Hindawi Publishing Corporation
Journal of Control Science and Engineering
Volume 2014, Article ID 631487, 20 pages
http://dx.doi.org/10.1155/2014/631487

Research Article

JAUS to EtherCAT Bridge: Toward Real-Time and Deterministic Joint Architecture for Unmanned Systems

Jie Sheng,[1] Sam Chung,[1] Leo Hansel,[1] Don McLane,[1] Joel Morrah,[1] Seung-Ho Baeg,[2] and Sangdeok Park[2]

[1] *Institute of Technology, University of Washington, Tacoma, WA, USA*
[2] *Division of Applied Robot Technology, Korea Institute of Industrial Technology, Republic of Korea*

Correspondence should be addressed to Jie Sheng; shengj2@uw.edu

Received 26 June 2013; Revised 12 December 2013; Accepted 26 December 2013; Published 27 April 2014

Academic Editor: Jason Gu

The Joint Architecture for Unmanned Systems (JAUS) is a communication standard that allows for interoperability between Unmanned Vehicles (UVs). Current research indicates that JAUS-compliant systems do not meet real-time performance guidelines necessary for internal systems in UVs. However, there is a lack of quantitative data illustrating the performance shortcomings of JAUS or clear explanations on what causes these performance issues or comparisons with existing internal communication systems. In this research, we first develop a basic C++ implementation of JAUS and evaluate its performance with quantitative data and compare the results with published performance data of Controller Area Network (CAN) to determine the feasibility of the JAUS standard. Our results indicate that the main reason of JAUS's poor performance lies in the latency inherent in the hierarchical structure of JAUS and the overhead of User Datagram Protocol (UDP) messages, which has been used with JAUS and is slower than the high-speed CAN. Additionally, UDP has no scheduling mechanism, which makes it virtually impossible to guarantee messages meeting their deadlines. Considering the slow and nondeterministic JAUS communication from subsystems to components, which is JAUS Level 3 compliance, we then propose a solution by bringing Ethernet for Control Automation Technology (EtherCAT) to add speed, deterministic feature, and security. The JAUS-EtherCAT mapping, which we called a JEBridge, is implemented into nodes and components. Both quantitative and qualitative results are provided to show that JEBridge and JAUS Level 3 compliance can bring not only interoperability but also reasonable performance to UVs.

1. Introduction

The United States Congress has mandated that by 2015 30% of all military vehicles must be Unmanned Vehicles (UVs) [1]. One major roadblock, however, is that most UVs are made by various manufacturers and come in many different makes and models. Communication with UVs, either between the UV and an Operator Control Unit (OCU) or another UV, is typically proprietary. As a result, UVs that coordinate on missions must be specifically designed to communicate with each other. If a specific UV necessary for a mission is unavailable, there is no way to substitute another UV. In addition, OCUs come in many forms and the person charged with operating these vehicles must learn a new interface for each UV.

The Joint Architecture for Unmanned Systems (JAUS) is an initiative by the United States Department of Defense (DoD) to deal with interoperability issues between UVs. If all UVs used the same communication standard, any UV capable of performing a mission could be used, regardless of the manufacturer. By standardizing communication in UVs with the same functionality and commands, OCUs could be more consistently designed or even customized for the preference of operators.

The JAUS Working Group (JWG) was the first body tasked with developing and maintaining the JAUS standard [2]. Since then, the Society for Automotive Engineers (SAE) has taken over responsibility for JAUS [3].

Interoperability between UVs is an enticing prospect but there is another aspect to consider, performance. JAUS, first and foremost, is for use in mission critical vehicles for the military. Communication is not limited to tactical information such as mission planning. JAUS also has been

designed for facilitating communication between the control systems of UVs such as motors and braking systems, among others. Although interoperability is enticing, it cannot trade off with performance. Previous research indicates that real-time communication necessary in UVs cannot be achieved with JAUS in its current state [2, 4]. Other work regarding JAUS focuses on the interoperability aspect and does not mention issues with performance [5, 6]. In either case, quantitative data illustrating the performance shortcomings of JAUS are not present. But even with hard data we need performance measurements that are necessary in real-time, mission-critical vehicles with which we compare our results. JAUS is meant to be used in a variety of ground, air, and sea UVs, including passenger-style vehicles. As such, in order for JAUS to be considered as a communication standard for internal controls inside this type of vehicle, it would need to perform the same as, or better than, the current communication system which is used inside passenger vehicles.

This paper will investigate two main problems.

(1) Is JAUS a suitable communication standard for internal controls of passenger vehicles? If not, can quantitative data be presented to show the performance shortcomings of JAUS?

(2) If the JAUS is used to guarantee the interoperability between UVs, is there a solution to improve the communication performance?

Correspondingly, the two main contributions of the paper are as follows.

(1) We provide quantitative data illustrating that JAUS lacks real-time facilities, which was claimed (without the data support though) in [7]. The Controller Area Network (CAN) is the de facto standard for internal communication in passenger vehicles, controlling everything from luxuries like automatic windows and locks to necessities such as engine speed and antilock brakes [8]. By comparing the performance of JAUS based upon User Datagram Protocol (UDP) with that of CAN, the shortcomings of JAUS are quantified, and a seminal work [9] in the field of performance analysis of CAN systems is used as a benchmark for the performance analysis of JAUS.

(2) We design a JAUS-EtherCAT mapping, which we called a JEBridge, and implement it as well as its communication into JAUS Compliance Level 3 to improve the performance of JAUS communication. Due to the fact that JAUS lacks real-time facilities, but real-time, closed loop control design requires deterministic timing constraints, we utilize Ethernet for Control Automation Technology (EtherCAT) to avoid message traffic and thus improve the transfer rate. We provide scientific reference both quantitative and qualitative that shows how EtherCAT can help mitigate some issues on JAUS.

We note here that JAUS is a higher level protocol than CAN; so one could implement a JAUS layer over a CAN link/physical layer. However, the CAN bus is slow and has no real-time guarantees, and mapping the rich variety of JAUS messages to CAN's limited data fields is awkward. EtherCAT, however, can offer real-time guarantees and is faster. That is the motivation behind our JEBridge solution.

The rest of this paper proceeds as follows. Section 2 examines related work in the areas of JAUS and CAN and JAUS performance improvement. A brief review of concepts on nondeterministic and deterministic control, as well as EtherCAT, will be given in Section 3, followed by a detailed description about how CAN works in Section 4. Section 4 also discusses the two CAN research papers and their results, which will be used for comparison with the results of our experiments with JAUS. A detailed introduction of JAUS is given in Section 5. Section 6 explores the JAUS standard in greater detail first, then discusses our JAUS implementation, and further outlines the tests we perform for our experiments and the results we obtained. In Section 7, we first introduce JAUS Level 3 Compliance using JEBridge and its implementation; we then analyze the performance and the deterministic control of the JAUS Level 3 Compliance architecture using JEBridge. Section 8 concludes this paper.

2. Related Work

Current JAUS research indicates that performance is a significant problem. In [10], the authors created the KUL-1 gas-powered vehicle to test their JAUS implementation. The vehicle has navigation control, obstacle detection, path planning, and vehicle control capabilities. The authors point out that the latency in the vehicle is unacceptably high.

Efforts have been made to improve the JAUS performance. The authors of [4] created a partially JAUS-compliant communication framework for UVs. The primary focus of the framework is communication internal to a UV as opposed to communication between UVs. The authors chose not to implement a fully JAUS-compliant framework due to latency issues with JAUS. The issue of latency in fully JAUS-compliant systems also is noted in [2]. The authors created a UV called the Armadillo. Due to high latency caused by routing messages through the node, the authors removed it, opting to route messages directly to components. This conflicts with interoperability and full JAUS compliance but gave Armadillo improved performance. In [10], the authors point out that the latency in the vehicle is unacceptably high.

Clearly, creating fully JAUS-compliant vehicles is a difficult task. The primary purpose for JAUS, at least at this stage of development, is interoperability and not performance, since the JAUS documentation makes no mention of performance requirements [2]. In order for JAUS to be adopted as a viable communication standard, performance must be a priority. To assess the performance of JAUS-compliant systems, we must first obtain quantitative data in regards to the performance of JAUS. Then we need a suitable performance benchmark representative of performance requirements in vehicles today.

The benchmark we will use is CAN. The work in [9] is important in the field of CAN performance. In this paper, Tindell and Burns adapt CPU scheduling techniques [11] to

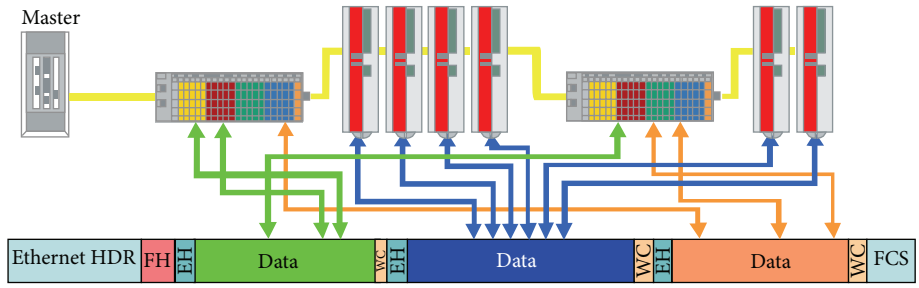


Figure 1: EtherCAT frame processing [13].

the problem of bounding performance in CAN systems. The results of their work will be the performance standard that we will use to assess the feasibility of JAUS as a communication standard for UVs.

We note that Albus et al. [12] developed a reference model for Unmanned Vehicle systems called 4D/RCS. They stated that 4D/RCS architecture is naturally adaptable to the DoD/Army standards in a combined domain of vehicle systems, combat support, and software engineering. 4D/RCS provides a methodology by which military systems, that meet the operational requirements in the Joint Architecture for Unmanned Ground Systems (JAUGS) Domain Models, can be engineered to meet the performance specifications defined in the JAUGS Reference Architecture. Unlike JAUS, which does not provide the guidelines for functionally organizing components, 4D/RCS describes in detail the functions and associated interfaces necessary for each node to provide sensory processing, world modeling, knowledge management, value judgment, and behavior generation. Furthermore, it describes that the functional loop should be replicated throughout all the nodes inside a system. In fact, 4D/RCS can serve as organizational frameworks for the JAUS components and messages, which, however, is beyond the scope of this research work.

3. Background

In this section, we will briefly review the concepts of deterministic control and EtherCAT. We will discuss JAUS and CAN in more details in separate sections soon.

3.1. Deterministic Control. In a local area network using User Data Protocol (UDP) over Ethernet, timing is nondeterminate. Despite that, on the time scales in which humans perceive events, the internet protocol is generally fast enough. For lower level systems, however, the situation is different. Latency still has to be good enough. But here "good enough" is far more stringent. For instance, motion controls of robotic arms or traction motors have closed loop control cycles in the millisecond range. They continually sample sensors and make adjustments to actuator drive. Deterministic timing with bounded latency is an absolute requirement.

Through our research we found that JAUS's poor performance lies in the latency inherent in the hierarchical structure of JAUS and the overhead of UDP messages, which has been used behind JAUS. Moreover, UDP has no scheduling mechanism, which makes it virtually impossible to guarantee messages meeting their deadlines and thus cannot meet the deterministic timing requirement for lower level closed loop control systems. Considering the slow and nondeterministic JAUS communication from subsystems to components, our solution is to introduce Ethernet for Control Automation Technology (EtherCAT) to add speed, deterministic feature, and security.

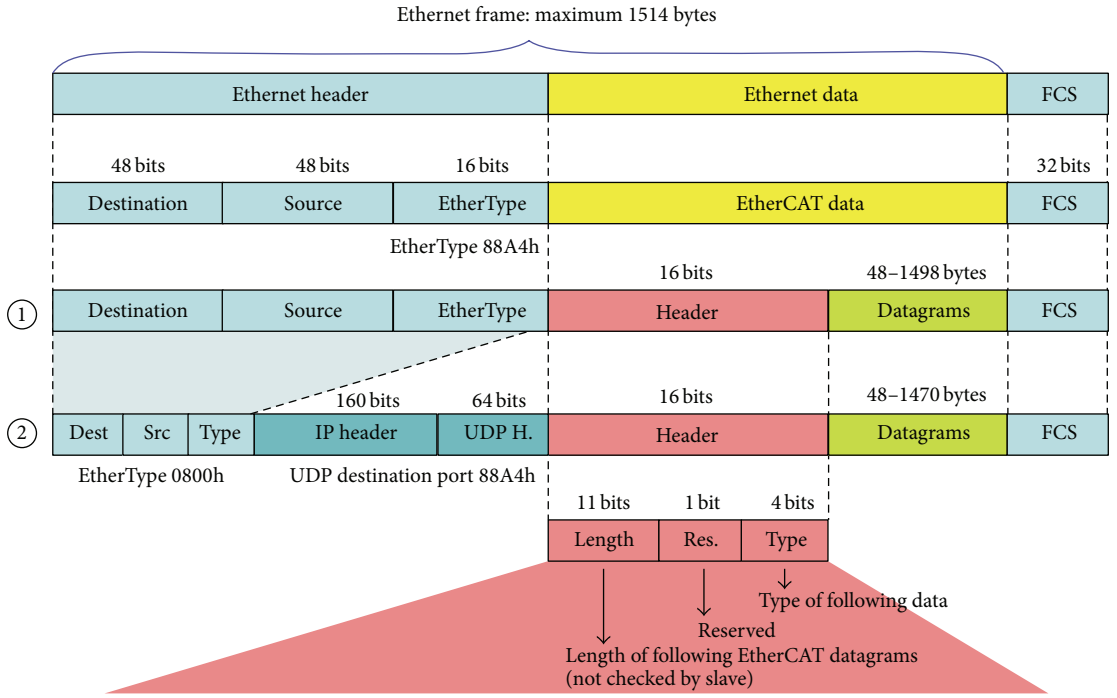
3.2. EtherCAT. EtherCAT is a real-time, high speed, and flexible Ethernet based protocol. In comparison to other Ethernet based communication solutions, EtherCAT utilizes the available full duplex bandwidth in a very efficient way because it implements a "processing on the fly" approach. The Ethernet frames in Figure 1 are sent by a master device and read and written by all EtherCAT slave devices while they are passed from one device to the next [13].

The EtherCAT protocol is optimized for process data which is embedded in the standard IEEE 802.3; see Figure 2 for Ethernet and EtherCAT frame structure.

The Ethernet header consists of Destination (6 bytes), Source (6 bytes), and EtherType (2 bytes). Also, it shows an Ethernet frame using the Ether type 0x88A4. We are interested in the first case (the circled number 1 in Figure 2), which is the EtherCAT protocol. The second case (the circled number 2 in Figure 2) is for a case when we want to send an EtherCAT packet over a UDP. The EtherCAT protocol consists of the EtherCAT protocol header (2 bytes) which contains the EtherCAT frame size in bytes (11 bits) and a protocol type (4 bits set to 1 for EtherCAT) followed by EtherCAT telegrams, which is shown in Figure 3. Each EtherCAT telegram starts with a telegram header (10 bytes) followed by the process dataand is terminated with a working counter (2 bytes) [7, 13].

4. Controller Area Network (CAN)

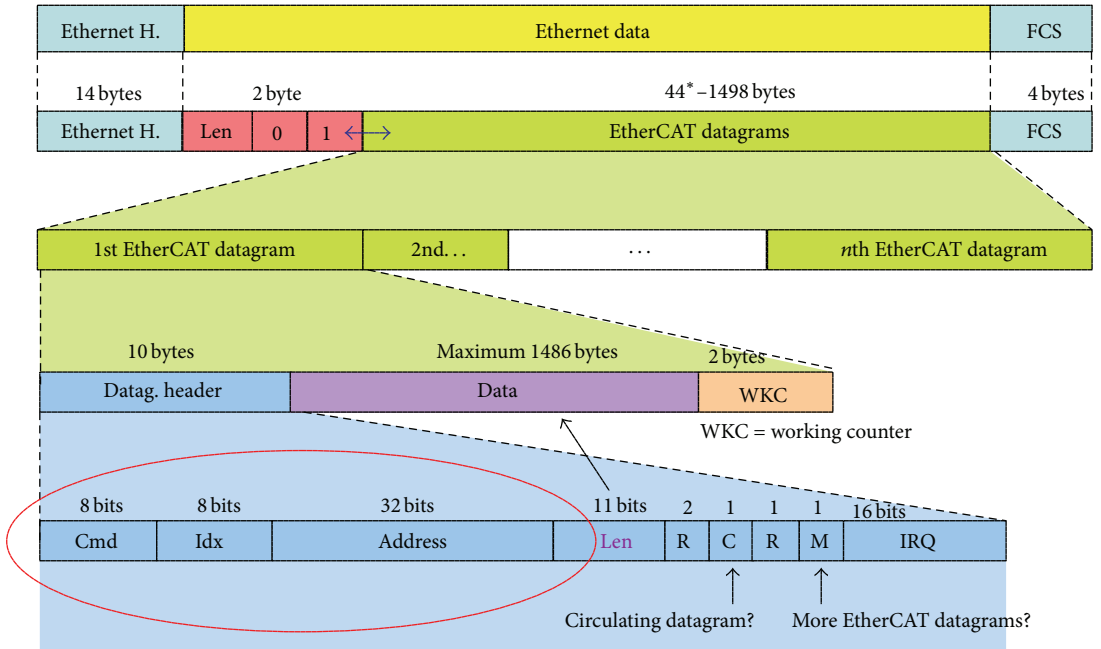
4.1. The CAN Standard. The Bosch company began development of CAN in 1983 [14]. By providing a standardized bus communication network, the problems brought by point-to-point communication using wires have been greatly reduced [8, 15]. As a serial communication standard used in passenger vehicles, CAN supports communication between Electronic Control Units (ECU) interfacing with sensors and actuators



(1) Simple EtherCAT communication

(2) EtherCAT communication over internet

Figure 2: EtherCAT frame structure (slide 28 of [7]).



*add 1–32 padding bytes if Ethernet frame is less than 64

Figure 3: EtherCAT datagram (slide 33 of [7]).

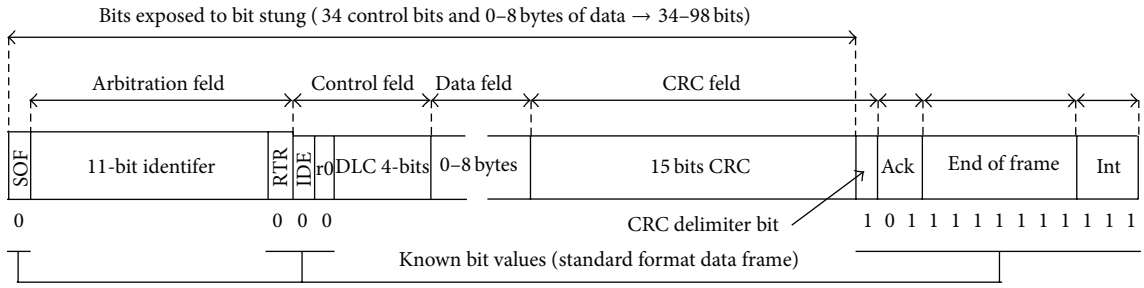


Figure 4: Standard CAN message format [17].

as well as host Central Processing Units (CPUs) and the bus. Facilitating the operation of everything from motors and braking systems to windshield wipers and door locks, CAN is currently the most popular communication standard in the automotive industry with annual sales exceeding 400 million [8] and thus provides a suitable benchmark with which we compare the JAUS standard.

ECU wired to the bus act as an interface between the bus and a local CPU which process data or initiate outgoing messages. Transmission of messages occurs at varying speeds depending on the needs of the network. Class A networks are less than 10 kb/s, Class B networks operate at 10–125 kb/s, and Class C networks range from 125 kb/s–1 Mb/s [8].

There are two versions of the protocol with the most significant difference in terms of the length of the identifier field: one protocol version has an identifier field of 11 bits, which is termed Standard CAN Version 2.0A, while the other has a length of 29 bits called Extended CAN Version 2.0B [16]. Further discussion of CAN will focus only on the Standard CAN since the benchmark analysis we will be using evaluates that format. Figure 4 illustrates the Standard CAN message format.

The maximum message size of Standard CAN messages, with no stuff bits, is 111 bits. There are 44 control bits, a maximum of 8 bytes, or 64 bits of data, and 3 interframe bits used in between message transmission.

The Standard CAN uses six bits of either all 0 s or all 1 s to indicate an error. It is problematic if a nonerror message contains a sequence of six 0 s or 1 s since the message will be viewed as an error. To overcome this, CAN utilizes a bit-stuffing scheme. During the creation of a message, if five bits in a row are all 0 s or all 1 s, the sixth bit is a stuff bit of the opposite polarity. Nodes receiving the message will automatically remove the stuff bits before processing the message. This solves the problem of differentiating between error and nonerror messages but at the cost of increased message size in certain messages. Messages with sequences of six or more consecutive 0 or 1 bits will be longer than messages with fewer, or no, sequences of six or more 0 or 1 bits due to the added stuff bits. As a result, the lengths of CAN messages are variable which means some messages will take longer than others to send.

The maximum size of a Standard CAN message, including stuff bits, can be easily calculated using equation (1) from [9], where s_m is the number of data bytes in the message. The number of control bits subject to bit stuffing is 34 and there are a total of 47 control bits:

$$8s_m + 47 + \left\lfloor \frac{34 + 8S_m}{5} \right\rfloor. \quad (1)$$

Another type of message is the Remote Transmission Request (RTR) message. This is used to signal a receiving node which has the same identifier to transmit the data that has for the identifier. This type of message is used for data which is used infrequently and does not factor into the analysis of [9].

4.2. CAN Performance Analysis of Tindell and Burns [9]. The performance of CAN is generally measured in terms of worst-case response time. This measurement begins when a task first starts to queue a message and ends when the message arrives at its destination [9]. The problem of bounding response times in a CAN system is a scheduling problem since many messages are vying to be sent, all with deadlines of when they must be received. Consequently, bounding response times in CAN is a difficult problem and a great body of work deals with the subject with varying methods and results.

For our experiments, we have selected the work of Tindell and Burns [9]. Although the schedulability analysis in [9] has been revised and replaced by that in [18], we use [9] as our benchmark reference for a couple of reasons. First, the work is cited in over 200 papers dealing with performance and scheduled in CAN. In addition, the research has influenced the design of the Motorola msCAN peripheral and the Volcano Network Architect CAN scheduled analysis tool. The second reason is that this work does not account for lost packets, error transmissions, and so forth. In our research we evaluate the simplest form of JAUS and this work provides a simple CAN analysis suitable for comparison.

The test case for this work was developed for a Class C network according to the performance guidelines in [3]. The analysis does not account for error transmissions or RTR messages. As a result, the analysis is simpler since all messages are transmitted periodically and no sporadic messages interfere with the analysis.

The worst-case response time for a message "m" is calculated using (2), where R_m is the worst-case response time for message m:

$$R_m = J_m + w_m + C_m, \quad (2)$$

where J_m is the queuing jitter for message m and is the time required to place a message in the queue. w_m is the worst-case time in which a message sits waiting in the queue for a lower-priority message to finish transmitting or for messages with a higher priority to transmit. C_m is the time required to actually send the message. So the worst-case time response is the time required to load the message into the queue, the time spent waiting to gain access to the bus, and the time to transmit the message.

We note that Table 1, contains results from the analysis of Tindell and Burns [9]. Their analysis made use of the SAE benchmark for CAN systems. In the SAE benchmark, there are 53 different messages. Some are sent periodically, while others are sporadic. For the sake of simplicity, Tindell and Burns have given all messages a period.

In the SAE benchmark, each message has a unique number to identify the message type. The deadline of each message does not correspond with the numbering. So in Table 1, while the numbering in the Signal number column is not sequential, it is in order of priority.

The messages in Table 1 are ordered in highest-to-lowest priority from top to bottom. The column "T_m" represents the period of message "m," which is the smallest time between consecutive queuing events for message m. An "x" mark indicates that a message misses its deadline.

In the original work, fifty-three messages were included. For our analysis, the first eighteen will be more than sufficient. We have defined all the parameters which appear except for the period "T_m." Tasks running on local CPUs are responsible for initiating the transmission of messages. The events which trigger a task to transmit a message do so within a time frame. The smallest difference in the time of consecutive executions of a task is the period.

We will briefly demonstrate how the values in Table 1 are obtained. We will calculate the worst-case response time of the first row under the column corresponding to 250 Kbit/s. The calculation comes directly from (2). The queuing jitter, or J_m, is not factored into the calculations in Table 1. However, the rest of the calculation remains intact.

w_m is the longest time which a message will have to wait for the wire to be clear for transmission. Since the first row corresponds to the message with the highest priority, the longest it will have to wait is for one message. This message is, in the worst case, the maximum message size. To calculate this value, we use (1), filling in 8 for s_m. This will give us the number of bits which will be sent, which is 130 bits. To calculate the time required to send these bits we simply look to the speed of the bus, 250 Kbits/s. 1 bit is sent in 1/250000 s or 4 μs. Multiplying this by 130 we find that in the worst case, the message with the highest priority will have to wait 0.00052 s or 0.52 ms.

The next calculation is C_m which is just the time needed to send the current message. Again we use (1), except that we use 1 for s_m since the size of the message is only one byte. The number of bytes in this message is 63 and multiplying this by the time required to send a single byte, 4 μs, we find the time is 0.000252 s or 0.252 ms. Adding this value to the value from the previous paragraph we find the worst-case response time for the message with the highest priority on a bus with a speed of 250 Kbits/s is 0.772 ms. The subsequent values in the column can be calculated by finding the value of C_m as we just did and adding to the worst-case response times of all of the previous messages.

5. Joint Architecture for Unmanned Systems (JAUS)

JAUS was created to be a universal UV communication standard to allow for interoperability among all UVs made by different manufacturers including avionic, marine, and ground vehicles which are compliant with the standard. The goal is for JAUS to be used for tactical functionality, such as mission planning and coordination with other UVs during the execution of missions, as well as internal communication for controlling engines, braking systems, and so forth. Current systems which are responsible for the low-level control of vehicles, such as CAN, have strict performance requirements. UVs developed for military use also have strict requirements since the UVs operate in hostile environments. In order for JAUS to be considered as a viable standard, it must be able to meet current performance standards for mission-critical vehicles.

5.1. JAUS Structure. The JAUS network is structured as a hierarchy, which is shown in Figure 5. At the top is the system, which is made up of all of the UVs which will use the network to communicate. Each UV is considered a subsystem. A subsystem is the highest element inside a UV. The software operating at the subsystem level, which processes incoming messages and forwards outgoing messages, is called the Communicator. All JAUS messages entering and exiting a UV must pass through the Communicator [19]. If JAUS messages are exchanged only between subsystems, the system is Level I Compliant. Beneath the subsystem is the node which is a computer or embedded system onboard a UV. The software operating at the node level is called the Node Manager, which is responsible for routing messages to the components it oversees and for routing outgoing messages from its components to the subsystem. Since JAUS messages are exchanged between both subsystems and nodes, the system is Level II Compliant. At the lowest level is the component which is software that directly interacts with hardware such as a motor, camera, or steering. In this case, the system is Level III Compliant since JAUS messages are exchanged between subsystems, nodes, and components.

An incoming message to a JAUS-compliant UV must first communicate with the subsystem. The message is then routed to the appropriate node, which forwards the message to the component that should receive it. When a message is sent out to another UV, the message must be routed upwards. If a component is sending a message to another subsystem, it must first send the message to its managing node. The node forwards the message to the subsystem which then sends the message to the subsystem of the intended UV. From there, the message is propagated down to the intended recipient.

Internal communication must also be routed through the hierarchy and no direct interaction between components is

Table 1: CAN performance results [9].

Signal number	Size/bytes	J/ms	T/ms	D/ms	R (125 Kbit/s)	R (250 Kbit/s)	R (500 Kbit/s)	R (1 Mbit/s)
14	1	0.1	50.0	5.0	1.544	0.772	0.386	0.193
9	1	0.2	5.0	5.0	2.048	1.024	0.512	0.256
49	1	0.2	5.0	5.0	2.552	1.276	0.638	0.319
42	1	0.2	5.0	5.0	3.056	1.528	0.764	0.382
8	1	0.1	5.0	5.0	3.560	1.780	0.890	0.445
7	1	0.1	5.0	5.0	4.064	2.032	1.016	0.508
43	1	0.1	5.0	5.0	4.568	2.284	1.142	0.571
11	1	0.1	5.0	5.0	x 5.072	2.536	1.268	0.634
32	1	0.1	5.0	5.0	x —	2.788	1.394	0.697
29	1	0.3	10.0	10.0	x 10.112	3.040	1.520	0.760
30	1	0.4	10.0	10.0	x —	3.292	1.646	0.823
53	1	1.5	50.0	20.0	x 25.232	3.544	1.772	0.886
48	1	1.4	50.0	20.0	x 29.768	3.796	1.898	0.949
46	1	1.3	50.0	20.0	x 39.344	4.048	2.024	1.012
44	1	1.2	50.0	20.0	x 39.848	4.300	2.150	1.075
40	1	1.1	50.0	20.0	x —	4.552	2.276	1.138
39	1	1.0	50.0	20.0	x —	4.804	2.402	1.201
27	1	0.9	50.0	20.0	x —	7.072	2.528	1.264
38	1	0.9	50.0	20.0	x —	7.324	2.654	1.327
37	1	0.8	50.0	20.0	x —	7.576	2.780	1.390
52	1	0.8	50.0	20.0	x —	7.828	2.906	1.453
26	1	0.8	50.0	20.0	x —	8.080	3.032	1.516
35	1	0.7	50.0	20.0	x —	8.332	3.158	1.579
51	1	0.7	50.0	20.0	x —	8.584	3.284	1.642
22	1	0.7	50.0	20.0	x —	8.836	3.410	1.705
34	1	0.6	50.0	20.0	x —	9.088	3.536	1.768
20	1	0.6	50.0	20.0	x —	9.340	3.662	1.831
50	1	0.6	50.0	20.0	x —	9.592	3.788	1.894
31	1	0.5	50.0	20.0	x —	9.844	3.914	1.957
47	1	0.5	50.0	20.0	x —	12.616	4.040	2.020
28	1	0.5	50.0	20.0	x —	12.868	4.166	2.083
19	1	0.5	50.0	20.0	x —	13.120	4.292	2.146
25	1	0.4	50.0	20.0	x —	13.372	4.418	2.209
17	1	0.4	50.0	20.0	x —	13.624	4.544	2.272
45	1	0.4	50.0	20.0	x —	13.876	4.670	2.335
24	1	0.3	50.0	20.0	x —	14.128	4.796	2.398
16	1	0.3	50.0	20.0	x —	14.380	4.922	2.461
18	1	0.3	50.0	20.0	x —	14.632	6.056	2.524
41	1	0.3	50.0	20.0	x —	14.884	6.182	2.587
23	1	0.2	50.0	20.0	x —	17.152	6.308	2.650
15	1	0.2	50.0	20.0	x —	17.404	6.434	2.713
6	1	0.9	100.0	100.0	x —	17.656	6.560	2.776
4	1	0.8	100.0	100.0	x —	17.908	6.686	2.839
2	1	0.7	100.0	100.0	x —	18.160	6.812	2.902
1	1	0.6	100.0	100.0	x —	18.412	6.938	2.965
12	1	0.4	100.0	100.0	x —	18.664	7.064	3.028
10	1	0.2	100.0	100.0	x —	18.916	7.190	3.091
36	1	1.7	1000.0	1000.0	x —	19.168	7.316	3.154
33	1	1.6	1000.0	1000.0	x —	19.420	7.442	3.217
13	1	1.2	1000.0	1000.0	x —	19.672	7.568	3.280
5	1	1.1	1000.0	1000.0	x —	22.444	7.694	3.343
3	1	1.0	1000.0	1000.0	x —	22.696	7.820	3.406
21	1	0.3	1000.0	1000.0	x —	22.948	7.946	3.469

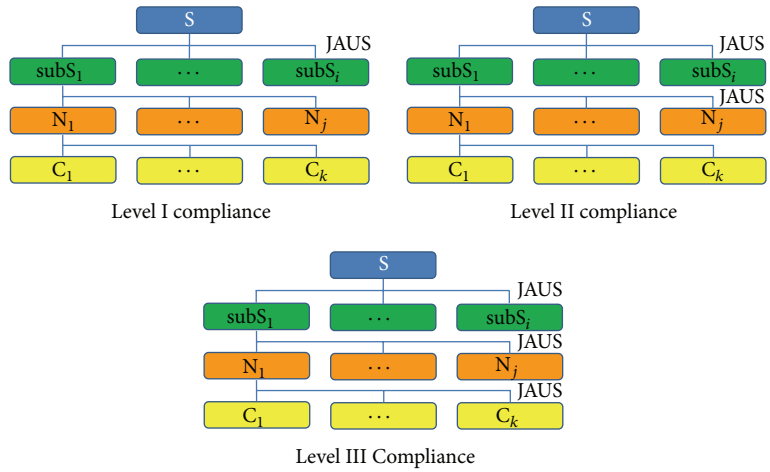


Figure 5: JAUS hierarchy and Compliance Level. S: system, subs: subsystem, N: node, C: component.

allowed. For instance, if a component belonging to a node needs to communicate with another component belonging to the same node, the message must first go through the node. Similarly, messages going from one node to another in the same subsystem must first be routed through the subsystem. This hierarchical structure for message routing in Figure 5, as opposed to direct communication between components, contributes in part to the performance issues of JAUS [2] and is something we will address in our experiments.

All components, including the Communicator and Node Manager, in a JAUS-compliant system are required to have a unique name and JAUS IP address [19]. The name can be an upper- or lower-case letter, a number, or an underscore. The JAUS IP is a dotted-decimal value, the same as a standard IP address. The only difference is that JAUS IPs are used internally to identify JAUS components and generally cannot resolve to an IP address.

5.2. JAUS Messages. There are currently one hundred fifty-seven JAUS messages [20]. The JAUS message header is 16 bytes of data. The destination and source of JAUS messages are encoded in the header as four bytes each, one byte for each octet. It also contains a 16-bit field, the Command Control field, which designates the type of message, and a field for data length, among others. The size of JAUS messages is variable, restricted only by the frame size of the protocol in which the message is encapsulated. The JAUS header is depicted in Figure 6.

JAUS currently allows for "experimental" messages. Developers who feel unable to perform all necessary functionality with the current one hundred fifty-seven messages can create their own messages within the experimental message range. Experimental messages are available but should be used sparingly as they are not part of the standard and interoperability between JAUS-compliant UVs may suffer if experimental messages are present in some and not others.

JAUS currently has not defined a communication protocol. It is believed, however, that UDP and RS-232 will be claimed as the protocols of choice for JAUS [22]. Additionally, transmission of audio/video data is not yet supported [21] and the specifics of mission planning have yet to be defined. Another very important facet the JAUS standard has yet to address is performance.

5.3. JAUS Communication. One of the principal goals of JAUS is to provide a level of interoperability between intelligent systems that has been missing in the past. Towards this end, JAUS defines functional components with supporting messages but does not impose regulations on the systems engineer that govern configuration.

JAUS does have one absolute, unwavering requirement that can have an effect on configuration. To achieve the desired level of interoperability between intelligent computing entities, all messages that pass between JAUS defined components (over networks or via airwaves) shall be JAUS compatible messages [19]. In order to present a JAUS communication, we provide a diagram shown in Figure 7.

This figure depicts two JAUS defined components in the right two big boxes and a dedicated hardware device in the left small box. An explanation of the figure is given below.

(1) Node 1 is configured as a positioning unit and is dominated by the Global Pose Sensor component. The Differential Global Positioning System (DGPS) is configured as a dedicated device within the node. The DGPS is setup in streaming mode so that positioning data is sent out over Recommended Standard 232 (RS-232) at some regular interval.

(2) Node 2 is configured as a Path Driving unit and is dominated by the Global Path Segment Driver component. It relies on Global Pose Sensor messages to perform its path-tracking task.

(3) The Global Pose Sensor component task reads shared memory whenever necessary to get the latest DGPS information. (Important note: so far all of the data

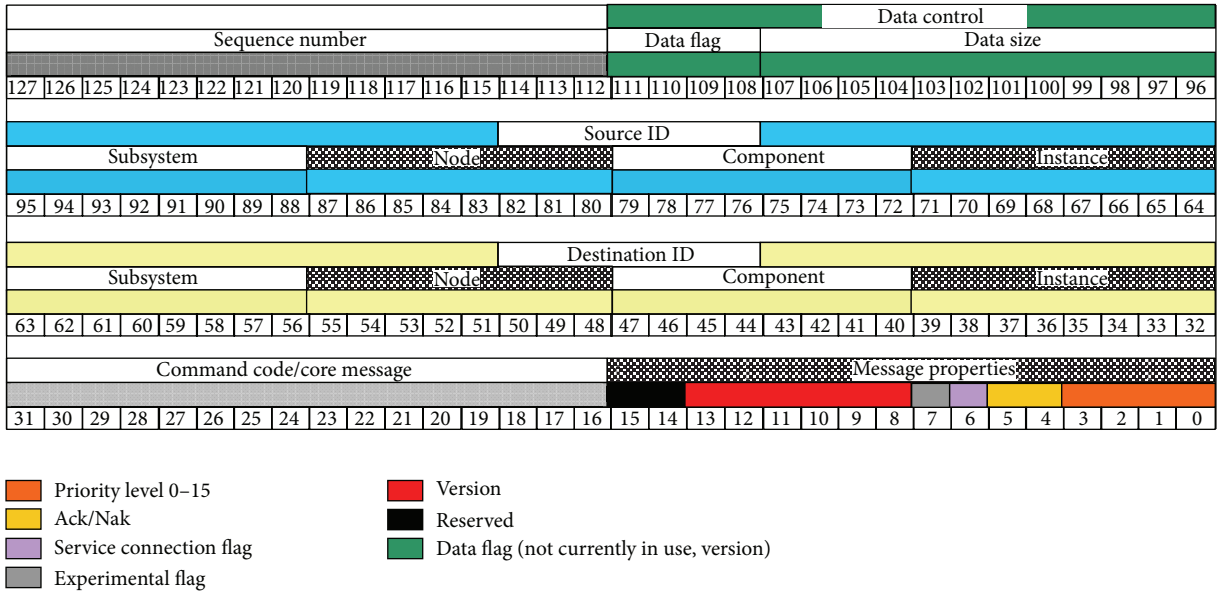


Figure 6: JAUS message header [21].

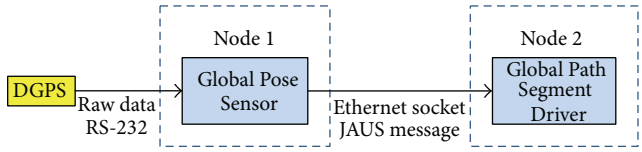


Figure 7: JAUS communication example configuration.

is in raw (non-JAUS) format because typically dedicated sensors, like the DGPS, do not support JAUS messages.)

(4) When the Global Pose Sensor component completes its position calculations, it sends a JAUS formatted Global Pose Sensor message to Node 2. This sequence loops continually. Notice that although the streaming data coming from the DGPS to the I/O server and on to shared memory is not in JAUS format, the interoperability rule remains unbroken. Only when data is sent between JAUS defined components, must it be formatted into a JAUS compatible message.

6. JAUS Performance Evaluation

6.1. JAUS Implementation. For our experiments, we created a simple JAUS implementation focusing on the kernel, or core part, of JAUS, called "k-OpenJAUSC++," written in C++ for the Windows operating system. The project contains two subprojects: The "Lib" subproject and the "Communicator" subproject. The "Lib" subproject contains classes dealing with constructing and processing messages and is static. Once the message creation and processing functions have been created, there is really no need to modify them. Since the majority of work occurs in the "Communicator," we compile it as a Dynamic Link Library (dll) to simplify the Communicator. The "Communicator" is much more dynamic, responsible obtaining the configuration of the JAUS system, running the server, and processing messages. The class diagram for the "Communicator" and "Lib" and the sequence diagram for the "Communicator" receiving a message appear in Figures 8, 9, and 10, respectively.

It is already known that existing, full-featured JAUS implementations have performance issues. The question that arises is what causes these performance issues. We have opted to create a simple JAUS implementation in order to evaluate the performance of the JAUS standard at its core. By testing the performance on a simple JAUS implementation, determining the causes of the performance is much simpler. We believe that the main cause of latency in the JAUS standard is due to the hierarchical message passing scheme of JAUS coupled with the use of UDP as the transport protocol. As such, our JAUS implementation focuses on creating a JAUS message, sending it using the UDP protocol in compliance with the JAUS standard, and processing the message when it is received. This allows us to more easily evaluate the effects of the hierarchical JAUS message passing scheme and the use of UDP as the protocol. Only one application is necessary for use as subsystems, nodes, and components. Each application has a multithreaded server which is capable of creating and sending, routing, or processing JAUS messages. If a message is received that is intended for another subsystem, node, or component, it is forwarded to the next element in the path to the destination. If a message is received by the intended recipient, the message is processed. As a result, our code is reusable, requiring minimal modifications only to instances which need to send or receive a message.

The network of the JAUS system is defined in an XML file. Each subsystem, node, and component application that is running has a copy of the XML file so it is able to determine where to send JAUS messages. This requires

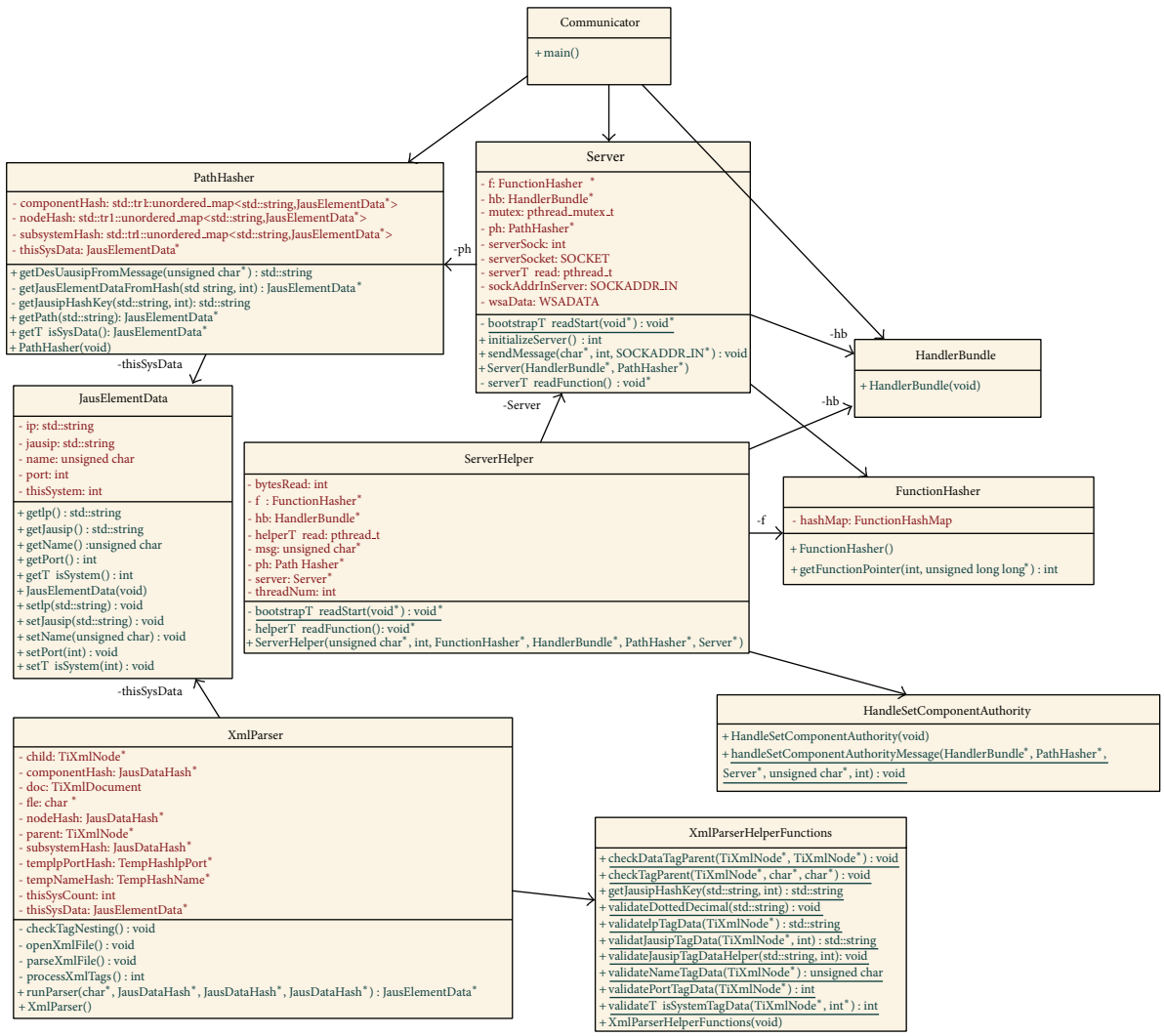


Figure 8: Class Diagram for Communicator.

the entire JAUS system to be mapped out ahead of time. The JAUS standard does allow for broadcasting for discover of components, but for simplicity, we have opted to use static system configurations. The XML file contains the JAUS name, IP address, receiving port, and JAUS IP address of each subsystem, node, and component that is in the system. Additionally, a fifth tag labeled "thissystem" contains a value of 1 to indicate the information for the current application and 0 for all others. Algorithm 1 contains a sample XML file used in our implementation.

When our implementation begins, the "Communicator" is the entry point into the application. This calls the "PathHasher," which is responsible for storing the system information from the XML file, the Server, which listens for incoming messages, and the "HandlerBundle," which is used for passing data, API access, and so forth, to the functions which process messages.

The "XMLParser" class is responsible for extracting the data from the configuration XML file. For the core XML parsing capabilities, we used the open-source TinyXML [23] code. The data for each subsystem, node, and component is stored in a "JausElementData" instance. The "XMLParser" provides the "PathHasher" with a pointer to the "JausElementData" instance which holds the data for itself to allow for each application to quickly determine its own data.

All of the "JausElementData" instances are stored in one of three unordered maps, or hash maps, one for subsystems, one for nodes, and the third for components. The "PathHasher" maintains these three hash maps for use in determining where a JAUS message should be sent next to arrival at its destination in compliance with the JAUS standard.

The "PathHasher" resolves where to route messages in the following way. The hash keys resolve to an instance of the "JausElementData" class which contains the data for a single subsystem, node, or component specified in the XML file. In the hash map for the subsystems, the key is the first octet in the destination JAUS IP as a string. For the nodes, the key value is the first octets in the JAUS IP, including the decimal

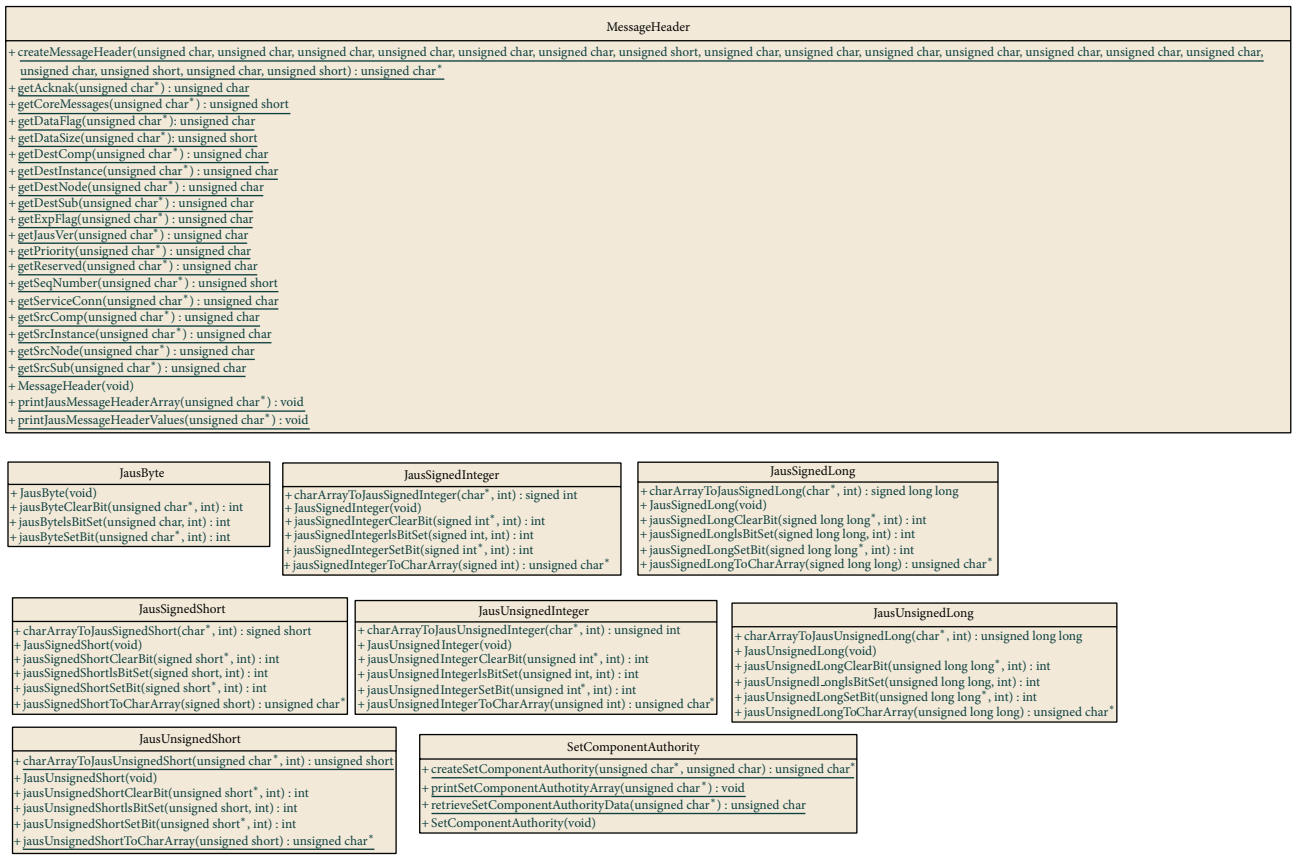


Figure 9: Class Diagram for Lib.

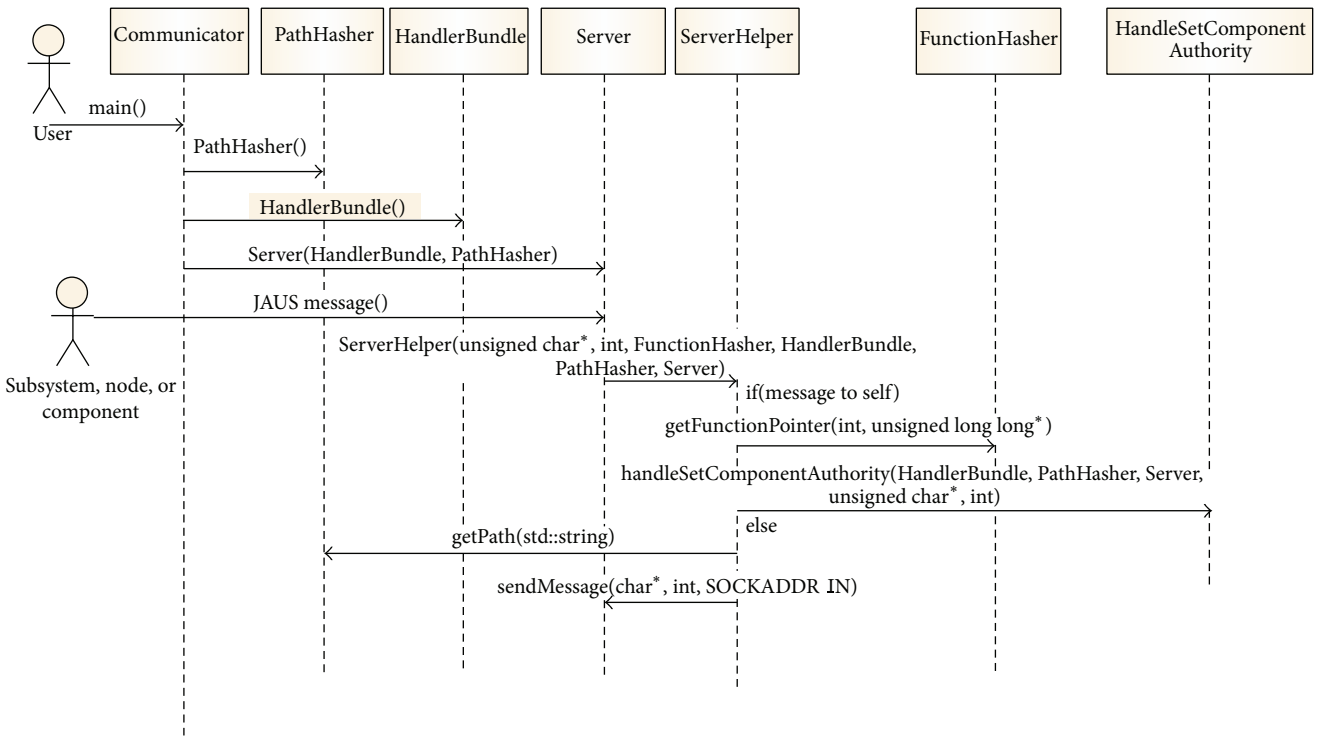


Figure 10: Sequence diagram of Communicator receiving a message.

```
<?xml version="1.0"?>
<system>
        <subsystem>
                <name>a</name>
                <ip>128.208.244.165</ip>
                <jausip>1.0.0.0</jausip>
                <port>39777</port>
                <thissystem>1</thissystem>
        </subsystem>
        <subsystem>
                <name>b</name>
                <ip>128.208.244.229</ip>
                <jausip>2.0.0.0</jausip>
                <port>39777</port>
                <thissystem>0</thissystem>
        </subsystem>
</system>
```

Algorithm 1: JAUS configuration file.

point, as a string. For the components, the key value is the JAUS IP, in its entirety, as a string.

To see why this works, consider a node which receives a message. The node must determine whether to send the message to its subsystem, process the message, or forward the message to one of the components it oversees. The node first checks if the JAUS IP in the message matches its own JAUS IP. If not, it compares the first two numeric values of its JAUS IP to the first two numeric values of the JAUS IP the message it is intended for. If they match, the message is intended for a component which the node oversees. The node uses the destination JAUS IP from the message as a hash key in the hash map for the components to get the IP address and port that it should forward the message to. If the values do not match, the node must pass the message to its subsystem; it uses the first numeric value of its own JAUS IP as the hash key in the hash map for the subsystems. By using hash maps, routing within JAUS systems can be efficiently handled.

At each point along the path a JAUS message takes from its origin to its destination, there are a few validation steps that all applications take when a message is received. The message is checked to determine that its size is at least the size of the header, 16 bytes, the JAUS version byte is checked to determine that it is the correct JAUS version and the length of the data portion of the JAUS message is compared to the message size field in the header to ensure that they match. For our simple JAUS implementation, this is sufficient validation.

When a message is received by the intended application instance, the message must be processed. Each message type has a handler class which contains the code to process the message. The JAUS standard, as of Version 3.3, has one hundred fifty-seven messages [21] and performing as many if else statements to determine the message type could negatively impact performance. Hash maps are used to make the process of resolving an incoming message to the class which should process it more efficiently. In the command core field of the JAUS header is a numerical value which indicates the type of message. This integer value is used as a key to a hash map which resolves to a pointer to the static function which processes the message. The number of message types is static, which means that during the execution of the program no values are added or removed from the hash map, which means the performance of the hash map will not degrade. This allows the application to determine, in constant time, the proper function to process each message type.

The "FunctionHasher" resolves to a specific function in a handler class for each type of message. When a handler class is processing a message, it may need access to external variables, classes, and so forth. For instance, if a message is received which must alter the speed of the engine in a UV, the handler class must have access to the API for engine control. The "HandlerBundle" class deals with this problem. This class is passed to all message handling classes which can use any classes, variables, or data included in "HandlerBundle." This way, developers can avoid altering the core functionality of the JAUS implementation. All that needs to be done is to include the necessary elements in the "HandlerBundle" class and then code each message handling class to process messages in the appropriate manner.

6.2. JAUS Experiments. We conducted eight different experiments to assess the performance of the JAUS standard when using the UDP protocol. All of the experiments test the time required to construct, send, receive, and process a JAUS message in order to determine worst-case response-time results. We used a single message type for all of our tests, the "Set Component Authority" JAUS message. The message has the standard 16-byte JAUS header and the payload is a single byte. The reason for selecting this message type is that it is similar to a CAN message in size. The worst-case response-time analysis for CAN generally uses the maximum message size of 132 bits. The "Set Component Authority" message is 17 bytes, or 136 bits, a trivial difference from the CAN message. This allows us to rule out message size as a significant

source of latency in the JAUS system. The tests are listed below.

(1) Test the time required to send a JAUS message from a component in one subsystem to a component in another subsystem.

(2) Test the time required to send a JAUS message from a node in one subsystem to a node in another subsystem.

(3) Test the time required to send a JAUS message from a subsystem to another subsystem.

(4) Test the time required to send a JAUS message from a component to another where both components are in the same subsystem and share the same node.

(5) Test the time required to send a JAUS message from a component to another where both components are in the same subsystem but have different nodes.

(6) Test the time required to send a JAUS message from a component to its node.

(7) Test the time required to send a JAUS message from a node to its subsystem.

(8) Test the time required to send a JAUS message from a component to its subsystem.

For all of the experiments, a subsystem and its nodes and components are all on the same computer. The first three experiments require the use of two different computers since the message passes between different subsystems at some point during each of these experiments. This allows us to examine how performance is affected by messages which must pass through a router to reach their final destination in addition to the effect that longer paths through the JAUS hierarchy have. For the last five experiments, only a single computer is needed since all of these experiments take place within a single subsystem. Tests 4 and 5 are complete JAUS transactions in full compliance with the JAUS standard which allows us to determine how shorter and longer paths through the JAUS hierarchy perform. Tests 6 and 7 allow us to gauge the speed of the UDP protocol since they both test direct communication between elements in a JAUS system. Test 8 gives us the time necessary for a message to get to a component once it has entered a UV through the subsystem.

In the experiments, a subsystem and its nodes and components are all kept on the same computer for a more favorable measure of time. Current research has indicated that the performance of JAUS is too slow when using the UDP protocol and we are operating under that assumption [2]. If the elements in the subsystems are on separate computers, the transmission time will be longer since the message will have to travel through wires and routers at each point to reach its destination. Transmitting messages between elements which are on the same computer requires less time since the messages stay on the same motherboard. This provides a best-case performance for the experiments and for the performance of JAUS. A favorable outcome of our experiments when compared with the CAN results indicates further research is necessary to pinpoint the source of latency in JAUS. If, even under favorable conditions, the performance of JAUS cannot compete with CAN, we have found the source of latency.

For each experiment, we would like to determine the time required for one subsystem, node, or component in a JAUS system to reach another subsystem, node, or component elsewhere in the same JAUS system. In the first three experiments, the sending and receiving elements are on different computers, so getting the start time on the first computer and the finish time on the second computer may not provide accurate results due to the clocks in the computers not being synchronized. To overcome this, we determine the round trip time and halve it.

Each experiment will test the transmission time of 1000 messages so that we have an adequate sample from which a reasonable transfer time can be inferred. A "for" loop, counting from 0 to 999, inclusive, is used to send and track the tests. The sending subsystem, node, or component, which we will refer to as the Initiator, will get the current time and save the value in an array using the value from the "for" loop as the index to which the value is saved. Then, the JAUS message is constructed with the sequence number field containing the index value from the "for" loop. The message is then sent and routed through the JAUS hierarchy, in compliance with the standard, to its destination, which we will refer to as the Receiver. The Receiver processes the message and retrieves the index value from the sequence number field. It then constructs its own JAUS message and sets the sequence number in the message to the value of the sequence number in the message which it received from the "Initiator." The "Receiver" then sends its message back to the "Initiator." When the "Initiator" receives this message, it processes it and retrieves the sequence number. The current time is then saved in the array in the index corresponding to the sequence number from the message. This process occurs 1000 times, with a 100 ms pause between each message sent by the "Initiator."

From the data we glean four statistics: the minimum transmission time, the maximum transmission time, the average transmission time, and the standard deviation. The difference in the start and finish time is calculated for each message and divided by two to get a reasonable estimate of the transmission time of a one-way trip for the JAUS messages. UDP is a connectionless protocol so packets may be lost during transmission. The array containing the time values is zeroed out prior to each message being sent. When calculating the statistics, the application checks if the return time is 0, which indicates that a packet was lost during the test. The number of tests used as the divisor in both the average and standard deviation calculations is accordingly adjusted to not include lost packets. Tests four through eight are all only on one machine, which means we could run the tests by simply taking the start time at the "Initiator" and the finish time at the "Receiver." We opted to perform all of the tests in the same manner to maintain consistency.

The time measurement tool which is used to measure the elapsed time is the "QueryPerformanceCounter" tool provided by Microsoft in Visual Studio 2008. The "QueryPerformanceCounter" provides time data in the form of clock

cycles per second since the computer was started. To obtain the time in seconds, the number of clock cycles the computer performs each second is divided into the time value.

For our experiments, we used two identical computers running the Windows XP Service Pack 3 operating system. The chips are the Intel 2 Quad Core running at 2.40 GHz and the ram size is 3.0 GB.

6.3. Results and Analysis. Our analysis highlights a few different problems causing performance issues with JAUS. The hierarchical message passing scheme required by JAUS is one source of latency. Directing messages through subsystems and nodes, instead of directly between sending and receiving components, introduces latency into the system. Another issue is the use of the UDP protocol. In addition to being slower than CAN systems with fast bus speeds, UDP provides no scheduling mechanism. The results of our experiments are provided in Table 2 and a graph in Figure 11.

6.3.1. JAUS Hierarchy. We first examine how the hierarchical routing scheme of JAUS affects performance. In Figure 11, the results of each of the eight tests are plotted. In addition to the mean, the standard deviations above and below the mean are also plotted to give a sense of the variation in transmission times. As hypothesized, the more the subsystems, nodes, and components which a message must traverse to reach its destination are, the greater the time is.

Transmitting from component to component in Test 1 requires the greatest time, transmitting from node to node takes less time in Test 2, and transmitting directly between subsystems in Test 3 takes the least amount of time of these three tests. Tests 4 and 5 test transmission times between two components in the same subsystem. In Test 4, both components belong to the same node while in Test 5 the components belong to different nodes. The message in Test 4 only needs to pass from the sending component to the node before reaching its destination at the receiving component. Test 5, however, requires the message to pass from the component all the way to the subsystem before it can be sent to the receiving component by way of its managing node, and the difference in performance is noticeable. Test 5 performs more poorly than Test 1 even though Test 1 must go through a router to bridge the gap between the two machines.

Tests 6, 7, and 8 test the necessary time to transmit a JAUS message between a component and node, a node and subsystem, and component and subsystem, respectively. Tests 6 and 7 are very similar since they both measure the time required for direct communication between elements in the JAUS hierarchy. Test 8 requires an additional hop due to the node and as a result underperforms in comparison to Tests 6 and 7.

The indirect message routing scheme of JAUS has a noticeable impact on performance as expected. We now use this data in the context of Table 1. The results in Table 2 detail the time required to send a minimal JAUS message for each of our tests. In Table 1, the first nine messages have a deadline of 5.0 ms. Using the average response time from Table 2, we see that in Tests 1, 2, 5, and 8 the transmission speed is too slow;

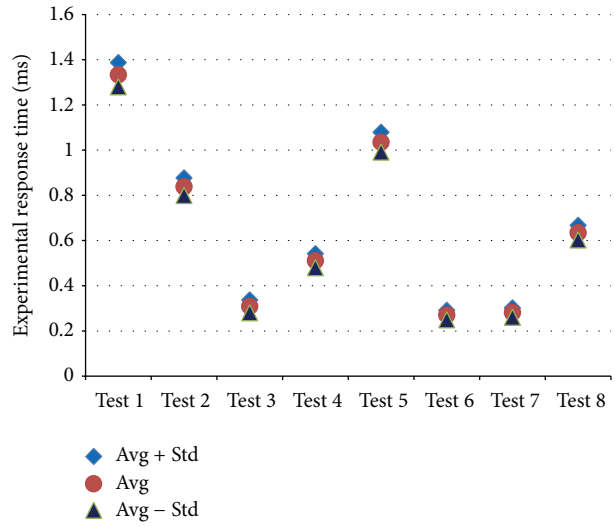


Figure 11: JAUS experiment test results.

Table 2: Experiment results (the time is in the units of ms).

Test number	Minimum response time	Maximum response time	Average response time	Standard deviation
1	0.417706	1.643011	1.333256	0.053709
2	0.417706	1.099552	0.837777	0.038997
3	0.275044	1.045092	0.308805	0.028363
4	0.417706	0.656319	0.510614	0.031094
5	0.417706	1.303495	1.035215	0.043867
6	0.227896	0.392914	0.270172	0.021158
7	0.234931	0.381488	0.281037	0.020506
8	0.417706	0.819728	0.634676	0.032273

if nine JAUS messages were sent consecutively, each with a deadline of 5 ms, not all of the messages would make their deadlines.

The configurations that are too slow have a significant amount of overhead, traveling through many nodes and subsystems. The solution, then, is to eliminate nodes and have all internal communication happening directly between nodes. This, however, is not enough, which will be demonstrated in the next section.

6.3.2. JAUS/UDP. To compare JAUS/UDP with CAN, we will ignore the JAUS hierarchy and use the results from Test 6, which tests the communication between a component and its node, a direct connection, which is also the smallest value from our results.

In Table 1, the first message is the highest priority message and its response time is representative of the response time in the CAN network of a message that is one byte in size. Since all of the messages in Table 1 are one byte, we can use the response time for the first message as a reasonable indicator of the speed of 1-byte messages in a CAN system. If we compare the response times for the first message in

250 Kbit/s, 500 Kbit/s, and 1 Mbit/s CAN systems with the response time for Test 6, we see that the JAUS message is actually faster than the 250 Kbit/s and 500 Kbit/s CAN systems. The problem is that raw speed is not the only problem with JAUS.

Consider the scenario in Table 1 again and use the average response time from our results in Test 6. The first nine messages have deadlines of 5 ms. The time to transmit a JAUS message from Test 6 is 0.270172 ms, so the time required to send the first nine messages is 2.431548 ms.

The problem that arises is a scheduling problem. In a scenario where all of the types of messages are being sent simultaneously, collisions will occur. As discussed previously, CAN is able to send the messages in order of priority even when collisions occur. This is not the case with UDP. If a collision occurs, all transmitting nodes stop, wait a random time period, and then attempt to retransmit. This makes it virtually impossible to know the order in which the messages will be transmitted. It is then possible that any of the ninth highest priority messages will not be one of the first nine messages sent. Any of the messages with deadlines of 5 ms must be one of the first 18 messages sent or they will miss their deadline. Messages with deadlines of 10 ms must be one of the first thirty-seven messages sent or they will miss their deadlines. Using JAUS and UDP it is possible that using JAUS and UDP all messages in the scenario in Table 1 will meet their deadlines. It is also possible that they will not make their deadlines and there is no way to guarantee which situation will occur. Obviously this level of uncertainty is not acceptable in mission-critical systems and thus JAUS/UDP cannot be used for internal controls of UVs.

7. JAUS/UDP to EtherCAT Bridge: JEBridge

To improve the performance of JAUS/UDP communication discussed in Section 6, we utilize EtherCAT as a solution, due to the fact that EtherCAT is an emerging, but promising, lower level protocol that has a lot of support. There is a general way to translate between JAUS and EtherCAT. A survey of available EtherCAT modules shows this to be impossible. EtherCAT modules have widely different device profiles. So the top down method, starting with JAUS and translating to EtherCAT, will not work. However, a bottom-up method, starting with a number of EtherCAT modules and deriving the translation, should be quite achievable.

The first question to ask when bringing EtherCAT into JAUS is where we should place the EtherCAT. In other words, at what level should we need to implement the EtherCAT? As mentioned before, motion control requires deterministic timing with bounded latency. So our approach is to implement EtherCAT at Level 3, where the EtherCAT bridge is inserted into components, and EtherCAT communication happens between nodes/subsystems. Figure 12 shows the insertion of EtherCAT into existing JAUS communication system.

In the following subsections, we will discuss the implementation of our JAUS-EtherCAT bridge.

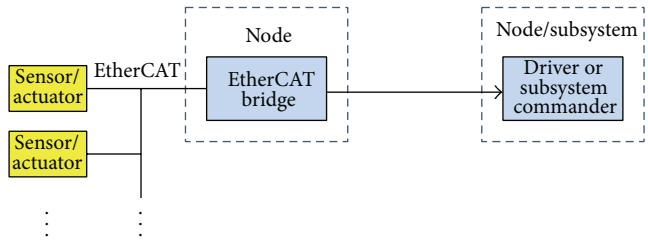


Figure 12: The next stage in the evolution to EtherCAT based sensors and actuators (Figure 3-4 in [19]).

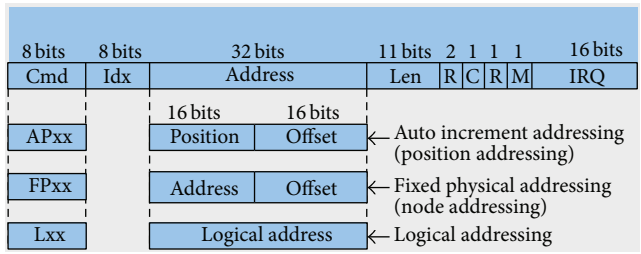


Figure 13: Address Field (slide 34 of [7]).

7.1. EtherCAT Addressing Schemes. EtherCAT modules fall into two categories: "native EtherCAT" and "transition modules" which use EtherCAT to encapsulate a legacy protocol, such as CANopen.

7.1.1. Native EtherCAT Module. We first consider native EtherCAT module. The important parts of the EtherCAT telegram header are the command (8 bits), the address (32 bits), and the telegram data size (11 bits). The EtherCAT command defines the way the address is evaluated by the EtherCAT slave devices. It may either be interpreted as a physical address (16 bits) with an offset (16 bits) within the address space of the EtherCAT Slave Controller (ESC) or as a logical address (32 bits) of a 4 GB virtual address space, and this is shown in Figure 13.

JAUS addressing consists of four fields: Subsystem, Node, Component, and Instance. Each field consists of 8 bits. Thus, the JAUS address is 32 bits total, the same size as the EtherCAT address. A higher level system engineer can easily adopt JAUS convention and philosophy as she assigns addresses to EtherCAT modules. The native EtherCAT-to-JAUS bridge is shown in Figure 14. Note that both JAUS and EtherCAT favor little endian integer representation. So in the above diagram Subsystem is the most significant byte, and Instance is the least significant byte, as convention would suggest.

7.1.2. Transition Module. Next we consider "transition module," in particular CANopen. CANopen device has to implement certain standard features in its controlling software: application (top layer), object dictionary (middle layer), and communication (bottom layer) shown in Figure 15.

The application part of the device actually performs the desired function of the device, after the state machine is set to the operational state. The application is configured by

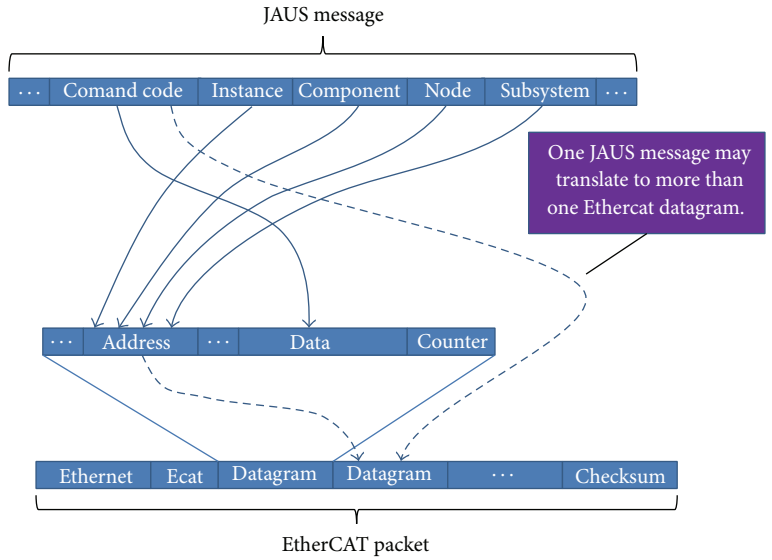


Figure 14: Address and data translation from JAUS to native EtherCAT.

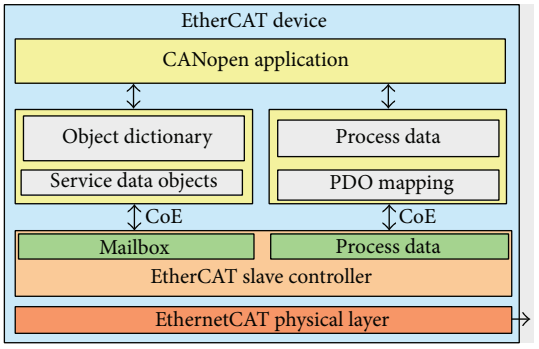


Figure 15: CANopen over EtherCAT device architecture (slide 128 of [7]).

variables in the object dictionary and the data are sent and received through the communication layer.

The object dictionary is an array of variables with a 16-bit index. Additionally, each variable can have an 8-bit subindex. The variables can be used to configure the device and reflect its environment, that is, containing measurement data.

The object dictionary is associated with Service Data Objects (SDO) protocol. The SDO protocol is used to set and read values from the object dictionary of a remote device. The device whose object dictionary is accessed is the SDO server and the device accessing the remote device is the SDO client. The communication is always initiated by the SDO client. In CANopen terminology, communication is viewed from the SDO server, so that a read from an object dictionary results in an SDO upload and a write to dictionary is an SDO download.

A communication unit implements the protocols for messaging with the other nodes in the network. Starting and resetting the device are controlled via a state machine. It must contain the states Initialization, Preoperational, Operational, and Stopped. The transitions between states are made by issuing a network management (NMT) communication object to the device.

In transition modules, the addressing scheme is the same, except that the address space for CAN is smaller; thus, some fields in the JAUS addressing scheme are unused. The transition modules to JAUS is shown in Figure 16.

7.2. Implementation of Native EtherCAT Module. We start the implement of the native EtherCAT module by defining the Ethernet and EtherCAT frame structure (refer to Figure 2 in Section 3) in a header file and then develop a function that initializes Ethernet header and EtherCAT frame header Type 1 (EtherCAT Device Communication). Ethernet header is initialized with values of Destination, Source, and Type.

We now want to send a Logical Write (LWR) packet. We first need to receive the JAUS message which is converted into a generic JAUS message. We then initialize the Ethernet header and the EtherCAT frame header. The EtherCAT datagram header is initialized with "Cmd" equaling LWR. Now, we want to implement the mapping shown in Figure 14. The JAUS destination addresses are packed and stored into the "Address" field of the EtherCAT. As mentioned earlier, each field of those JAUS "Destination ID" is 8 bits. Thus, the JAUS address is 32 bits total, the same size as the

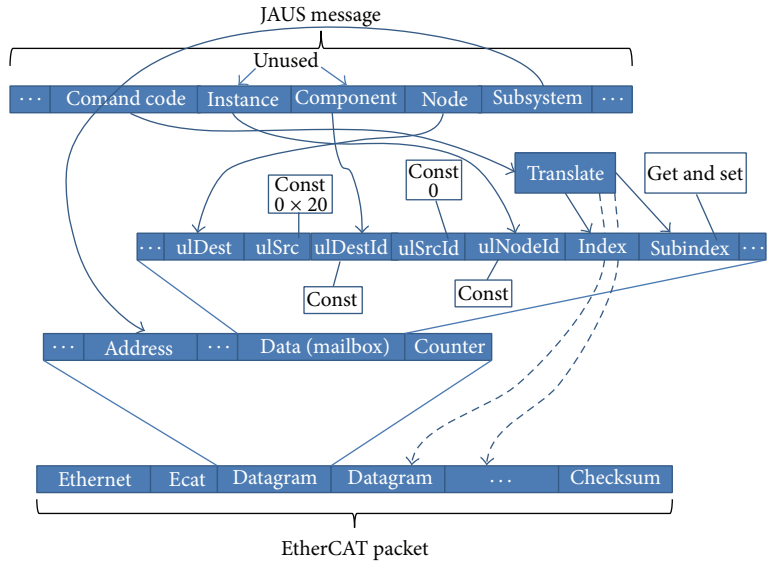


Figure 16: Address and data translation from JAUS to CoE (CAN over EtherCAT).

EtherCAT address. The packet data, which in this case an array of unsigned char, is then filled out with JAUS "SetWrenchEffort" data (as an example). The total packet length is then calculated.

The "CommandCode" of JAUS could translate into multiple EtherCAT datagram, but this is not implemented in our code. Here is the explanation for that specific mapping. Our example was for component which implemented all three axes. It is conceivable that we have three components each of which controls an axis, and in that case the single JAUS message that would be translated into three EtherCAT datagrams; and the resulting EtherCAT data from each axis would be put back into a single JAUS single message.

7.3. Implementation of Transition EtherCAT Module. We start the implement of the transition module by structure shown in Figure 17. The detailed definition and implementation of this frame is shown in Algorithm 2.

Now we are ready to send a CANopen packet named "SDO_Services_Packet" over EtherCAT. Our mapping described below is according to Figure 16.

(1) The first three mappings are mapping based on JAUS-CAN hierarchical structure. JAUS hierarchical structure is clearly defined in JAUS RA [19–21]. CAN structure is not defined in any CAN document. But we can claim that CAN hierarchy consists of Dest, DestId, and NodeId. Our justification for this claim is based upon CANopen Master Protocol [24]. Thus, JAUS Node, Component, and Instance are naturally mapped into CAN Dest, DestId, and NodeId, respectively. Note that JAUS Subsystem is not mapped into anything, because CAN already knows which Subsystem the CAN device is for.

(2) "ulSrc" of EtherCAT is filled with a constant value of "0x20" being consistent with [24].

(3) "ulSrcId" is filled with a constant value of "0" being consistent with [24].

(4) "ulIndex" takes the highest 8 bits of the JAUS "CommandCode."

(5) "ulSubIndex" takes the lowest 8 bits of the JAUS "CommandCode." In "ulSubIndex," there are two possible values: get and set (0 and 1).

The "Index" of CAN could translate into multiple EtherCAT datagram, but this is not implemented in our code. Here is the explanation for that specific mapping. Our example was for component which implemented all three axes. It is conceivable that we have three components each of which controls an axis, and in that case the single JAUS message that would be translated into three EtherCAT datagram; and the resulting EtherCAT data from each axis would be put back into a single JAUS single message. Codes for the implementation of our mapping are skipped here but we would be happy to share upon your request.

7.4. Quantitative Analysis: Performance. JAUS messages are short, and short messages over an Ethernet channel are very inefficient [25], as shown in Figure 18.

7.5. Qualitative Analysis: Nondeterministic versus Deterministic Control. JAUS is designed to be independent of communication protocol; however, the overwhelming protocol choice appears to be JAUS over UDP. Thus, our observations apply to this case. UDP has both advantages and disadvantages. The advantages include commodity hardware and quick prototyping (plug and play). The disadvantages include nondeterministic timing. UDP is a peer-to-peer, CDMA (Collision Detect Multiple Access) protocol. When a collision occurs, the packet is lost. EtherCAT is a master-slave protocol, so collisions should not occur and timing is

```
struct CAN_Packet_Header
{
    UINT32        ulDest;
    UINT32        ulSrc;
    UINT32        ulDestId;
    UINT32        ulSrcId;
    UINT32        ulLen;
    UINT32        ulId;
    UINT32        ulSta;
    UINT32        ulCmd;
    UINT32        ulExt;
    UINT32        ulRout;
};

struct CAN_Packet_Body
{
    //MbxHeader
    UINT16        Length
    UINT16        Address;
    UINT8     Channel_Prio;
    UINT8         Type_Cntr;
    //CoE_Cmd
    UINT16        Num_Res_Type;

    //CoE_Specific_Data
    UINT8         ulNodeId;
    UINT16        ulIndex;
    UINT8         ulSubIndex;
    UINT32        ulDataCnt;
};

struct CAN_Packet_Data
{
    UINT8        abSdoData[100];
};
struct CAN_Packet
{
    CAN_Packet_Header          can_pkt_header;
    CAN_Packet_Body            can_pkt_body;
    CAN_Packet_Data            can_pkt_data;
};

struct SDO_Services_Packet
{
    EthernetHeader              ethernet_header;
    EthercatFrameHeader         ethercat_frame_header;
    EthercatDatagramHeader
    ethercat_datagram_header;
    CAN_Packet                  can_pkt;
    WorkingCounter              wkc;
};
```

Algorithm 2: CoE frame structure (header file: ecat_packet.h).

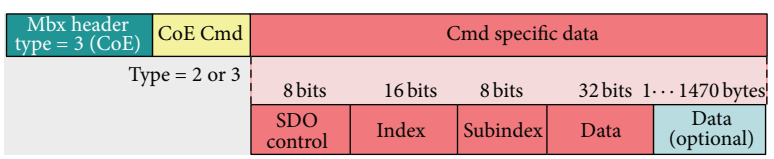


Figure 17: CANopen over EtherCAT devicearchitecture (slide 131 of [7]).

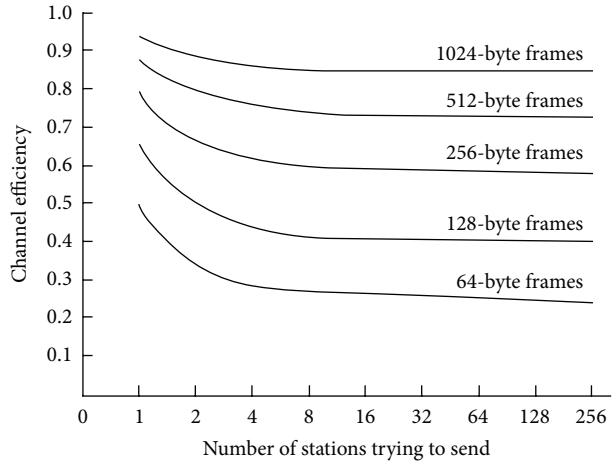


Figure 18: Channel efficiency over the total packet size with TCP overhead 20 bytes [25].

deterministic. Application software must detect this and retry the transmission.

UDP/Ethernet and EtherCAT share the same physical layer. Cabling and switching infrastructure are identical. Acceptable, though still nondeterministic, performance over Ethernet requires bandwidth utilization of under about 30%. By contrast, EtherCAT can utilize over 90% of the media bandwidth.

A JAUS Subsystem can be large and complex, and complexity can increase over time as things are added. Things could be added without the oversight of a system engineer.

Ethernet is susceptible to a DOS (Denial of Service) attack. Nodes are computers; they have operating systems which could be infected by malware, or an infected computer could be plugged into the network. Flooding a network with traffic will effectively shut it down. This is particularly easy to do with UDP, since, at the protocol level, UDP has no congestion control (unlike TCP). An EtherCAT network should be more robust, since devices enforce congestion control, and there would be no temptation to plug a computer into the network.

8. Conclusions

In this research we have created a basic JAUS implementation in order to evaluate the performance of the standard at its core. Even in its simplest form, JAUS is unable to perform at the level of CAN for a few different reasons. The hierarchical structure through which messages must be passed contributes significantly to latency issues. Additionally, the UDP protocol provides no means for prioritizing messages, making it virtually impossible to make reasonable guarantees of message delivery times, and is simply slower than CAN with higher bus speeds. Consequently, JAUS/UDP is not a suitable communication standard for internal control systems of UVs.

Although JAUS is not suitable for communication in internal, real-time systems in UVs, it can still be used for the purpose of interoperability. By using EtherCAT for internal controls and JAUS as the subsystem level protocol, both interoperability and performance can be achieved, as we presented in the paper. Manufacturers are still able to use their existing internal systems and need only provide a way to interface, or translate between, JAUS and their proprietary methods.

Regarding our JEBridge solution, we note that closed loop control systems require deterministic time, which JAUS does not guarantee. Thus, we do not recommend Level 3 JAUS Compliance and recommend using EtherCAT in its place. EtherCAT does not compromise the goal of "reusable components" [19–21], because of the range of available EtherCAT modules [26]. In addition, there are advantages to forgoing JAUS Level 2 Compliance and using EtherCAT even when critical timing is not a requirement, since EtherCAT provides a robust backbone communication mechanism. Of course, we remain solid supporters of JAUS Level 1 Compliance.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was supported by Dual Use Projects of MKE (Ministry of Knowledge and Economy) contract titled "Development of Quadruped Legged Robot Platform Technology." Although JAUS was originally an initiative starting in 1998 by the United States Department of Defense to develop an open architecture for the domain of unmanned systems, it is now an open international standard that is supervised by SAE International and the research group planned to develop a robot architecture based upon it. An earlier version of this paper was approved by ADD Dual Use Technology Center for Public Release. The authors also would like to thank anonymous reviewers for their valuable comments and suggestions which lead to substantial improvements of their paper.

References

[1] United States Senate, "Fiscal Year 2001 Defense Authorization Bill," Sec 217.

[2] SAE, "Class C application requirement considerations-SAE J2056/1," in *SAE Handbook*, pp. 23.366–23.371, 1993.

[3] S. Rowe and C. Wagner, "An Introduction to the Joint Architecture for Unmanned Systems (JAUS), Open Skies," 2008, http://www.openskies.net/papers/07F-SIW-089%20Introduction%20to%20JAUS.pdf.

[4] S. J. Lee, D. M. Lee, and J. C. Lee, "Development of communication framework for unmanned ground vehicle," in *Proceedngs of the International Conference on Control, Automation and Systems (ICCAS '08)*, pp. 604–607, Seoul, Republic of Korea, October 2008.

[5] D. Barber, S. Leontyev, B. Sun, L. Davis, D. Nicholson, and J. Y. C. Chen, "The mixed-initiative experimental testbed for collaborative human robot interactions," in *Proceedings of the International Symposium on Collaborative Technologies and Systems (CTS '08)*, pp. 483–489, Irvine, Calif, USA, May 2008.

[6] M. Sugiura, K. Kobayashi, K. Watanabe, and T. Ohkubo, "Development of unmanned ground vehicle for IGVC JAUS challenge," in *Proceedings of the SICE Annual Conference*, pp. 2719–2722, Tokyo, Japan, August 2008.

[7] EtherCAT Technology Group, "EtherCAT Communication," 2008, http://lectoraatmechatronica.wikispaces.com/file/view/EtherCAT%20communication.pdf/346392478/EtherCAT%20communication.pdf.

[8] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1204–1223, 2005.

[9] K. Tindell and A. Burns, "Guaranteeing message latencies on Controller Area Network (CAN)," in *Proceedings of the 1st International CAN Conference*, pp. 1–11, Mainz, Germany, 1994.

[10] H. Woo, M. Kim, and J. Kim, "Development of multiple communication using JAUS message set for unmanned system," in *Proceedings of the International Conference on Control, Automation and Systems (ICCAS '07)*, pp. 2374–2377, Seoul, Republic of Korea, October 2007.

[11] N. C. Audsley, A. Burns, M. F. Richardson, and K. Tindell, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–2292, 1993.

[12] J. Albus, H. Huang, E. R. Messina et al., "4D/RCS: a reference model architecture for unmanned vehicle systems version 2.0," 2002, http://www.robotictechnologyinc.com/images/upload/file/4DRCS.pdf.

[13] "EtherCAT Master, Cross Platform Stack," 2011, http://www.esd-electronics-usa.com/Shared/Handbooks/EtherCATMasterDevelopersManual.pdf.

[14] CAN in Automation (CiA), http://www.can-cia.org/.

[15] S. Talbot and S. Ren, "Comparison of FieldBus systems, CAN, TTCAN, FlexRay and LIN in passenger vehicles," in *Proceedings of the 29th IEEE Conference on Distributed Computing Systems Workshops*, pp. 26–31, Montreal, Canada, 2009.

[16] H. Chen and J. Tian, "Research on the controller area network," in *Proceedings of the International Conference on Networking and Digital Society (ICNDS '09)*, pp. 251–254, Guizhou, China, May 2009.

[17] T. Nolte, H. Hansson, and C. Norstrom, "Probabilistic worst-case response-time analysis for the controller area network," in *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 200–2207, 2003.

[18] R. Davis, A. Burns, R. Bril, and J. Lukkien, "Controller Area Network (CAN) schedulability analysis: refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[19] The JAUS, "Reference Architecture Specification," Volume II, Part 1, *Architecture Framework*, Version 3.3, 2007.

[20] The JAUS, "Reference Architecture Specification," Volume II, Part 3, *Message Set*, Version 3.3, 2007.

[21] The JAUS, "Reference Architecture Specification," Volume II, Part 2, *Message Definition*, Version 3.3, 2007.

[22] J. Pedersen, "A practical view and future look at JAUS, RE2," White Paper, May 2006, http://www.thesciencedude.com/projects/RESEARCH/MASSystem/References/JAUSwhitepaper.pdf.

[23] "TinyXML," http://www.grinninglizard.com/tinyxml/.

[24] "CANopen Master, Protocol API," 2009, http://www.hilscher.com/files_manuals/CANopen%20Master%20Protocol%20API.pdf.

[25] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, Pearson, 5th edition, 2010.

[26] EtherCAT Technology Group, "EtherCAT products," http://www.ethercat.org/en/products.html.