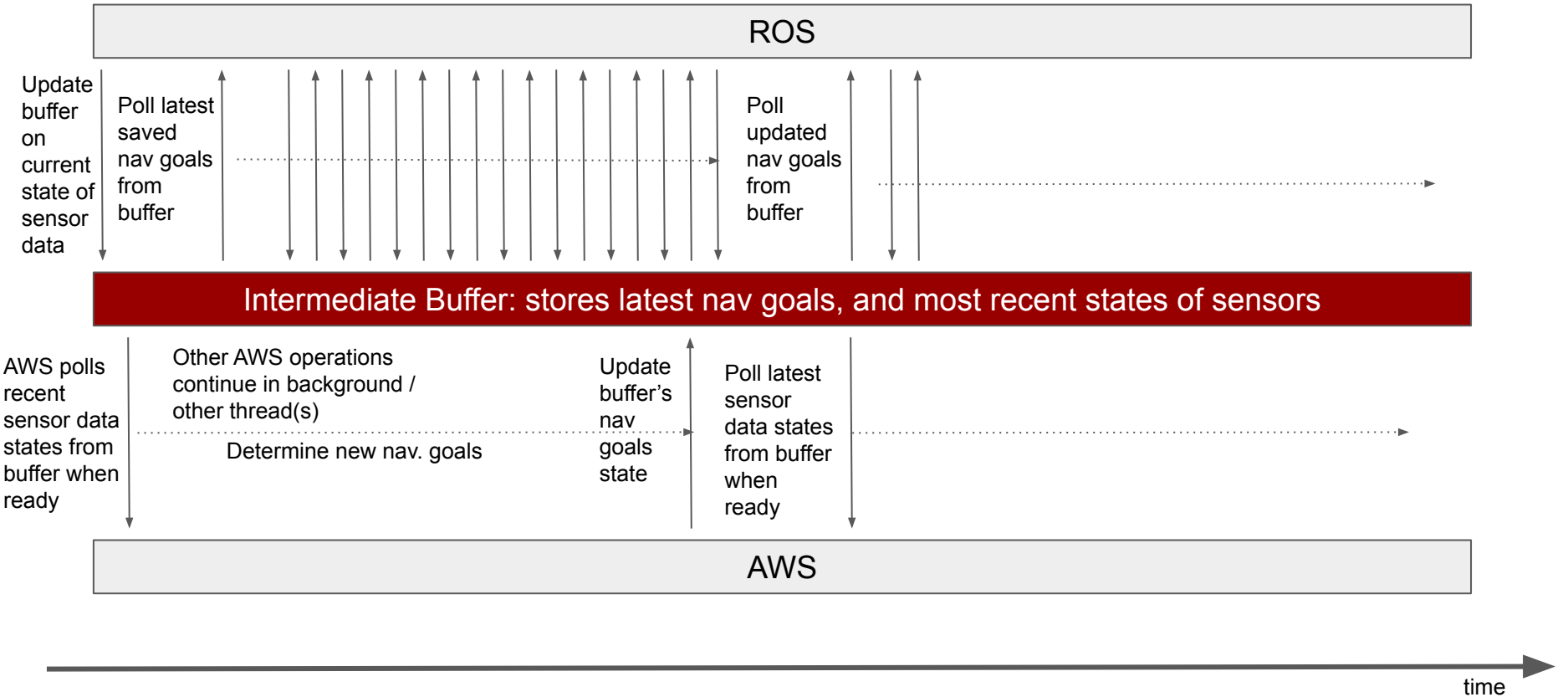
Option #2:
Continuous updating, latest state storage

- Continuously polls the vehicle for data
- Stores the latest states of the vehicle in a buffer
- Whenever the AWS requests data from ROS, it retrieves the latest vehicle data from the buffer
- Likewise, the bridge will keep publishing the latest information from AWS to the vehicle until AWS provides an update
- This requires the bridge to operate within either a while(true) loop, or to use multi-threading

Option #1: Simple Service/Client interaction



Option #2: Continuous updating, latest state storage



Methods of implementing TCP/IP connection

Option 1:

- Use the *rosbridge_suite* package for the ROS side of the TCP/IP connection, and utilize the *roslibpy* API for the AWS side. Sending input to ROS via the *roslibpy*-based API would require a properly formatted JSON string containing message data, and the output of the API when reading from ROS is a formatted JSON string containing message data. On the ROS side, there would be no direct interaction with JSON - simply publish to specific topics that the API will then subscribe to.
  - The main reason for doing so is that the code is easier to write and maintain
  - As both the *rosbridge_suite* and *roslibpy* packages are actively maintained and used by the community, there is greater reliability and scalability of the connection mechanism than writing one by ourselves

Option 2:

- Use the *rosbridge_suite* package for the ROS side of the TCP/IP connection, and utilize a self-written Python class for the AWS side. The method of interacting with the bridge will remain the same as in Option 1, but instead of using the *roslibpy*-based API, it is a completely self-written API.
  - In case using the *roslibpy* library in conjunction with the AWS code gives dependency issues, etc., another option is to directly create a Python script to run the connection to *rosbridge*'s TCP/IP socket instead. This comes at the cost of potentially worse stability.

Option 3:

- Create our own implementation of a TCP/IP socket in C++/Python on the ROS (vehicle) side, and use a self-written Python class for the AWS side. The Python class will take JSON inputs and return JSON outputs, and the ROS node will do the same. The JSON data processing on the ROS side will happen outside the bridge node.
  - This is in case we would like to use JSON messages as our standard of communication, and not the standard ROS message structure. However, this again comes at the cost of a much higher risk of instability.

I personally lean towards Option 1, as it has the highest stability, and will be the easiest for others to use and develop further on.