2020 16th IEEE International Conference on Automation Science and Engineering (CASE)
August 20-21, 2020, Online Zoom Meeting

ROSFuse: A High-modularity ROS to Firmware Instrumentation Bridge for Robotic Sensors

Christopher Robinson, *Student Member IEEE*, Shamsudeen Abubakar *Member IEEE*,
Sumit K. Das *Member IEEE*, and Dan O. Popa, *Senior Member IEEE**

***Abstract*—In this paper we present a modular software protocol for extending a variable dataspace within a microcontroller firmware system (MCU) that allows robotic sensor data to be streamed via the Robot Operating System (ROS) architecture. This protocol copies the data formatting structure inherent to ROS messages and implements a local DN bridge to allow for asynchronous bi-directional data transport over any communication channel. We implement a demonstration of this system on a mobile robot test bed to manage communications between the sensor data acquisition MCU and the primary control computer, and use this test case to measure the efficacy of the protocol through latency and packet loss, and tracking validation by comparison to other measurement systems on the robot.**

***Index Terms*—data protocol, assistive robot, sensing protocol, sensor suite**

I. Introduction

In the course of development of large robotic systems, direct access to lower level hardware assets can rapidly become problematic [1]. Off-the shelf control hardware is rarely equipped to communicate with pin-level hardware [2]. Typically, CPU systems expect interaction to occur above the Hardware Abstraction Level (such as Ethernet, USB, Serial, and other related common communication protocols). Such interfaces can often be adapted to low-level sensors and actuators. Access to these assets on a CPU is typically in short supply compared to the span of instrumentation implemented on a robot [3]. For instance, one rarely finds more than 6-10 USB ports on a CPU, but this does not easily support sensor numbers in excess of a dozen.

This issue could be approached by increasing port availability- by using USB hubs or Ethernet switches for instance- but this introduces problems as well. The reliability of port expansion, management of network systems, and the added bulk, complexity, and resource consumption are all potentially intractable on certain projects. Beyond these pragmatic concerns, there is also the effect on efficiency; complexity introduced from multiple communication hubs, added latency, and potentially significant computational overhead all limit performance.

The natural and common response to these challenges is to implement low-level tasks with an appropriately low-level computation system and pass the resultant data to an interface for the CPU. This is where a CPU/MCU based system becomes most attractive, and where an efficient, scalable protocol for sharing data is essential.

Robots have utilized this approach for decades, and there is a vast body of work associated with the equally vast number of options, which are most commonly specific to the application case. This is generally a matter of both expedience and pragmatism- a purpose-built design is typically the fastest and most reliable method of implementing communication between hardware systems [4]. However, the principle design issue these kinds of systems present is one of scalability [5]. When a system is designed optimally for a single use case, it is often the work of a complete re-design to expand it to accommodate changes.

In this paper, we are proposing a lightweight firmware-to-software bridge protocol which handles data transport between MCUs and CPUs to address these issues. In particular, due to the ubiquity of ROS as a fundamental design tool for robotic software, we designed our system to integrate with, and mirror the structure of, ROS. In constructing our protocol this way, we garner many of the benefits of the ROS architecture across an embedded system interface.

A. Prior Work

One method of achieving the same goal of interfacing compliance is via a hardware-based interface device. A widely adopted example of this is the National Instruments roboRIO. The roboRIO is a dedicated hardware platform carrying many embedded protocols and associated transport software. One such similar application to our demonstrator for ROSFuse is found in [6]. While highly efficient, this example also highlights common weaknesses of these hardware-focused interfaces. Though the roboRIO contains a wide range of interfaces, custom hardware is still required to bridge the gap between an atypical low-level sensor and the roboRIO. Other concerns include the cost-to-benefit ratio. With generalized hardware interfaces, there is often far more systematic and cost overhead present than is appropriate for an application case. When a dedicated microcontroller and a small amount of support hardware is more fitting, solutions like the roboRIO exhibit a low return on investment.

Alternative models have also been proposed which seek to remedy this problem, most notably architectures to manage communications across many devices. Below, we discuss several of these and discuss how our approach compares.

In [7], H-ROS, an architecture was developed towards achieving a ROS-compatible hardware standard. The archi-

* The authors are with Next Generation Systems Research group, Department of Electrical and Computer Engineering, University of Louisville, Kentucky, USA, {christopher.robinson.2@louisville.edu}

tecture builds on ROS and has software features that facilitate access to the robot hardware and compartmentalization of software in different modes. Interoperability, reusability and reconfigurability are presented as the main benefits. However, because it is targeted for industrial use, it is built with features that limit wider adoption. One limiting feature is the Ethercat Protocol used in the physical layer, which presents a whole level of hardware integration beyond the base system being used.

A driver to interface Arduino-based robots with ROS is presented in [8]. Simulation and real-world navigation tasks are used to show functionality of the driver. While demonstrated to work well with different Arduino microcontrollers, scalability is shown only by adding only three single-value-output sensors to its test robot. Further, the authors indicate that significant programming is required to integrate these sensors to ROS via the driver, as is typical of such bespoke systems.

By way of contrast, we also consider [9], in which integration to ROS of hardware systems is approached with a hardware-level solution, implemented as a unified system firmware on an FPGA. While this provides exceptional performance, the usage of single system chips is highly limiting with regards to modularity. Like with the system discussed in [6], efficiency is very high, at the expense of the overhead of programming an FPGA system.

A similar approach to our concept has also been applied to the specific topic of sensor fusion in [10]. Herein the authors leverage the natural modular nature of ROS nodes to produce a generalised sensor fusion package which can be interfaced with other ROS ecosystem members. As with our methodology, the use of systematic design- relying on the publisher/subscriber paradigm- allows for the generation of scalable systems through ROS.

B. Contributions

Our protocol was designed to possess following advantages over other frameworks:

- Scalability: Our protocol demonstrates strict linear relations between packet length and delay, and provides a mechanism for adding devices by editing only a configuration file.

- Simplicity: our structure is designed specifically to copy the ROS ethos of configuration, allowing changes to variables to be made entirely independent of the source code

- Efficiency: we have made use of the ROS publisher/subscriber model for datasharing through message types. This allows us to implement variable sharing with minimal latency.

We illustrate the advantages of ROSFuse in an application sensorizing a mobile manipulator robot in our lab. The robot included 32 sensors, whose data is streamed to a central CPU during environmental mapping and navigation. Results show that sensorization using the ROSFuse system enables high-speed transmission with a large number of sensors.

The paper is organized as follows: In section II we describe the ROSFuse protocol in detail; in Section III we outline the systems used in the validation task; in section IV we analyze data collected from the robot, and identify specific systematic performance metrics; Finally, in Section V we present our conclusions and discuss future work.

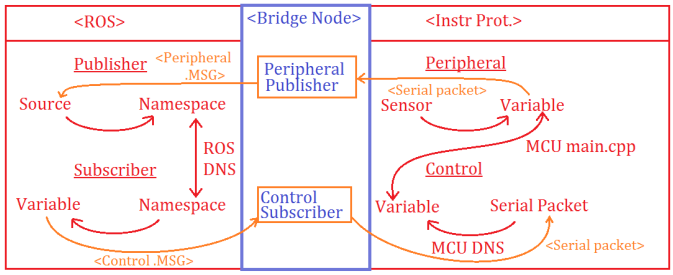

Fig. 1. Firmware to ROS bridge process workflow

II. Protocol Description

In this section, we detail the design of the ROSFuse communication protocol. ROSFuse is structured to topically mimic ROS's publisher/subscriber framework, including the use of message definition files to define data types. The primary goal of the data share bridge is to make available in the ROS namespace variables which are set within the MCU, and vice-versa.

At the top level, this is achieved by a pair of processes: a ROS node running on the CPU, and an interrupt-driven process on the MCU. The ROS node retains a list of the topics initialized for data sharing, and the MCU process transfers data to and from variables internally. The ROS node subscribes to topics shared with the MCU and publishes topics shared from it, while the firmware interrupt routine parses data into variables, and transmits shared variables. This full workflow is illustrated in Figure 1, and the procedure for message parsing in Algorithm 1.

A. Packet Structure

In order to implement any communication protocol, we must first select a packet type for the transmission itself. We select a simple string-based, delimited structure, as indicated in Figure 2.

The advantages of using this format are three-fold:

I. Use of a string format simplifies parsing- though a numerical formatting system would enable for lower-overhead communication, the use of message namespace conventions when coupling to ROS dramatically simplifies the process of integration.

II. Messages formatted with start and end characters via the native string type can be of functionally unbounded length

III. The use of the String type obviates the need for a complex encoding protocol for carrying strings- because packets are patched directly into a ROS topic, matching packet names to topics is both convenient for readability and identification.

As a further note, because the use of message type descriptors is copied from ROS to the firmware, there need not be any data included within the messages themselves to guide

Algorithm 1

procedure ROSFUSE_PARSE((Controls,Peripherals,Port))
  if Port.available() **then**
    while $Port.next() \neq$ "&" **do** Port.read()
    $packet \leftarrow$ ""
    while $packet[-1] \neq EOL$ **do**
      $packet \leftarrow packet + Port.read()$
    $Message \leftarrow packet.split(",")$
    for $\forall peripheral \in Peripherals$ **do**
      if $peripheral.label == Message[0]$ **then**
        $Current \leftarrow peripheral$
    for $datum \in peripheral$ **do**
      if $datum.type[a] == Float$ **then**
        $floatMembers[a] = Message[a]$
      if $datum.type[a] == String$ **then**
        $stringMembers[a] = Message[a]$
      if $datum.type[a] == Int$ **then**
        $intMembers[a] = Message[a]$
  for $control \in Controls$ **do**
    $packet \leftarrow$ "&"
    for $datum \in control$ **do** $packet \leftarrow packet + String(datum)$
    $packet \leftarrow$ "EOL"
    Port.write(packet)

type selection, reducing packet size.

B. Message Definitions

In keeping with the design goal of matching the operational characteristics of the data bridge to ROS, we define two generic ROS message types to correspond to transmitted data. Each message type contains variable length arrays to store data, illustrated in Figure 3.

On initialization of the ROS bridge node, the configuration file describing the message is read, generating instances of these topics. This configuration file serves to replace a concrete message definition for each data packet, enabling broad scalability. Further, the parameters for the hardware communication layer are within this configuration file.

Each message contains three primary fields- arrays for containing floating point, string, and integer data. In both the ROS and firmware packages, the data type field determines which array a specific datum is stored in. For instance, if the 4th member of a packet is an integer, then it will be stored in the 4th element of the integer member array, This indexing structure lets the MCU side system store any data type within a single object.

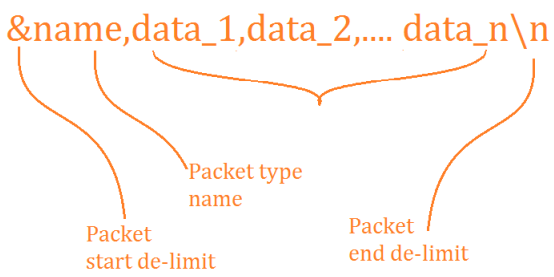

Fig. 2. Transport-layer packet structure

C. MCU Firmware

The primary component of the firmware side of the bridge is a C package, which defines the object handlers for both types of message, as well as a utility function for the transmission layer.

Within this package, there is an allocation for master lists of controls and peripherals. For controls, this list provides access to the variable list associated with the transmission, and for the peripherals the list of transmissions to be made. Objects for each message are created within the main execution loop and populate the list at time of creation.

Each peripheral initializer takes as input the transmission channel, name of the packet, number and type of data members associated with the transmission. The constructor for controls follows the same profile, where the process of variable insertion is handled upon receipt of a serial packet. Within the serial parser, a packet label is searched in the list of packet names for the associated control object, and data is placedinto the object's corresponding data arrays.

The transmission read function required as input only a pointer to the object handler, and therefor is suitable for use within interrupt routines.

D. CPU Software

For the ROS implementation the task of reading, parsing, and writing to the transmission is augmented by the additional task of building and maintaining the ROS topic space. The ROS node package defines object types for the peripherals and controls, and packet parsing and variable storage components are exact parallels of the used in the MCU, just as the message structure and corresponding objects are analogous.

The ROS component begins with the configuration file, an example of which is illustrated in Figure 4. This file defines the ROS topics and subscribers. It begins a subscriber topic for each control, and a publisher for each peripheral. The primary operation loop of the script alternately reads the transmission queue, collects peripheral reports and publishes them into the ROS topic space.

By contrast, control topics for the MCU are attached to

Control .MSG	
*Header	
String	name
int	number_members
int[]	data_types
float[]	float_members
string[]	string_members
int[]	int_members

Peripheral .MSG	
*Header	
String	name
int	number_members
int[]	data_types
float[]	float_members
string[]	string_members
int[]	int_members

Fig. 3. ROS messages corresponding to Controls and Peripherals

```
# Essential Serial port information
<SERIAL>
  <COM>/dev/ttyACM0
  <BAUD>115200

<CONTROL>
  <DEVICE_NAME>relays
  <NUM_PARAMETERS>3
  <DATA_TYPES>2,2,2

<PERIP>
  <PERIP_NAME>block_1
  <NUM_READINGS>14
  <DATA_TYPES>2,2,2,2,2,2,2,0,0,0,0,0,0,2
  <UNITS>
bool,bool,range,range,range,adc_raw,adc_raw,roll,pitch,yaw,accx,
accy,accz,temperature
```

Fig. 4. Example of the configuration file for building a set of publishers and subscribers in the ROS bridge node, structured so that each message does not require a custom MSG definition in ROS

the corresponding subscribers. When one of these topics is published, the subscriber collates the data from the publication, formats it into the transmission format, and writes it to the transmission medium.

III. Implementation System

To experimentally evaluate the performance of the protocol, we apply the above defined software components to the instrumentation of low-level sensors on a robotic platform.

A. MCU hardware

We have built a custom interface board based on the PJRC Teensy 3.6 Arduino-compatible microcontroller. This MCU was selected for its high clock rate and large amount of IO, which considerably simplifies the PCB design.

Several of the sensors communicate over multi-device protocols, with 32 total devices reporting a total of 56 measurements. In our application, the sensors are arranged in 4 blocks of 8 each, with the control outputs being routed through individual terminal lines on the board.

B. Transmission Medium

For this system, we are using the built-in USB/Serial connection for transmission. The Teensy USB/Serial adapter always communicates at the USB limit of 12 Mbit/s, but we enable the standard hardware serial as well, enabling alternate baud rates for testing the effect of transmission speed on queuing loads and data rates.

C. Sensors

The sensor set was chosen to implement navigational assistance, making for a particularly diverse set of sensors.

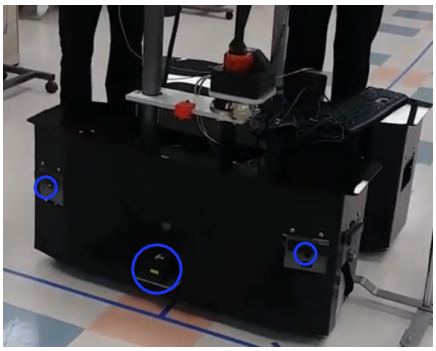

Fig. 5. Image of the ARNA mobile robot, with the locations of the two front-side sensor blocks and the LIDAR highlighted

This is useful for testing as it enables increasing the number of sensors, variation in the data types, and the proportion thereof. As mentioned, the sensors are arranged in four 'blocks'. Each block contains three ultrasonic distance sensors, two button based contact sensors, two IR distance sensors for cliff detection, and one 9-DOF IMU.

D. Testbed Validation

To validate the efficacy of ROSFuse in a working context, we implement it on the Adaptive Robot Nursing Assistant (ARNA) platform, performing a navigation task. On this robot (seen in Figure 5), ultrasonic sensors are used for obstacle avoidance, and a LIDAR range scanner is used for autonomous navigation. In order to demonstrate that the protocol is working effectively, utilize the range of the LIDAR scan corresponding to the sensing region of ultrasonic sensors in the same area. We compare these measurements as a function of time- one collected via the protocol and one independent of it.

To properly correlate the regions, we collected both LIDAR and ultrasonic readings during a mapping exercise. Three of the 12 ultrasonic sensors on the robot share range with the LIDAR sensor, and within the LIDAR sweep, certain regions (θ_{L1}, θ_{R1}, θ_{R2},) correspond to their visibility range, as Illustrated in Figure 6. We average the distance readings from the LIDAR scan across these areas.

The combined data set for these four sensors is plotted in Figure 7, where the LIDAR data is divided into the three averaged regions corresponding to each ultrasonic sensor. On this graph, we can see that the Ultrasonic readings track well with the LIDAR data, staying at each time step within about a 15% margin of the LIDAR data, which is well within the measurement uncertainty of the ultrasonic sensors.

One notable observation is that there is no ultrasonic tracking periods of distances greater than approximately 3.2m. This is due to hard-coded limits on the timing period of the ultrasonic sensor readings which limit their detection range to 2^7 (128) inches.

IV. Data Collection & Analysis

In this section, we present diagnostic data illustrating the performance of the protocol under varying loads. In particular, we examine effectiveness using observations of these trends to predict behavior over a wider range of conditions.

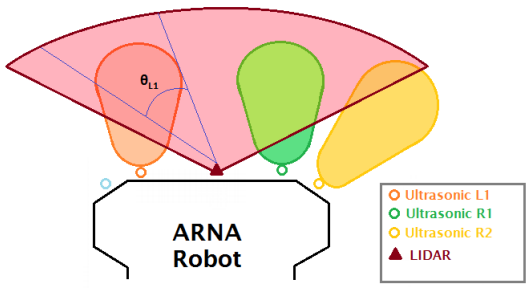

Fig. 6. Illustration of the placement of the LIDAR and ultrasonic sensors, with relative sensing ranges shown (not to scale), and the corresponding angle in the scan range to ultrasonic L1, θ_{L1}

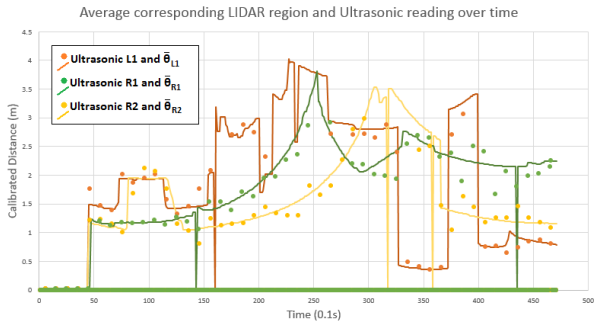

Fig. 7. Plot of data from LIDAR corresponding region and Ultrasonic sensors mounted on the mobile robot as a function of step time (100 ms increments). In this figure, solid lines are the LIDAR readings, and single points are ultrasonic readings.

A. *Transmission speed*

To evaluate transmission speed, we implemented an interrupt-based timing check via the internal clock counter on the MCU, and the microsecond precision system clock on the CPU. For each of these assets, we collected interval period data across a range of transmission delays from 1ms to 80ms, and packet sizes from 1 to 10 random floating point numbers. Figure 8 illustrates this averaged timing data. In this plot, we can observe the trending across both time- and data-density.

The first, and most important, observation we can make about these curves is that all are linear across both data density and period of transmission, with the correlation coefficient $\sigma \geq 0.98$. Further, each loading curve possesses an intercept within 3% of the average, indicating that the offset due to execution of ROSFuse is nearly constant with respect to packet size.

Second, the latency curves, illustrates that the overhead induced by ROSFuse is constant across period and load, fixed at 187 μs. Further, the high correlation across different periods indicates a well-fit latency curve, as transmission lag does not vary with transmission period.

Figure 9, similarly, illustrates the relationship between transmission time and packet size across the same period delays as measured for the MCU side of the protocol. In addition to bearing the same linear relationship, there is a

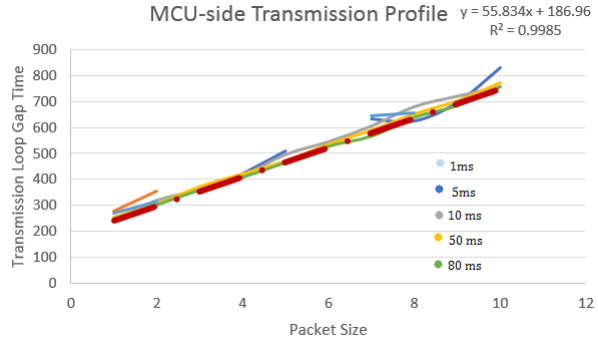

Fig. 8. Loading Profile for the MCU side protocol, illustrating the linear profile, with the fit curve revealing the processing overhead limit of 187 μs and the per-datum rate of approximately 55μs. Note that this is the upper bound of latency, not loop-back, timing

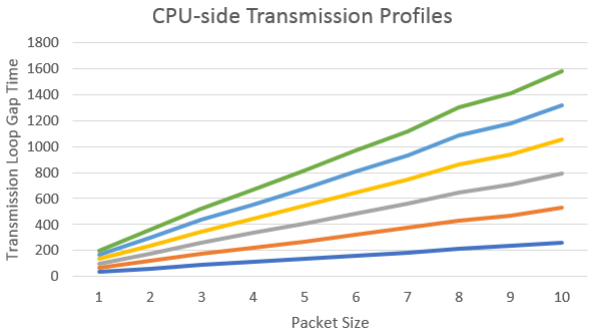

Fig. 9. Loading Profile for the CPU side protocol, illustrating the clear linearity as a function of packet size, across all sampled transmission periods

consistent change in slope with each cange in delay period, indicating that the rate change in transmission speed is constant with respect to the delay, meaning that the overhead latency is indeed constant, and due to factors on the MCU side, which in this case bolsters the argument towards non-length dependent overhead. This is to say that changes in packet size influence the transmission latency only by virtue of demanding longer transmission timesb

B. *Packet Loss*

To examine packet losses, we slow the transmission clock to 9600 baud, focusing on the MCU reception side. The CPU is easily able to outpace the MCU, even at maximal data rates. Conversely the CPU is capable of overwhelming the processing of the MCU at this lower transmission speed, with periods from 1ms to 20ms all bearing some degree of packet loss. The packet sizes at which loss occurs spans from nearly all at 1ms to only intermittent losses at 20ms. The data representing these losses is shown in Figure 10, along with a parallel comparison to a direct data transmission via ROS Serial with no special handling, for comparison.

We also note that packet losses occurring at measurable rates have a consistent local peak loss around the middle of the data rate sweep. We interpret this as actual transmission time varying little between packet sizes and processing time generally being faster than transmission time- therefor, short packets are processed faster, but more frequently, while long packets have an effective transmission delay, allowing the

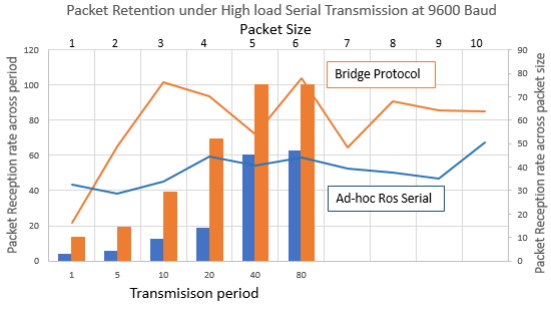

Fig. 10. Illustration of packet transmission as a function of transmission period and average throughput over packet size, showing the functional inverse relationship between packet information density and transmission throughput, and comparing losses between the bridge protocol and brute-force ROS Serial

MCU enough time to parse the data before the next packet finishes arriving.

Under these adversarial load conditions, we still receive a significant proportion of the transmitted data, with the average packet retention rate being around 13.7% for a single datum transmission at a 1 ms period. For a scope comparison, at 9600 baud, each character is allowed approximately 1.04 ms to transmit, meaning the 1ms period consumes 96% of the available transmission time. By contrast, a direct transmission with bare ROS serial retains only 4% of packets under these conditions.

The critical factor underwriting this relatively low loss rate is the rapid packet checking. When a malformed packet Arrives at the bridge, it is discarded as soon as the error is detected, freeing buffer space. ROSFuse may then catch the next packet before buffer overflow occurs. Given that the latency overhead is known to be 187 μs, the 1-datum latency 56μs, and the transmission bound of 1040μs, the available time for packet transmissions is $\frac{1040\mu s - 187\mu s}{56\mu s}$, fitting 15.5 transmissions per period. At the 96% consumption rate for 1 kHz transmissions, this yields a 14.9% upper bound for retention. Marginally more than our observed rate, but packet discard naturally cannot occur before some portion of the bandwidth is used reading the packet.

V. Conclusion

In this paper, we have presented and evaluated the performance of ROSFuse, a low-overhead, highly robust variable data-share system for integrating low-level hardware systems with ROS. We have described how this system offers benefits over other extant solutions to the problem of ROS integration by being more flexible and scalable than other software, lower overhead than framework models which seek to impose a design strategy, and sufficiently robust and efficient to outperform the native ROS protocols. Further, we illustrated the ease of implementation by utilizing the same definition file setup as ROS packages. This eliminated the need for source code changes to implement new hardware or modifications to old hardware. We also illustrated through experiments that the performance of the system remains effective under increasing loads. These experiments also demonstrated scalability, with the same hardware and software being used to test the 32 sensor board and the 10 reading performance experiment, observing high rate communication of 56 values with low latency (as tracked by the LIDAR comparison to the ultrasonic readings) directly into the ROS topic space.

Between the non-coding modular implementation for adding or changing data types, the paired bi-directional, medium-agnostic nature of the protocol, and the consistently low latency and linearity of speed over load, we believe that ROSFuse meets the standard of modular adaptability and scalability of ROS itself.

One issue relating to protocol stability is the relative processing speeds of the two processors. We would, going forward, like to examine the parameters bounding communication between two MCUs and two CPUs to investigate symmetric performance optimizations

Additionally, ROSFuse supports a single ROS master with multiple peripheral devices, however we have not, as yet, performed any testing under these conditions. It would be beneficial for the adoption of the protocol to parametrize the performance of the ROS node under multiple channel processing- current experiments do not incorporate any timing data associated with device level switching.

Acknowledgements

This work has been supported by US National Science Foundation grant 153412, and through the Kentucky EPSCOR KAMPERS project 1849213. The authors are with Next Generation Systems at the Department of Electrical and Computer Engineering Louisville, KY, USA.

References

[1] A. Kapoor, A. Deguet and P. Kazanzides, "Software components and frameworks for medical robot control," In *Proceedings 2006 IEEE International Conference on Robotics and Automation,*, pp. 3813-3818, IEEE 2006.

[2] Groza, Voicu. "Distributed framework for instrumentation hardware/software codesign." *Proceedings of the 17th IEEE Instrumentation and Measurement Technology Conference [Cat. No. 00CH37066]. Vol. 3.*, pp. 1567-1570. IEEE 2000.

[3] Arumugam, Rajesh, Vikas Reddy Enti, Liu Bingbing, Wu Xiaojun, Krishnamoorthy Baskaran, Foong Foo Kong, A. Senthil Kumar, Kang Dee Meng, and Goh Wai Kit. "DAvinCi: A cloud computing framework for service robots." *2010 IEEE international conference on robotics and automation,* pp. 3084-3089. IEEE, 2010.

[4] Petersen, Robert, and Ron Harrison. "Implementing a protocol abstraction layer architecture for interfacing heterogeneous software and hardware systems." *IEEE Autotestcon, 2005.,* pp. 240-247. IEEE, 2005.

[5] Vuletic, Miljan, Laura Pozzi, and Paolo Ienne. "Programming transparency and portable hardware interfacing: Towards general-purpose reconfigurable computing." *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004.,* pp. 339-351. IEEE, 2004.

[6] Cremer, Sven, Fahad Mirza, Yathartha Tuladhar, Rommel Alonzo, Anthony Hingeley, and Dan O. Popa. "Investigation of human-robot interface performance in household environments." In *Sensors for Next-Generation Robotics III, vol. 9859,* p. 985904. International Society for Optics and Photonics, 2016.

[7] Mayoral, Victor, Alejandro Hernández, Risto Kojcev, Iñigo Muguruza, Irati Zamalloa, Asier Bilbao, and Lander Usategi. "The shift in the robotics paradigm—The Hardware Robot Operating System (H-ROS); an infrastructure to create interoperable robot components." *In 2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS),* pp. 229-236. IEEE, 2017.

[8] Araújo, André, David Portugal, Micael S. Couceiro, and Rui P. Rocha. "Integrating Arduino-based educational mobile robots in ROS." *Journal of Intelligent Robotic Systems 77, no. 2*, pp. 281-298. 2015

[9] Strohmer, Beck, Anders Bøgild, Anders Stengaard Sørensen, and Leon Bonde Larsen. "ROS-Enabled Hardware Framework for Experimental Robotics." In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig),* pp. 1-2. IEEE, 2019.

[10] Ratasich, Denise, Bernhard Frömel, Oliver Höftberger, and Radu Grosu. Generic sensor fusion package for ROS, *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS),* pp. 286-291, IEEE 2015.