Design and implementation of a Node.js Based Communication framework for an Unmanned Autonomous Ground Vehicle

Martin K. Mwila, *CSIR Defence, Peace, Safety and Security,email: MMwila@csir.co.za* and Perseverance Mbewe, *CSIR, Meraka Institute, email: PMbewe@csir.co.za*

***Abstract*—This paper describes the methodology used to develop a communication platform that enhances interoperability between different types of components irrespective of the manufacturers and of the software platform. This framework is intended to be used on a distributed system where software and hardware modules are designed to control an autonomous Unmanned Ground Vehicle (UGV). A messaging architecture based on the Joint Architecture for Unmanned Systems (JAUS) was developed in Node.js to ensure platform independence. It was deployed on hardware platforms such as the Raspberry Pi and the BeagleBone Black in order to access various sensors on the platform and control hardware like stepper motor. This messaging architecture can also be implemented on conventional laptops running Windows operating system or Linux that run algorithms for localisation, terrain mapping and path planning. Initially regarded as a very limited language, JavaScript's true nature and power have only recently been appreciated in depth, A major move is now underway to apply this language in new and fascinating contexts. The ultimate goal of the framework was to structure communication and inter-operation of UGV components and a sensor suite within a network. The framework was implemented on the G-Bat, a UGV platform developed at CSIR DPSS Landward Sciences to test and simulate the communication part of an autonomous navigation system. The test was a successful step that paves the way for a more robust implementation of the framework in the future work.**

***Keywords*—Unmanned Ground Vehicle, Communication Framework, Socket.IO, Node.js, Asynchronous Waterfall Model, Joint Architecture for Unmanned Systems.**

I. Introduction

Research and development of an unmanned system for various uses is ongoing around the world. Autonomous navigation is still a challenge, because of the computation complexity involved in processing data form different sensors. Designing innovative communication system that scale to facilitate potential new usage patterns can pose significant challenges. This situation is particularly prevalent in the case where these services are to be delivered over existing protocols and inter-operate with legacy services. In addition, this work explores the design choices for such a service. The complexity of the design implementation and testing of autonomous vehicle systems requires the integration of software components that manage sensing, control, actuation and logging.

A single executable file that runs the entire system is impractical, due to the variety of tasks that must be addressed, and the difficulty of debugging a multi-threaded design. On the other hand, a component-based design permits a functional decomposition of system into smaller tasks, and allows those tasks to communicate via a message passing framework which would even abstract whether a set of processes are running on the same machine or over the network.

Many different technologies address this particular issue of middleware. One architecture put forth by the United States (US) Department of Defence is the Joint Architecture for Unmanned Systems (JAUS). This architecture is now a standard of the Society of Automotive Engineers(SAE). JAUS is a set of standards that govern the construction of and interactions between computers and components in these systems.

This work aims at implementing a distributed approach developed in Node.js that uses a message passing framework that emulates JAUS. Node.js uses the Google V8 JavaScript engine to execute code and a large percentage of the basic modules are written in JavaScript. It provides an event-driven architecture and a non-blocking I/O API that optimises an application's throughput and scalability.

These technologies are commonly used for real-time applications. JavaScript might just be the most widely deployed programming environment in history (every web browser in existence). JavaScript, a language originally created to be used inside the web browser, will run outside the browser but inside each UGV hardware component.

Running JavaScript outside the browser is made possible by Node.js. Node.js is a single threaded application that is fast, event driven and structured to use non-blocking I/O. This amazing feature of Node.Js will to cater for the real-time operation requirements. It is open source and cross platform. It is also lightweight and can easily integrate to the web and the Internet of things (IoT). Many developers already familiar with JavaScript can now develop components for autonomous navigation system that would work, irrespective of the hardware platform on which this communication software is running.

II. Background

A. Related work

Technologies aimed at developing a common communication framework based on the same approach can be found in literature. The OpenJAUS LLC, developed by Dr. Danny Kent is aimed at providing a complete and robust middleware solution for Unmanned Systems developers [1], [2], [3], [4] . It is developed in C++ and only runs on limited platform architectures and is available only for Windows and Linux (Ubuntu) systems. It is difficult to implement and requires an advanced knowledge of C++. In addition, the licensing mechanism of this SDK is complex and not cheap.

The other variants of the JAUS implementation found in the literature are the JAUS++ and the JAUS toolset. JAUS++ is an Open Source (BSD) C++ implementation of JAUS (i.e. JAUS SDK) developed by the ACTIVE Laboratory for use in real and simulated unmanned vehicles. This and other languages such as the Matlab iJAUS and the LabVIEW implementation of JAUS are designed to support the modeling, analysis, specification , simulation and testing of the protocol for distributed systems [5].

On the other hand, the Robot Operating System (ROS) which is a collection of software frameworks for robot software development providing an operating system-like functionality on heterogeneous computer cluster. ROS provides services such as hardware abstraction, low level device control and message passing between processes. It is released under the BSD license and is free for both commercial and research use. However, the main client libraries (C++, Python, and LISP) are geared towards a Unix-like systems and primarily on Ubuntu Linux. Other operating systems are listed as experimental and only supported by the community [6].

B. Motivation

While several groups have developed JAUS libraries for C++, LabVIEW, Matlab, and Java, none has yet been proposed in the literature that uses JavaScript. Node.js, a programming language that is asynchronous in nature, allows the communication software developed in this framework to be :

- single threaded;
- fast;
- non-blocking I/O;
- cross platform;
- open source;
- lightweight.

On the other hand, JavaScript programming language is very well known by a substantial number of developers and have a wide contributing community that share ready to use and extensively tested modules that address a wide range of applications. The development tools are simple and free (a text editor, Node.js and a browser). The data exchange is in a JSON format, that is universally understood by any programming language.

In addition, the components need not to be written in Node.js and can be developed in any other programming language as Node.js is very good at interfacing to other languages and can spawn other processes if compiled into binary code. These attributes give an edge to the communication framework this technology demonstrator proposes compared to the existing ones.

III. System Overview

The UGV is implemented as a subsystem of a complete Autonomous System designed at CSIR DPSS Landward Sciences and is depicted in Figure 1.

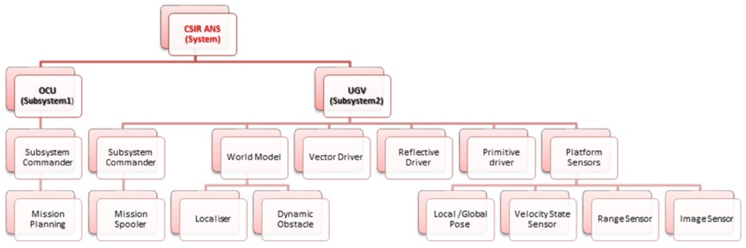


Fig. 1. Autonomous Navigation System Overview

IV. Objectives

The framework developed in this work is the entry point of all communication within a UGV as well as the whole autonomous navigation system. It can be extended to any other autonomous or robotic systems. The objectives pursued in this work were to:

- develop and implement the hardware and software components responsible for sending messages between the Operator Control Unit (OCU) and Unmanned Ground Vehicle (UGV) subsystems as well as the inter-node communications;
- coordinate the communication between modules and provide acknowledgement responses;
- develop and implement a Graphical User Interface (GUI) on a tablet PC that will be able to send manual commands (remote control) to the UGV and display the mission status of the UGV;
- build the wireless communication network from the OCU to the UGV;

V. Methodology

The communication framework is implemented as a messaging server using Socket.IO in Node.js . Some implementation of the Socket.IO client in Python is also available. The messages are all defined and assigned a message ID. The format is standardised according to the JAUS Transport Specification, and are routed to the required node by the communicator component (server). Each component responds to the message with a corresponding message and the data passing is in JSON format.

An acknowledgment to each message is provided to make the communication robust.

The communicator keeps the tree of connected node and broadcast the list to every node connected to the network to advertise the services available.

The communicator also allows control of the G-Bat via a web based interface. this capability is achieved by connecting an OCU (tablet) to the network via the Wi-Fi access point or remotely from the control office via a wireless modem. The communication framework was designed with following characteristics in mind:

A. *Reliability*

The Socket.IO based communication was chosen for the reliability of its connections. Connections are established even in presence of proxies, personal firewall and anti-virus software.

B. *Connection and disconnection detection*

The library implements a mechanism at the core level to allow the server and the client to know when the other one is connected or not responding any more. This feature has been used to implement a creation and update of a system tree of connected nodes.

C. *Auto-reconnection support*

The Socket.IO library supports auto-reconnection in the event of unscheduled disconnection unless otherwise instructed in the code.

D. *Room support*

This useful feature sends notifications to a group of users, or to a given user connected on several devices for example.

E. *Message synchronisation*

The waterfall model design was chosen to synchronise all messages in the system. It provided a linear sequential flow of passing of messages from the root subsystem (OCU) to its children. Figure 2 depict the waterfall model

The passing of messages to other subsystems starts when the destination of the vehicle is approved by the OCU.

The next subsystem begins its process only if the previous subsystem task is complete. As such, all the subsystems do not overlap. When a subsystem is blocked, a callback is invoked to restart the process from the beginning [7].

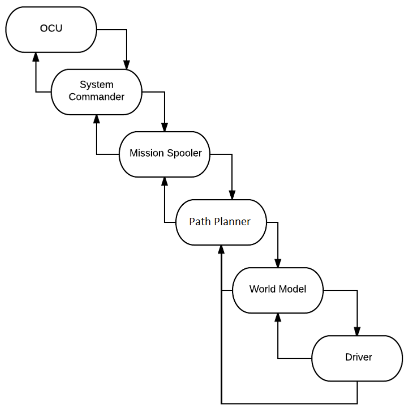


Fig. 2. Message synchronisation using waterfall model

VI. Implementation

A. *Hardware implementation*

The complexity is high as the technology has to integrate Web servers, http technology, TCP/IP as well as I/O access and control on Raspberry Pi as well as spawning child processes written a programming language other than Node.js. The change of paradigm in the programming philosophy from embedded C++ to JavaScript was challenging. However, the the Node.js user community was of a great assistance.

Figure 3 depict the hierarchical configuration of the UGV hardware.

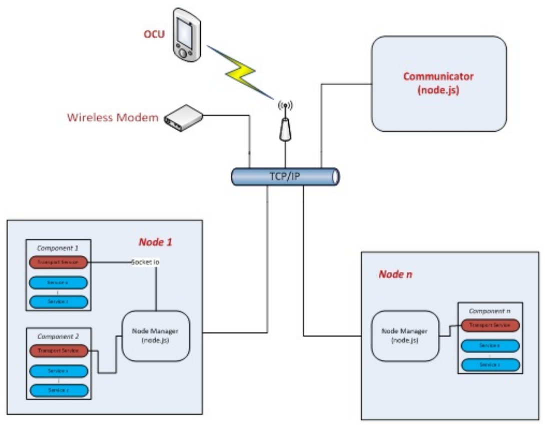


Fig. 3. Hardware Implementation Design

The Network and power supplies were installed on the G-Bat UGV in the Autonomous Lab at Landwards Sciences. The communicator software was installed on the ComSIS LTE MiFi 4G router, a communication device obtained from Tshwane University of Technology. This module houses a Linux PC running Node.js . The Communicator software will run there as a local service. Figure 4 depict the Communicator node and its services running on the ComSIS LTE MiFi router.

Fig. 4. Communicator node

The router is equipped with a DHCP server that also has a Wi-Fi access point to which the tablet used as an OCU can connect. The other component software were run on Raspberry Pi to simulated the other components of the UGV connecting to the network in order to test the communication software. These Single Board Computers (SBC) were connected to the G-Bat network using the Ethernet switch.

B. *Software implementation in Node.js*

The Communicator component maintains all data links to other subsystems within the system. The Communicator also allows for a single point of entry to any subsystem. It is implemented in Node.js as a Socket-Io server with socket clients being the other subsystem components. It also include as a Web server using with the client running in a browser at the OCU tablet.

The other nodes are implemented in Node.js as Socket-Io clients connecting to a server running on the Communicator node.

Room features and socket IDs are used for individual communications and broadcast. Each node send an acknowledgement message back to the sender node.

VII. Test Setup

The test was conducted in the laboratory on the G-Bat test platform as depicted in Figure 5.

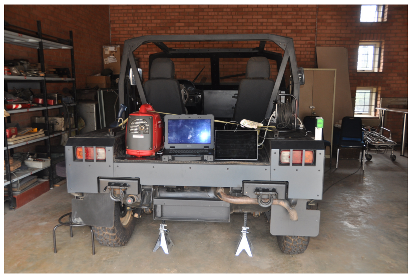


Fig. 5. G-Bat Test platform

The complete communication loop was tested with test code that initiated the process from the planner module to the primitive driver node via the Vector knowledge store node. Figure 6 depict the hardware setup for the test executed on the framework in the Autonomous Navigation Laboratory at CSIR DPSS Landward Sciences.

VIII. Test results

Below are screen shots of the test results. From a main computer we have been able to make secure shell (SSH) connections to nodes running on different single boards computers such as the Raspberry Pi and the BeagleBone

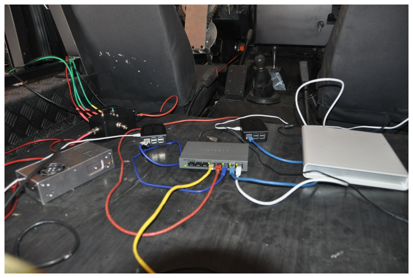
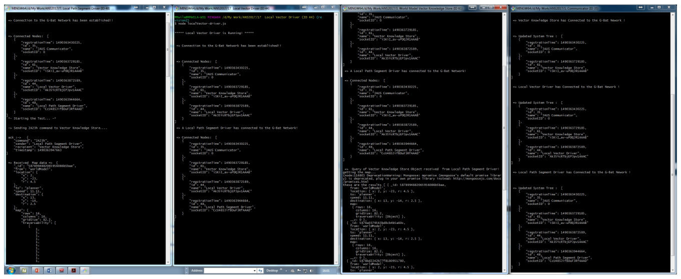


Fig. 6. Test setup (Hardware)



```
MINGW64:/d/My Work/ANS2017/7. Local Path Segment Driver (ID 48)

=> Connection to the G-Bat Network has been established!!


=> Connected Nodes:  [
    {
        "regstrationTime": 1490363430225,
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    {
        "regstrationTime": 1490363872589,
        "id": 44,
        "name": "Local Vector Driver",
        "socketID": "Ak35YcRTbjEPJpv1AAAC"
    },
    {
        "regstrationTime": 1490363944664,
        "id": 48,
        "name": "Local Path Segment Driver",
        "socketID": "CcU48SJifBDuF3RfAAAD"
    }
]
"- Starting the Test... -"

-> Sending 2A23h command to Vector Knowledge Store....

ack :->   {
  "command": "2A23h",
  "sender": "Local Path Segment Driver",
  "recipient": "Vector Knowledge Store",
  "timestamp": 1490363947663
}

=> Received  Map data =>  {
  "_id": "5878994882065956086b5bae",
  "from": "worldModel",
  "location": {
      "x": 2,
      "y": -23,
      "r": 4.5
  },
  "to": "planner",
  "speed": 11.11,
  "destination": {
      "x": 13,
      "y": -14,
      "r": 2.5
  },
  "map": {
      "rows": 16,
      "columns": 16,
      "gridSize": 82.2,
      "traversability": [
          [
              1,
              1,
              1,
              1,
              1,
              1,
              1,
              1,
              1,
              1,
              1,
```

```
MINGW64:/d/My Work/ANS2017/17  Local Vector Driver (ID 44)

MMwila@MMWILA-WS1 MINGW64 /d/My Work/ANS2017/17  Local Vector Driver (ID 44) (re
vision2)
$ node localVector-driver.js

***** Local Vector Driver is Running! *****


 => Connection to the G-Bat Network has been established!!


 => Connected Nodes:  [
    {
        "regstrationTime": 1490363430225,
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    {
        "regstrationTime": 1490363872589,
        "id": 44,
        "name": "Local Vector Driver",
        "socketID": "Ak35YcRTbjEPJpv1AAAC"
    }
]

 => A Local Path Segment Driver has connected to the G-Bat Network!

 => Connected Nodes:  [
    {
        "regstrationTime": 1490363430225,
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    {
        "regstrationTime": 1490363872589,
        "id": 44,
        "name": "Local Vector Driver",
        "socketID": "Ak35YcRTbjEPJpv1AAAC"
    },
    {
        "regstrationTime": 1490363944664,
        "id": 48,
        "name": "Local Path Segment Driver",
        "socketID": "CcU48SJifBDuF3RfAAAD"
    }
]
```

```
MINGW64:/d/My Work/ANS2017/3. World Model Vector Knowledge Store (ID 61)
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    {
        "regstrationTime": 1490363872589,
        "id": 44,
        "name": "Local Vector Driver",
        "socketID": "Ak35YcRTbjEPJpv1AAAC"
    }
]

 => A Local Path Segment Driver has connected to the G-Bat Network!

 => Connected Nodes:  [
    {
        "regstrationTime": 1490363430225,
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    {
        "regstrationTime": 1490363872589,
        "id": 44,
        "name": "Local Vector Driver",
        "socketID": "Ak35YcRTbjEPJpv1AAAC"
    },
    {
        "regstrationTime": 1490363944664,
        "id": 48,
        "name": "Local Path Segment Driver",
        "socketID": "CcU48SJifBDuF3RfAAAD"
    }
]


 =>  Query of Vector Knowledge Store Object received  from Local Path Segment Driver!
getting the map...
(node:11680) DeprecationWarning: Mongoose: mpromise (mongoose's default promise librar
y) is deprecated, plug in your own promise library instead: http://mongoosejs.com/docs
/promises.html
these are the results [ { _id: 5878994882065956086b5bae,
    from: 'worldModel',
    location: { x: 2, y: -23, r: 4.5 },
    to: 'planner',
    speed: 11.11,
    destination: { x: 13, y: -14, r: 2.5 },
    map:
     { rows: 16,
       columns: 16,
       gridSize: 82.2,
       traversability: [Object] },
    __v: 0 },
  { _id: 5878a0379563bd8cb4b5a60c,
    from: 'worldModel',
    location: { x: 2, y: -23, r: 4.5 },
    to: 'planner',
    speed: 11.11,
    destination: { x: 13, y: -14, r: 2.5 },
    map:
     { rows: 16,
       columns: 16,
       gridSize: 82.2,
       traversability: [Object] },
    __v: 0 },
  { _id: 5878bd2242b77f8180951780,
    from: 'worldModel',
    location: { x: 2, y: -23, r: 4.5 },
    to: 'planner',
```

```
MINGW64:/d/My Work/ANS2017/1. Communicator (ID 35)

=> Vector Knowledge Store has Connected to the G-Bat Nework !



=> Updated System Tree :  [
    {
        "regstrationTime": 1490363430225,
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    }
]

=> Local Vector Driver has Connected to the G-Bat Nework !



=> Updated System Tree :  [
    {
        "regstrationTime": 1490363430225,
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    {
        "regstrationTime": 1490363872589,
        "id": 44,
        "name": "Local Vector Driver",
        "socketID": "Ak35YcRTbjEPJpv1AAAC"
    }
]

=> Local Path Segment Driver has Connected to the G-Bat Nework !



=> Updated System Tree :  [
    {
        "regstrationTime": 1490363430225,
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    {
        "regstrationTime": 1490363872589,
        "id": 44,
        "name": "Local Vector Driver",
        "socketID": "Ak35YcRTbjEPJpv1AAAC"
    },
    {
        "regstrationTime": 1490363944664,
        "id": 48,
        "name": "Local Path Segment Driver",
        "socketID": "CcU48SJifBDuF3RfAAAD"
    }
]
```

Fig. 7. Overview of the system's communications

Black and have been able to monitor the communication messages at each node level.

The following figures show samples of messages obtained during the test. In these figures, the node IDs, message IDs and timestamps can be seen. The tests also shows that the framework maintains a list of connected nodes and broadcasts it to all nodes in real time as a node connects or disconnects. The acknowledgment messages can also be seen. All these messages are logged on the console and also in a file that can be used for debugging if any communication problem arise.

Figure 8 depicts the messages on a test segment driver node demonstrating a request of a transferability map.

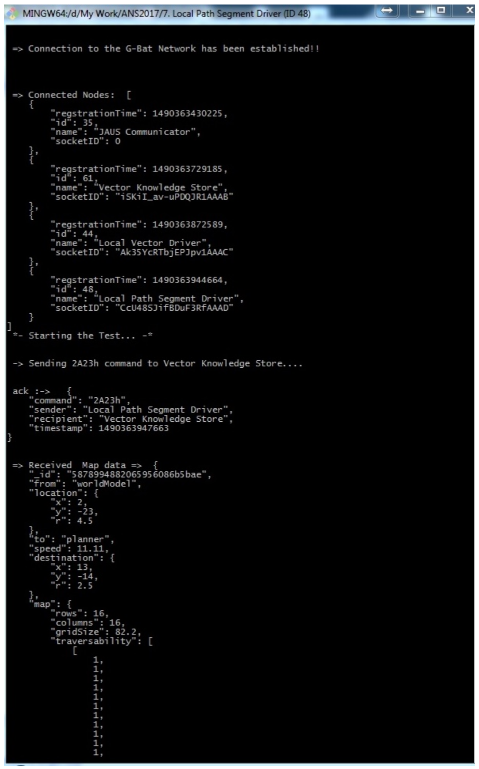
Figure 9 illustrates how the Communicator node manages the communications in the whole subsystem. A log file is saved on this node.

Finally, a tester module was written to simulate the whole system communication. A sample of the message generated is seen in Figure 10.

The communication speed has been measured from the timestamps that were implemented in software and has been found to be excellent. For a map of 16x16 grids, the communication delay introduced by this type of data size has been found to be negligible.

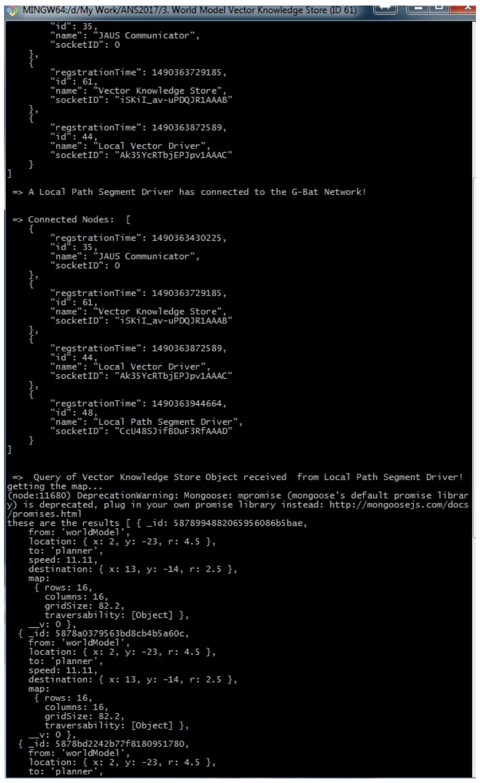
IX. Performance evaluation of Socket.IO in Node.js

There has been a lot of debate about the relative performance of single-threaded like Node.js versus multiple-threaded appli-


MINGW64:/d/My Work/ANS2017/7. Local Path Segment Driver (ID 48)

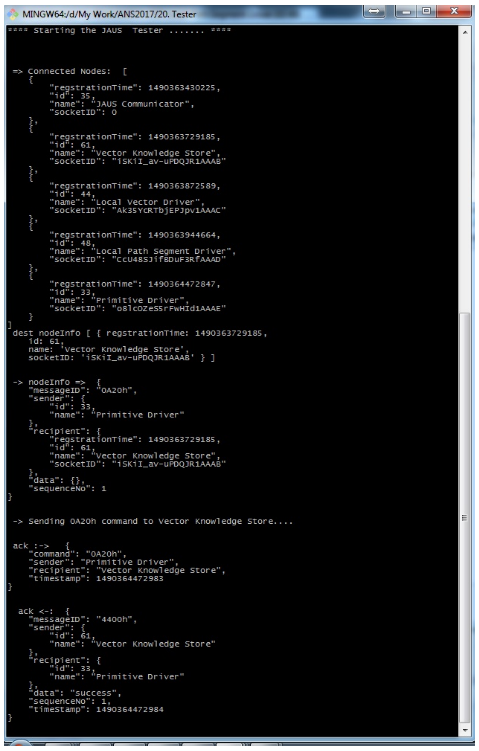
```
 => Connection to the G-Bat Network has been established!!



 => Connected Nodes:  [
    {
        "regstrationTime": 1490363430225,
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    {
        "regstrationTime": 1490363872589,
        "id": 44,
        "name": "Local Vector Driver",
        "socketID": "Ak35YcRTbjEPJpv1AAAC"
    },
    {
        "regstrationTime": 1490363944664,
        "id": 48,
        "name": "Local Path Segment Driver",
        "socketID": "CcU48SJifBDuF3RfAAAD"
    }
]
 *- Starting the Test... -*


 -> Sending 2A23h command to Vector Knowledge Store....


 ack :->   {
    "command": "2A23h",
    "sender": "Local Path Segment Driver",
    "recipient": "Vector Knowledge Store",
    "timestamp": 1490363947663
}


 => Received  Map data =>  {
    "_id": "5878994882065956086b5bae",
    "from": "worldModel",
    "location": {
        "x": 2,
        "y": -23,
        "r": 4.5
    },
    "to": "planner",
    "speed": 11.11,
    "destination": {
        "x": 13,
        "y": -14,
        "r": 2.5
    },
    "map": {
        "rows": 16,
        "columns": 16,
        "gridSize": 82.2,
        "traversability": [
            [
                1,
                1,
                1,
                1,
                1,
                1,
                1,
                1,
                1,
                1,
                1,
```

Fig. 8. Segment driver module communications


MINGW64:/d/My Work/ANS2017/3. World Model Vector Knowledge Store (ID 61)

```
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    {
        "regstrationTime": 1490363872589,
        "id": 44,
        "name": "Local Vector Driver",
        "socketID": "Ak35YcRTbjEPJpv1AAAC"
    }
]

 => A Local Path Segment Driver has connected to the G-Bat Network!


 => Connected Nodes:  [
    {
        "regstrationTime": 1490363430225,
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    {
        "regstrationTime": 1490363872589,
        "id": 44,
        "name": "Local Vector Driver",
        "socketID": "Ak35YcRTbjEPJpv1AAAC"
    },
    {
        "regstrationTime": 1490363944664,
        "id": 48,
        "name": "Local Path Segment Driver",
        "socketID": "CcU48SJifBDuF3RfAAAD"
    }
]


 =>  Query of Vector Knowledge Store Object received  from Local Path Segment Driver!
getting the map...
(node:11680) DeprecationWarning: Mongoose: mpromise (mongoose's default promise librar
y) is deprecated, plug in your own promise library instead: http://mongoosejs.com/docs
/promises.html
these are the results [ { _id: 5878994882065956086b5bae,
    from: 'worldModel',
    location: { x: 2, y: -23, r: 4.5 },
    to: 'planner',
    speed: 11.11,
    destination: { x: 13, y: -14, r: 2.5 },
    map:
     { rows: 16,
       columns: 16,
       gridSize: 82.2,
       traversability: [Object] },
    __v: 0 },
  { _id: 5878a0379563bd8cb4b5a60c,
    from: 'worldModel',
    location: { x: 2, y: -23, r: 4.5 },
    to: 'planner',
    speed: 11.11,
    destination: { x: 13, y: -14, r: 2.5 },
    map:
     { rows: 16,
       columns: 16,
       gridSize: 82.2,
       traversability: [Object] },
    __v: 0 },
  { _id: 5878bd2242b77f8180951780,
    from: 'worldModel',
    location: { x: 2, y: -23, r: 4.5 },
    to: 'planner',
```

Fig. 9. Communicator module sample messages



```
**** Starting the JAUS  Tester ....... ****



 => Connected Nodes:  [
    {
        "regstrationTime": 1490363430225,
        "id": 35,
        "name": "JAUS Communicator",
        "socketID": 0
    },
    {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    {
        "regstrationTime": 1490363872589,
        "id": 44,
        "name": "Local Vector Driver",
        "socketID": "Ak35YcRTbjEPJpv1AAAC"
    },
    {
        "regstrationTime": 1490363944664,
        "id": 48,
        "name": "Local Path Segment Driver",
        "socketID": "CcU48SJifBDuF3RfAAAD"
    },
    {
        "regstrationTime": 1490364472847,
        "id": 33,
        "name": "Primitive Driver",
        "socketID": "o8lcOZeS5rFwHId1AAAE"
    }
]
 dest nodeInfo [ { regstrationTime: 1490363729185,
    id: 61,
    name: 'Vector Knowledge Store',
    socketID: 'iSKiI_av-uPDQJR1AAAB' } ]


 -> nodeInfo =>  {
    "messageID": "0A20h",
    "sender": {
        "id": 33,
        "name": "Primitive Driver"
    },
    "recipient": {
        "regstrationTime": 1490363729185,
        "id": 61,
        "name": "Vector Knowledge Store",
        "socketID": "iSKiI_av-uPDQJR1AAAB"
    },
    "data": {},
    "sequenceNo": 1
}


 -> Sending 0A20h command to Vector Knowledge Store....


 ack :->   {
    "command": "0A20h",
    "sender": "Primitive Driver",
    "recipient": "Vector Knowledge Store",
    "timestamp": 1490364472983
}


  ack <-:  {
    "messageID": "4400h",
    "sender": {
        "id": 61,
        "name": "Vector Knowledge Store"
    },
    "recipient": {
        "id": 33,
        "name": "Primitive Driver"
    },
    "data": "success",
    "sequenceNo": 1,
    "timeStamp": 1490364472984
}
```

Fig. 10. Tester module sample messages

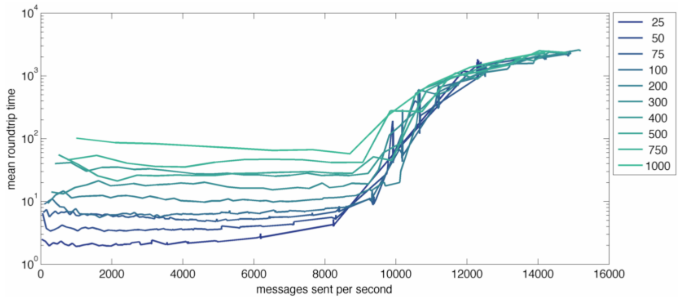


Fig. 11. Communication Framework Performance Tests

cations such as Python. This paper is not aimed at making a direct comparison between the two, but rather at analysing performances such as latency, number of message per second and number of concurrent connections for the application it is designed for which is the UGV.

In a single-threaded environment like node, scheduling tasks is handled with a queue. When the rising round trip times start happening, it means that tasks that are being added to the queue more quickly than they can be processed. Once the server reaches that point, performance of the communication framework is going to degrade very rapidly until the rate of adding new tasks falls below the jamming threshold.

Figure 11 illustrates how the communicator node manages the communications in the whole subsystem. The time in the figure is expressed in milliseconds(ms). The right hand side of the figure depict the legend according to the number of connected nodes. I can be seen that the more the number of nodes communicating, the longer the latency (round trip time). During the examination of these graphs, it is important to remember that any response time greater than 100ms represents a fatal situation if we aimed at detecting any obstacle within a meter when the vehicle is moving at the speed of 40km/h. It has to be understood that when the vehicle will cover a distance of 1.1 meter in 100ms when travelling at 40km/h.

On the graph, it can be seen that the mean round trip time start be greater than 100ms with a 1000 nodes are connected when the number of messages per second sis greater than 9000. As a result, his system will perform perfectly even with a 1000 nodes connected and sending 8000 messages per second.

X. Conclusion

All message passing and responses were demonstrated as well as the timing and logging thereof. Although Node.js is single threaded and the latency will increase with the number of connections and message sent. the system has proven to be suitable for the application in a UGV with a very small number of connected nodes. The work presented in this paper is a novel implementation of a communication framework in JavaScript and truly cross platform and light weight (single threaded). Similar communication frameworks have been implemented in C++ for the most part and not truly cross platform. It is also accessible to the majority of developer and has a natural path to the Web and Internet because JavaScript is the language of the Web. The demonstration was successful as the SBC were used to achieve all thDetails of the proceedingse expectations in terms of message communication requirements.

References

[1] Danny Kent. ”Robotic Manipulation and Haptic Feedback via High Speed Messaging with the Joint Architecture for Un-manned Systems (JAUS).” *AUVSI Unmanned Systems Conference*,2004.

[2] Galluzzo, Thomas and Danny Kent ”The openjaus approach to designing and implementing the new sae jaus standards”. *AUVSI Unmanned Systems Conference*, 2010.

[3] Touchton, Bob, et al. ”Perception and planning architecture for autonomous ground vehicles.”. *Computer 39.12* , 2006.

[4] Crane Iii Carl D. et al. ”Team cimars navigator: An unmanned ground vehicle for the 2005 darpa grand challenge.” *The 2005 DARPA Grand Challenge. Springer Berlin Heidelberg, 311-347.*, 2007.

[5] BAZN, FEDERICO, et al. ”JAUS Interface for Tools Development in the robotics field.” *Computer Science Technology Series: 245.*, 2006.

[6] Quigley, Morgan, et al. ”ROS: an open-source Robot Operating System.” *ICRA workshop on open source software. Vol. 3. No. 3.2.* , 2009.

[7] R. Elghondakly, S. Moussa and N. Badr, ”Waterfall and agile requirements-based model for automated test cases generation,” *2015 IEEE Seventh International Conference on Intelligent Computing and Information Systems (ICICIS)*, Cairo, 2015, pp. 607-612.