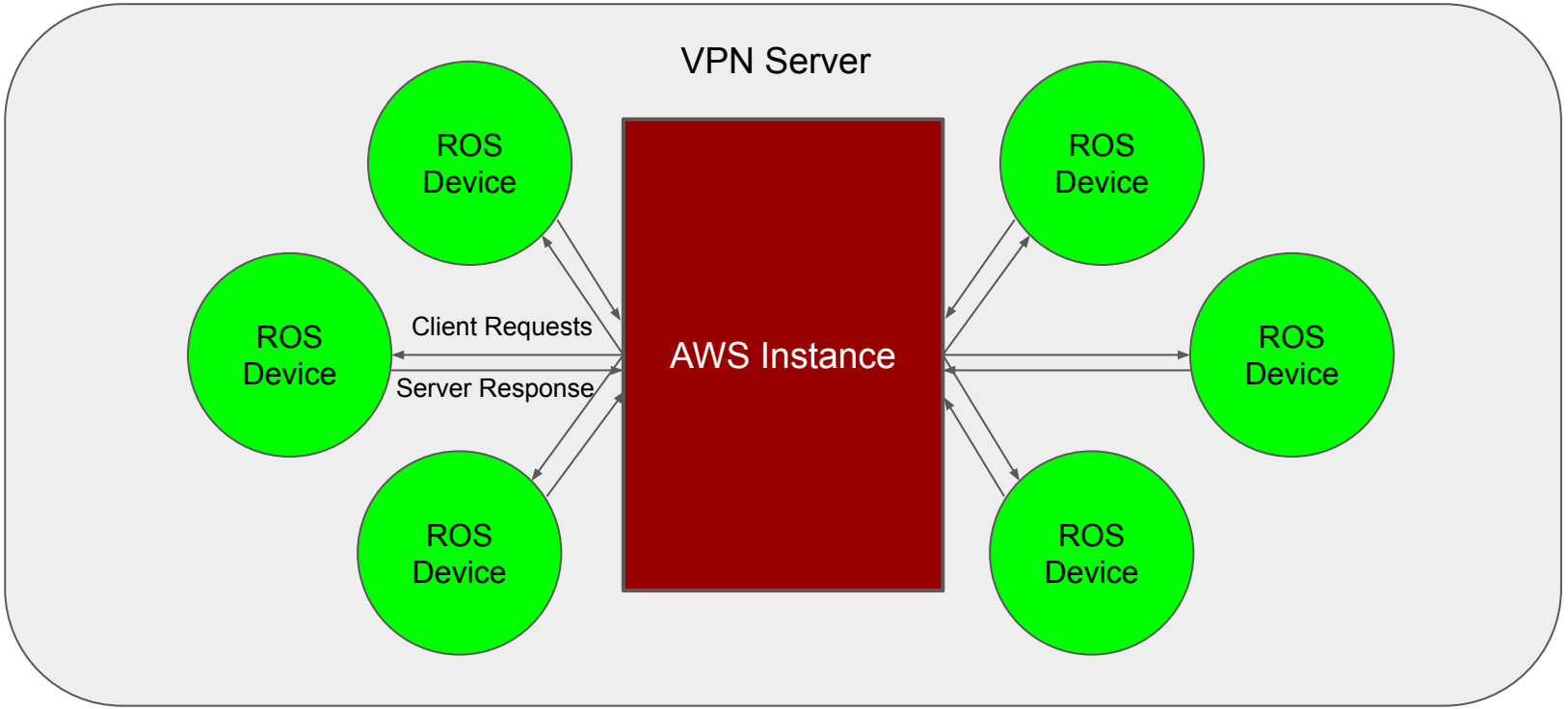
Implementation Options

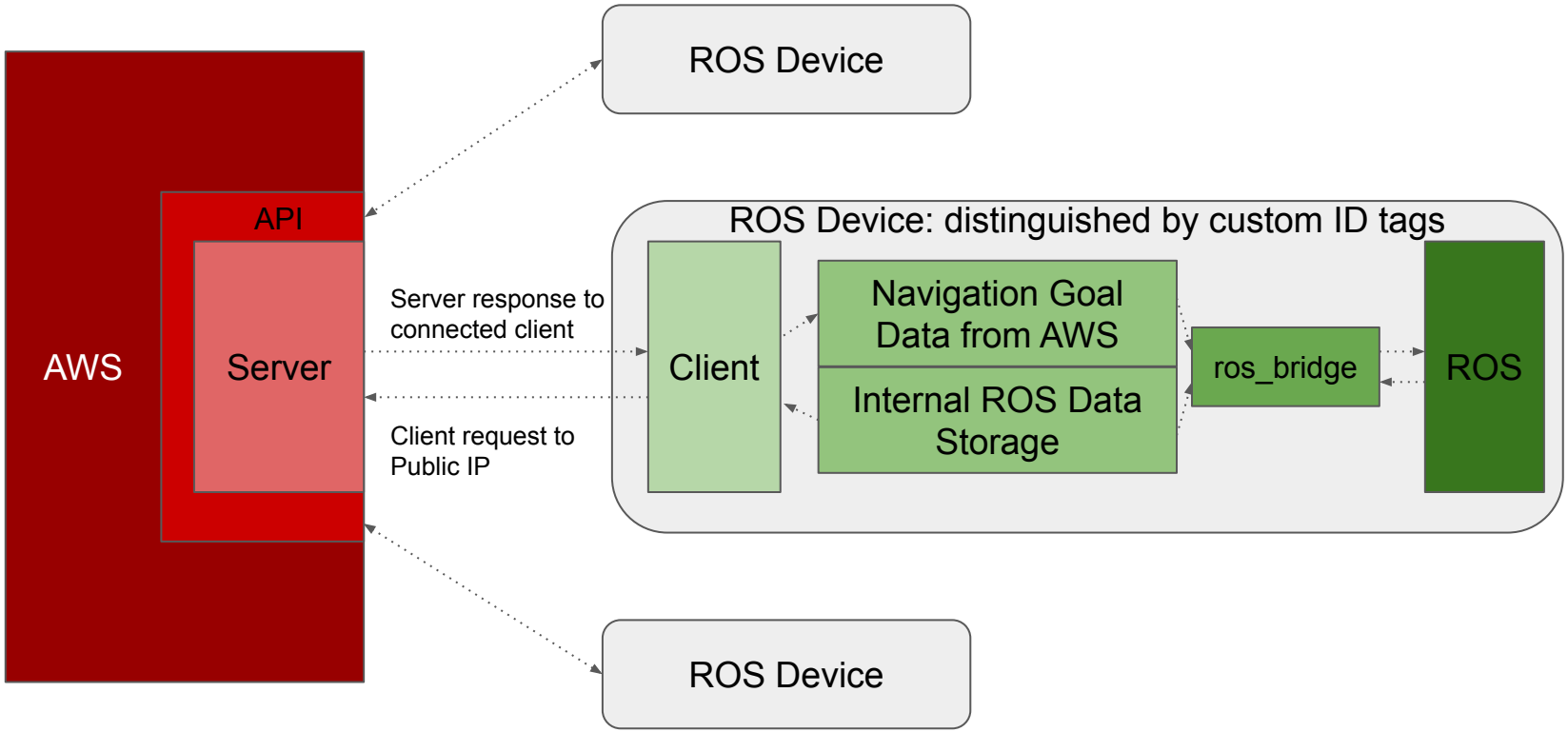
- The main issue with the previous implementation ideas was that only the AWS instance had a public IP, while the ROS devices did not. That required the AWS instance to act as a server, and not as a client - which was what the previous implementations desired.
- There are three potential workarounds:
  - Using the standard implementation within a VPN
  - Making AWS run the server **(chosen implementation)**
  - Using a client-server connection to bind AWS to the device's socket

VPN based

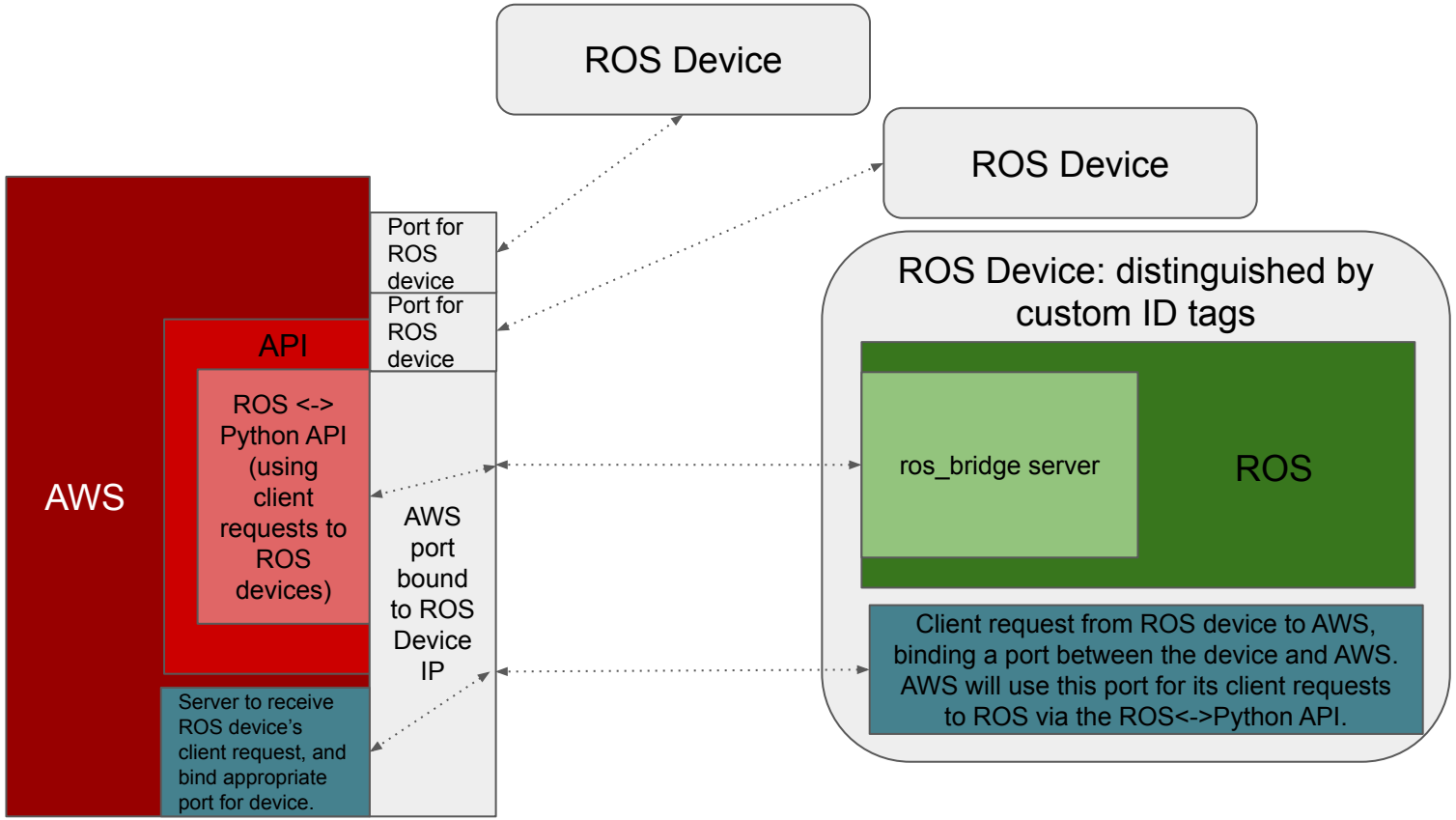


Access other ROS devices via their local IPs on the shared VPN server

Making AWS run the server: Basic implementation



Binding AWS to the device via client-server sockets



Chosen implementation

- The chosen implementation was to have AWS run a server. This was because:
  - The work required to get this implementation running was the least.
  - Time was limited.
- Reasons for not choosing other implementations:
  - VPN based: VPNs were faulty/unreliable in the past, making them less desired. Additionally, routing data through a VPN server might have added extra overhead and time delay than desired.
  - Having AWS bound to the device via client-server sockets: While nice in theory, it isn't known if this will work, and to avoid spending time on a potentially unusable idea, this approach was not chosen.

Final Implementation

- Structured like shown in diagram for “AWS running a server” implementation.
- Specifics:
  - AWS (Server):
    - The server runs in a separate thread on the AWS instance.
    - It uses a custom data-storage object to hold data going to the ROS device, and to store data from the ROS device. The data object has an API for other threads in AWS to retrieve the ROS data, and provide nav goals to ROS.
    - The messages to ROS must be in JSON format. The specific layout of the JSON message is specified in the code docs.
  - ROS (Client):
    - There are four layers on the device - ROS, an intermediary ROS to Python API, a websocket handler, and a top-level layer which binds these all together.
    - ROS uses the *rosbridge_suite* package to send information via websockets.
    - The intermediary ROS to Python API picks up and stores these messages in an internal data structure.
    - The websocket handler sends and receives data to and from AWS.
    - The top-level binding layer instantiates the API, and sends the stored ROS data to AWS via the websocket handler. It then retrieves the AWS response from the handler, parses it, and sends the response into ROS via the API.