

Interprocess Communication for Robotics Applications

Edward M. Roderick*

Abstract—Modern robotic systems are developed as a collection of independent processes. For these processes to function together, an inter-process communication system (IPC) must be implemented. Many prebuilt systems are available and this paper presents five popular options (UDP, TCP, ZeroMQ, ACH, and ROS) as candidates for robotics. An analysis is presented for each based on communication latency and design metrics as applied to robotics. An example design is presented with analysis of the IPC selection process.

I. INTRODUCTION

As modern robotic systems grow in complexity, it becomes increasingly beneficial to implement a multi-process control system. This provides a modular system architecture. A modular architecture protects the overall system from individual components failing[1]. Critical system processes can be allocated additional resources to ensure continued operation during a partial system failure. By predefining the inputs and outputs for each subsystem, multiple teams of designers can use differing languages, allowing for optimization of each task. Additionally, the overall system can be distributed across a variety of hardware (full computers, single board computers, micro-controllers)[2], [3], [4].

Many IPC options are available to designers and each must be evaluated on a per system basis to determine the best candidate. This paper will focus on evaluating several mainstream options and how they apply to design metrics of robotic systems. As robots operate in real time, the latency and data integrity of each communication step is of critical importance[5]. For complex systems (humanoids, robots operating in proximity with humans, etc) with multiple processes running concurrently, communication delay can result in actuators responding to obsolete data[4]. This paper serves to provide a guide to evaluating the capabilities of Sockets, Shared Memory, and ROS.

II. BACKGROUND

Several IPC systems that are commonly used today were researched and evaluated. Sockets was evaluated utilizing UDP, TCP, and ZeroMQ. The ACH library was used for an example of shared memory. The details of their use and implementation are presented in the following.

A. Shared Memory (ACH)

Shared memory offers the fastest read/write speeds for communication between processes within a single computer. As part of the POSIX¹ IPC library, shared memory is

available to virtually every robotic system. Real time robotics require that the IPC system be able to prioritize the most recent data over data received sequentially (non head of line blocking)[4], [2], [1]. Basic shared memory provides this by overwriting variables but is susceptible to synchronization issues and all previous data is now lost[2].

ACH² builds on basic shared memory and provides a message bus (channel) between multiple writers and multiple readers[1]. An ACH channel consists of a data buffer and an index buffer that are stored in a shared memory file. The channel provides synchronized access via a mutex and a condition variable. This allows for priority inheritance and prevents starvation[2], [1]. The ACH library is open source and formally verified[2], [4] making it suitable for widespread use.

ACH channels provide high levels of flexibility in communication and programming. They can be configured to read the most recent data or in order it was received[1]. A read command can block until data is received or it can poll the channel as needed. Figure 1 below is an example of an ACH based control system for the Hubo robot.

ACH provides LAN based communications by utilizing sockets messages to transport channel data. Each system running the ACH network daemon (ACHD) can write directly to remote shared memory resources connected to the network[4]. This allows for CPU intensive processes to be offloaded from the robot to a local high performance computer.

B. Sockets: TCP, UDP, and ZeroMQ

Sockets provides inter-process communication over a network via the TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). It is built into most operating systems which makes code highly portable. Sockets communicate with a client/server architecture with either point to point or multicast messages[6]. Utilizing IP addresses and existing internet infrastructure allows processes to communicate with remote locations. This makes it possible to quickly and cheaply connect the robot to sensors, controllers, and embedded systems. As part of the POSIX library, sockets allows many different operating systems and programming languages to communicate with no extra software[1], [7]. The library can easily be integrated and is found as a core transport mechanism for many existing robot frameworks[8]. Modern network routers offer gigabit connections enabling sockets to provide fast communication between distributed processes.

* Author is affiliated with George Mason University, Fairfax VA, 22030, USA. eroderi2@gmu.edu.

¹POSIX: Portable operating system interface. A set of standards for maintaining compatibility between operating systems.

²Available at <https://github.com/golems/ach>

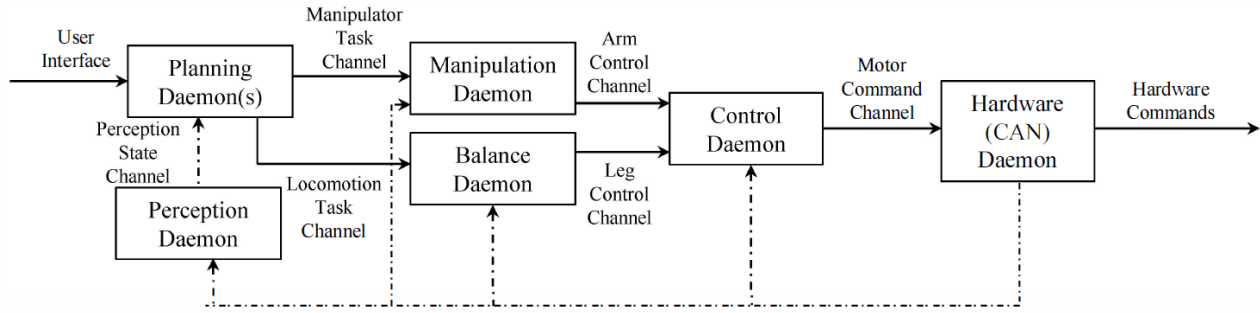


Fig. 1. ACH Communication Flow Chart for HUBO Robot[2]. Diagram shows control daemons used and communication paths used in the HUBO humanoid robot. Multiple publishers and subscribers for each channel show flexibility of implementation.

Sockets message buffers are head of line blocking and disregard new messages when full[2]. This is problematic for robotics where sensor data is of critical importance.

1) *TCP*: TCP offers robust messaging between a single client and a single server process. Initial handshaking is required during connection and can cause additional latency for the initial message[7]. Each message transmission triggers a response from the recipient. This allows TCP to identify dropped messages and retransmit[7]. While good for data integrity, response messages and retransmission increase latency.

2) *UDP*: UDP offers one quick directional communication between processes. UDP supports client/server and multicast topologies. Processes are bound to IP address(s) and port number(s)³. There is no handshaking or direct link between the processes. This allows multiple clients to send data to a single server. UDP inherently has less latency than TCP due its 'fire and forget' approach to messaging. Data integrity suffers as there is no way to identify missed messages[7]. This method favors transmission speed and has less latency than TCP.

3) *ZeroMQ*: ZeroMQ⁴ (ZMQ) is a messaging system that extends on the foundations of sockets. It provides additional functionality to TCP by sending messages to a system topology instead of specific IP addresses[9]. ZMQ prevents sending to specific IP addresses by design. The available messaging patterns is open ended. Common patterns include publish/subscribe and request/reply[9], [10]. Figure 2 below shows examples of how topologies can be arranged.

Only one messaging pattern is allowed for a topology and they cannot be interconnected[9]. This guarantees that data will arrive at their intended location(s). ZMQ separates its stack into end to end and hop by hop layers. Unlike TCP/IP, each ZMQ end to end protocol has its own hop by hop protocol[9]. This allows each specific messaging pattern to have their own routing functionality. Bidirectional messages can be sent to specific nodes in the topology. Each intermediary node can determine if downstream sections of

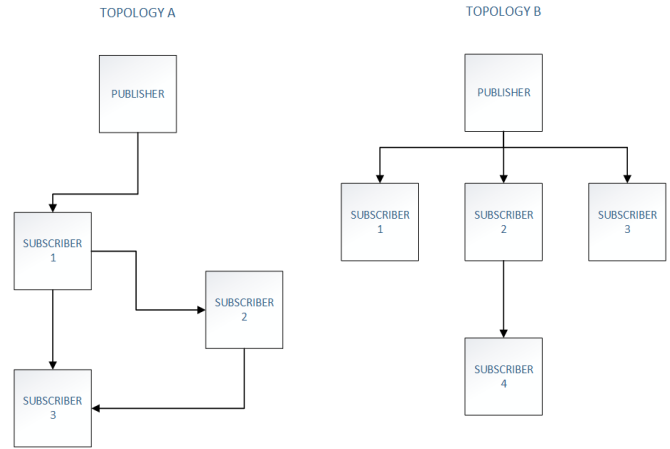


Fig. 2. ZeroMQ Example Topologies[9]. Figure shows two example topologies that can be implemented with ZeroMQ. Publishers are shown sending data to subscribers with multiple paths.

the network are unreachable and can signal the original client to resend now or wait until connectivity has been reestablished.

ZMQ provides synchronization for data in multithreaded applications[10]. This functionality is hidden from the developer and allows for complex, scalable messaging in multithreaded systems[10]. Common methods for dealing with synchronization (mutex locks, etc) can block threads when scaled to high number of processes. ZMQ abstracts complex lock free synchronization methods behind the familiar interface of sockets[10].

C. ROS

ROS⁵ (Robot Operating System) is a flexible framework for developing robotic systems. Offering more features than an IPC, ROS utilizes TCP as its core transport mechanism for communication between processes[11], [12]. In addition to the IPC, ROS offers a wide library of prebuilt tools and a vast user base from which to draw on modular solutions that can plug into any ROS system[11], [5]. This reduces development time and amount of low level code that needs

³Multiple IP addresses and port numbers only available for a multicast UDP socket

⁴Available at www.zeromq.org

⁵Available at www.ros.org

to be written for each piece of hardware[3]. Figure 3 below shows an example ROS network diagram.

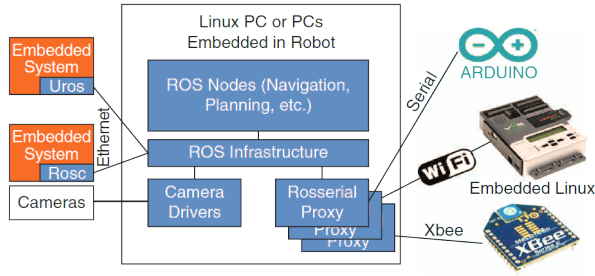


Fig. 3. ROS Example Network Diagram[3]. Network diagram shows how a ROS system can be setup to incorporate a wide variety of off the shelf hardware and the internal processes the robot would need for communication.

Commercially available embedded systems can interface with the ROS system by adding interface processes (ROS topics)[3] to the core system. ROS was designed to support higher level robot functions instead of the control of individual motors/sensors/etc. Integration of subsystems is left to the designer as ROS does not support fieldbuses[3]. ROS can be implemented as a combination of an embedded PC, proprietary systems with custom interfaces, and ROS Messaging. Embedded PCs provide smooth ROS integration but fail to offer hard real time support[11]. Existing robots can be managed from ROS through topics acting as translators. This abstracts the lower level programming and allows for the existing systems to operate in real time[11], [3]. The IPC system for ROS operates on remote procedure calls and publish/subscribe protocols for messaging.

The ROS IPC system allows for custom messages interface with embedded systems. Rosserial, rosc, and Rosbridge are methods of passing these messages[12], [3]. Rosserial provides a proxy over a C++ client that can be ported to any system that supports the language. Rosserial provides a ROS-like API to publish, subscribe, offer and consume RPC services over serial. The proxy can act as a bottleneck increasing message latency. Rosc allows for direct connection and messaging to systems supporting C, but TCP/IP overhead can overwhelm low level peripherals. Rosbridge allows dynamic socket and websocket access to the full capabilities of ROS[3].

III. TEST PROCEDURES

In order to evaluate each of the discussed IPC methods, a test system was developed that incorporated a laptop, 100mbps wired network connection, a Raspberry Pi single board computer (SBC), and an OpenCM9.0 microcontroller board (Arduino architecture) designed for command and control of Dynamixel brand servomotors. The system transports messages over ethernet, serial, and USB. For consistency, the serial link was set to the same baud rate (9600) as the USB link. The objective of this test was to measure the round trip latency of a message by utilizing each of the aforementioned

IPCs. Only one communication link (Link 1 or Link 2) varied for each test and UDP was selected as the control for the other. Figure 4 below shows the overall system diagram of the test setup.

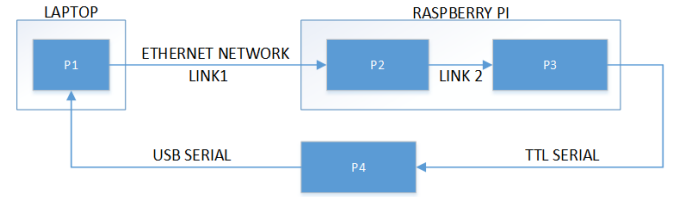


Fig. 4. Test Setup Block Diagram

Process P1 generates a random one byte message send sends the data over the network to process two. P2, P3, and P4 all have blocking read functions and immediately pass the message on to the next. This was done to simulate a more complex system while minimizing processing delays from the results. P1, P2, and P3 are written in Python while P4 was written in its native Arduino code. P1 utilizes the most accurate time function available to python. All communication was performed synchronously to eliminate processor clock frequencies affecting results. 10000 samples were processed for each test. The logical flow chart of the program can be seen below in Figure 5.

The objective was to simulate each IPCs ability to receive, process, and act on incoming sensor data. To accomplish this, the test was designed to send single byte messages at a high enough frequency to emulate sensor data without overloading each processor. As noted in the test results seen in prior research[2], the initial communication of TCP can cause a spike in message latency. To eliminate this impact from our data we utilize a test message that must be successfully received prior to the start of data logging.

The theoretical minimum latency for each communication link was evaluated to ensure the results obtained were a factor for the IPC method and not of processing delays and technology limitations. For the 100Mbps ethernet link, the theoretical transit time for a message would be 8 microseconds. Both the serial and USB communication links were set to 9600bps with a minimum transit time of 0.833 ms. This results in a total minimum latency of 1.675 ms for a single byte message. As the results will show, recorded latency ranged from 3 to 6 ms. All data being recorded well above the theoretical min concludes that the test yields viable results.

IV. TESTS

A. Results Summary

After completion of all testing, no IPCs dropped any messages during transit. This can be attributed to the ideal testing conditions. Hardwired network communication and single byte package size allowed for both TCP and UDP packets to be received despite UDP's lower inherent reliability. The introduction of a wireless networks or lower quality routers may cause more of a differentiation to arise. The fastest

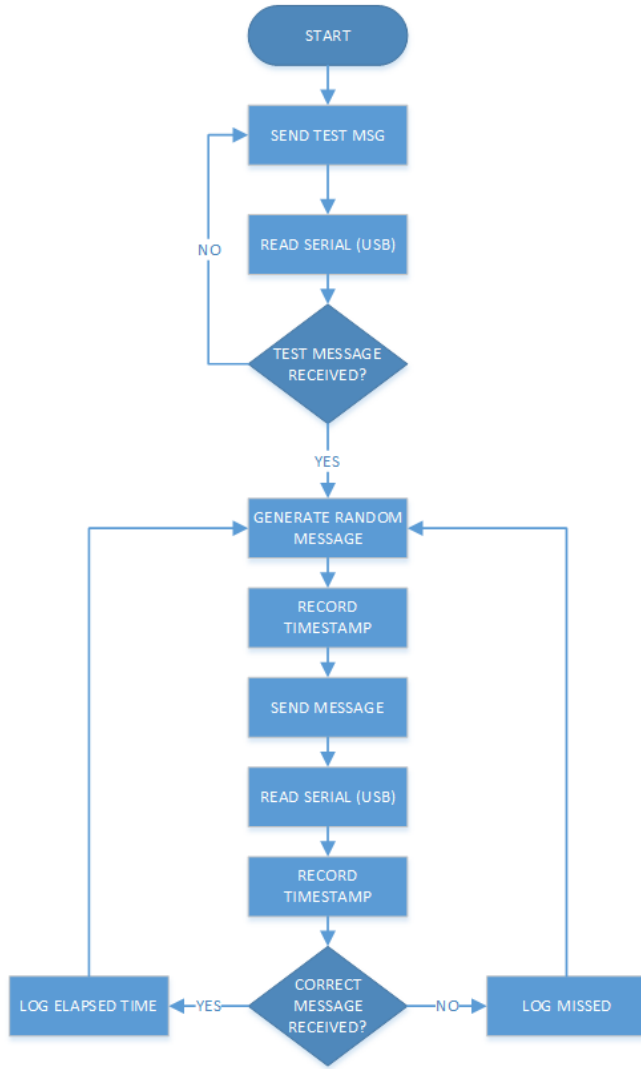


Fig. 5. Test Program Flow Chart

overall system consisted of UDP network messages and ACH messages for the internal communication through the Pi. There was negligible difference in transit time between TCP and UDP over the network. Similar delays at this stage of the test can be attributed to the routers throughput. TCP response messages are processed at or below the microsecond accuracy level and does not impact our test results. A summary of all data collected is presented below for the network link in Table I and the internal Raspberry Pi link in Table II.

B. UDP Results (Control)

The control of this experiment utilized UDP messages over the network and UDP messages sent to the loopback address on the Pi. No missed messages occurred despite UDP being the least reliable IPC method tested. Consistent results occurred with a standard deviation of 1.441 ms and the average transit was 3.834 ms. This was the second fastest configuration tested. Figure 6 below shows a summary of collected data with outliers removed for clarity.

TABLE I
RESULTS SUMMARY: NETWORK LINK

| Item | Network: Link 1 | | | | |
|--------------------|-----------------|--------|--------|-----|---------|
| | UDP | TCP | ZMQ | ACH | ROS |
| Average (ms) | 3.834 | 3.829 | 5.285 | N/A | 13.09 |
| Min (ms) | 3.359 | 3.399 | 3.943 | N/A | 4.860 |
| Max (ms) | 95.063 | 42.006 | 92.746 | N/A | 706.840 |
| Std Deviation (ms) | 1.441 | 0.768 | 3.639 | N/A | 1.292 |

TABLE II
RESULTS SUMMARY: RASPBERRY PI LINK

| Item | Raspberry Pi: Link 2 | | | | |
|--------------------|----------------------|--------|---------|--------|---------|
| | UDP | TCP | ZMQ | ACH | ROS |
| Average (ms) | 3.834 | 4.774 | 5.000 | 3.752 | 17.571 |
| Min (ms) | 3.359 | 4.079 | 3.853 | 3.219 | 5.680 |
| Max (ms) | 95.063 | 43.974 | 161.833 | 73.533 | 709.943 |
| Std Deviation (ms) | 1.441 | 1.160 | 3.171 | 1.705 | 9.958 |

C. TCP Results

TCP was tested in both Link 1 and Link 2 configurations with UDP as the second link. No missed messages were recorded as expected for TCP. For the network link, TCP performed with negligible difference to UDP. This would suggest that for small data packets and reliable network connections the additional delay incurred by TCP's response message is negligible. Figure 7 below shows a summary of collected data with outliers removed for clarity.

Differentiation arises when TCP is utilized for the Link 2 loopback transport. Additional latency was recorded for an average transit time of 4.774 ms and a standard deviation of 1.160 ms. This suggests the additional TCP overhead becomes an issue for embedded systems and would not be a valid candidate for an internal IPC. Figure 8 below summarizes the Link 2 test results.

D. ZMQ Results

ZeroMQ performed the slowest in both links of all the pure IPCs tested. It is expected that ZMQ performs slower than TCP as it includes additional packet information around

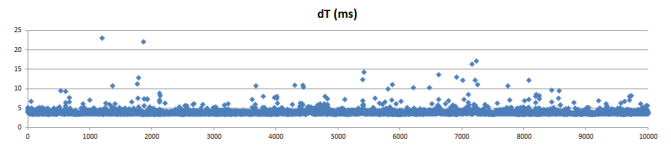


Fig. 6. UDP Control (Link 1 and 2) Test Results

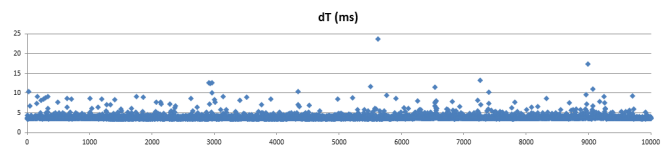


Fig. 7. TCP Link 1, UDP Link 2 Test Results

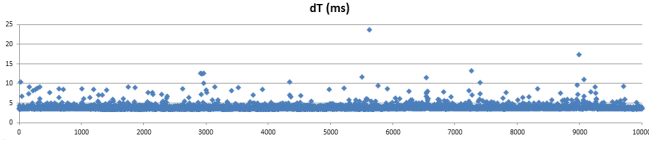


Fig. 8. UDP Link 1, TCP Link 2 Test Results

a standard TCP message. This test only determines the speed of a single process communicating in client/server pair and does not evaluate potential gains when implemented in a multifaceted topology. ZMQ was tested in both Link 1 and Link 2 configurations and results can be found below in Figure 9 (Link 1) and Figure 10 (Link 2) below.

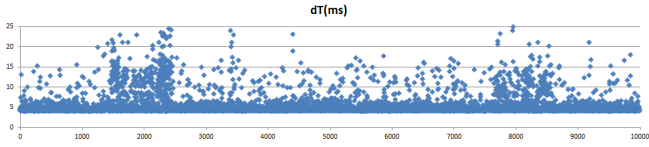


Fig. 9. ZMQ Link 1 , UDP Link 2 Test Results

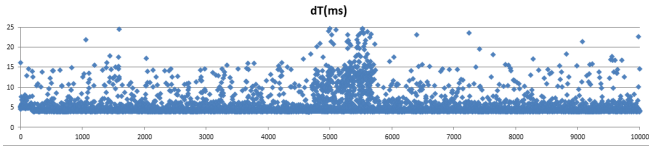


Fig. 10. UDP Link 1, ZMQ Link 2 Test Results

The data shows slower performance in both link configurations than UDP, TCP, or ACH. Certain periods of increased latency were observed and could be a result of messaging speed and increased overhead of ZMQ overwhelming the SBC for short durations. If ZMQ would be implemented for all links in a system, the delay would be compounded with each step.

E. ACH Results

ACH performed the fastest of all IPC methods tested. For consistency, UDP was used as the Link 1 transport over the ACHD library. ACHD can be configured as TCP or UDP messages and it was concluded that similar results would have been produced. Average transit time was 3.752 ms with a standard deviation of 1.705 ms. ACH exceeds the speed of UDP when transmitting internally and has the synchronization benefits of ZMQ without the additional latency observed. Figure 11 below shows a summary of collected data with outliers removed for clarity.

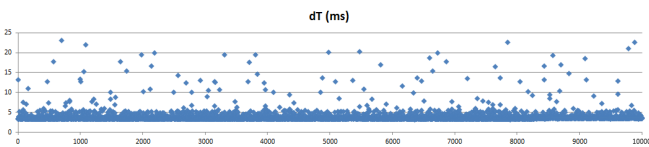


Fig. 11. UDP Link1, ACH Link 2 Test Results

F. ROS Results

ROS performed the slowest of all IPC methods tested. The test of link 1 was performed by running a ROScore instance on the full computer and a publish/subscribe link built between each process in the communication path. Running the ROScore on the networked computer introduced additional latency from network communications, but freed up processing power on the SBC. This test incorporates ROS messages in both link 1 and link 2 as it is the more logical implementation for a fully networked ROS system. The additional overhead of ROS messages and the extra communications with the ROScore process substantially impacted latency. Results can be found below in Figure 12.

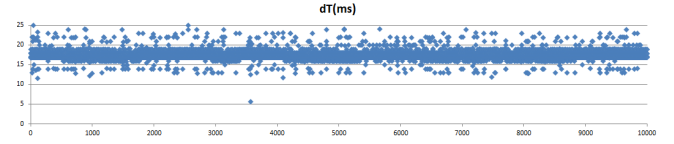


Fig. 12. ROS Link 1 and Link 2 Test Results

To further test the impact of ROScore on the system. A link 2 test was established to simulate a fully contained ROS system on the SBC with UDP communications over the network. Increased latency was recorded despite the fewer messages to a remote ROScore. This indicates that for embedded systems, the increased processing power required to run ROScore is detrimental to the operation of the robot. Results are found below in Figure 13.

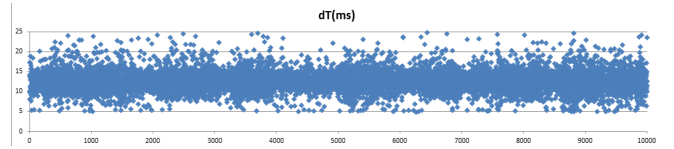


Fig. 13. UDP Link 1, ROS Link 2 Test Results

V. RESULTS

The selection of an IPC method for robotics must be done on a case by case basis by evaluating the design goals and intended implementation. Each of the IPCs discussed in this paper offer strengths and weaknesses that the designer will need to accommodate. Code portability, data integrity, message latency, and system architecture all play an essential role in the selection process. While standardizing a communication protocol for the entire system may make implementation easier at the onset of a project, additional effort to optimize the system at a link by link level will vastly improve the operation of the robot. The results of the testing and research done in this paper conclude that combining different IPC methods within a robotic system allow for optimal message latency and flexibility of the system.

The ACH messaging system provides the ideal combination of speed and flexibility and should be utilized for internal messaging within a contiguous set of hardware. With the ability to configure an ACH channel as either blocking

or non-blocking, the system can perform synchronously or asynchronously depending on the needs for the data. Additional configuration options allow the channel to be set as a FIFO or LIFO buffer without overwriting or losing any message data. The robot can access and react to the most recent sensor data while continuously logging the all data as it arrives. The synchronization provided internally by the data channels abstract the more complicated programming locks and allow for easily implemented, thread safe operation on multicore robots. As multicore embedded SBCs become increasingly available, ACH's synchronization allows for portability and upgradeable between single and multicore systems.

The use of sockets is recommended for all network communication to and from the robot. Sockets is a standard POSIX library and its use allows for modular addition of sensors, controllers, and other networked systems. Robots often operate in a real time environment and message latency is of critical importance. UDP excels at fast one directional communication. Data packets can be designed to be small in size to reduce, if not eliminate, any issues of dropped messages. Additionally, the data rate can be increased to the point where any missed message are made irrelevant by new data before the robot would notice. For situations where minimizing latency is a primary design goal and missed messages can be accepted then UDP is the recommended IPC. Figure 14 below details how an ideal system should be designed for minimal latency.

Processing the large amount of messages into an ACH channel is critical to ensure usability of this method. This eliminates the issues of lost messages after the sockets buffer fills and prevents data obsolescence that would occur from a robot receiving data faster than it can process it. This method can be used with both UDP and TCP. The implementation of TCP is recommended with data integrity is of critical importance. For an ideal system that favors complete data transmission over increased latency then TCP should be utilized in conjunction with ACH. Figure 15 details how an ideal system should be designed for data integrity.

The remaining IPC methods discussed in this paper should be used if a specific requirement dictates a specialized approach. The latency of a ZeroMQ message is detrimental to the real time operation of a robot. The only situation in which ZeroMQ should be implemented is when the system being designed requires a complex topology of processes that need to adhere to a publish/subscribe protocol. Robots are usually developed with a central command process that communicates with sub-processes, thus eliminating the advantages of ZeroMQ.

ROS provides a vast library of modules to draw on for implementing hardware into the system. It is open source and supported on a wide variety of platforms. This reduces the amount of low level coding that needs to be written for each piece of hardware. This is advantageous for quickly building a system and reducing the amount of effort at the programming level. It fails to provide any real time support and the message latency and the head of line blocking

nature of its TCP based messaging is problematic for high data transmission rates. ROS excels at building existing hardware and robotics into a viable system, but is not an ideal candidate for a custom built application.

While every robot is implemented differently, a combination of ACH channels and UDP/TCP sockets will be a viable application for the majority of systems. The flexibility and speed of the combined communication methods create a robust and modular link for real time robots to both internal and external components.

VI. IMPLEMENTATION

An example implementation of the information gathered is presented here. The objective of this design is to move blocks from an unknown starting location to a goal location. This is to be done remotely under user control with the only visual information coming from the robot. Visual information is collected via two webcams attached to a Raspberry Pi. The Raspberry Pi will receive commands from a remote terminal and control 4 servomotors by sending serial commands to the OpenCM motor controller. To achieve the required teleoperation, user input and both video feeds must be transmitted over a network.

The user will operate the robot with a joystick connected to a laptop. One process runs on the laptop to receive information from the joystick and transmit to the robot over the network via UDP. UDP was selected for this communication link to ensure the fastest communication to the robot. Low latency was a priority as any lag in commands being received by the robot will propagate to updating the video feedback and would make teleoperation difficult. Missed messages were of low concern as the frequency of message transmits would be replaced with new commands faster than the user can realize the delay. Command packets can be designed to be two bytes in size to minimize any risk of missed messages.

UDP was also selected for sending both video streams from the robot to be displayed on the user terminal. The video frame information required sending UDP messages at the maximum size limit for a single message. This did increase the probability of a message being missed, but the update rate of the video stream reduced on the impact of missed frames. As the video information is being interpreted by a human and not a computer, lower image resolution and frame rates were acceptable.

In order to have immediate response to user commands, the process collecting data from the joystick needed to operate at a high frequency. This will result in a large amount of messages being transmitted to the Raspberry Pi. Preliminary testing showed that while the user held the joystick down in a certain direction, a large quantities of the same command were stored in the robots UDP buffer. This would cause the robot to continue acting on outdated command information while it processed all received UDP commands. Increasing the update rate of the processes running on the PI to match incoming messages was not an acceptable solution as processing two USB video streams at higher rates was

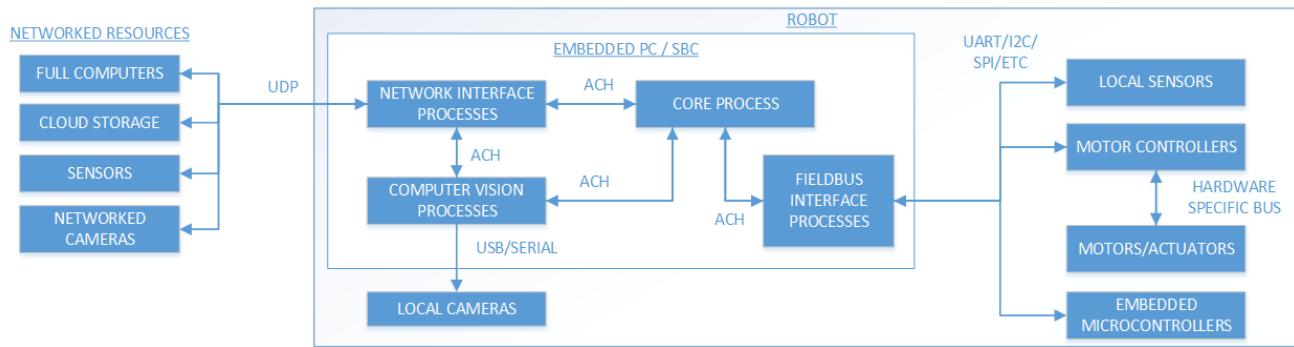


Fig. 14. Ideal System: Minimal Latency. This diagram shows how UDP and ACH IPC methods should be used for relevant system components.

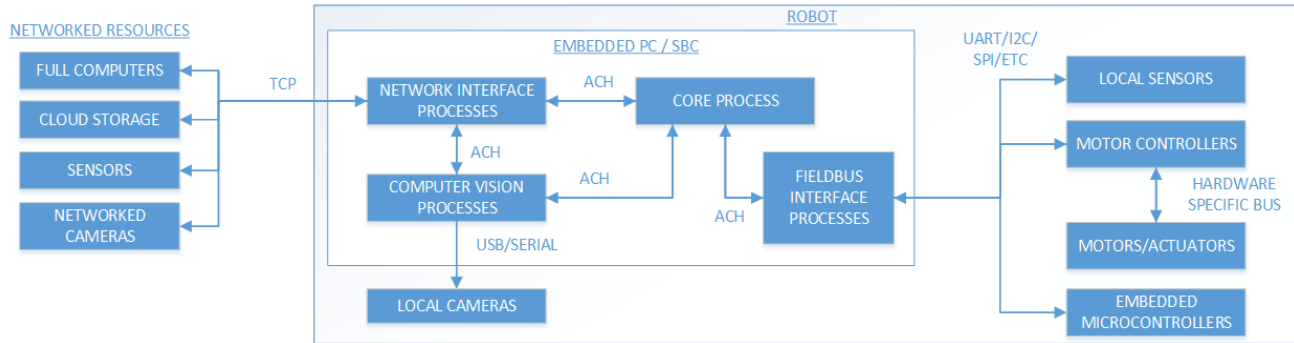


Fig. 15. Ideal System: Data Integrity. This diagram shows how TCP and ACH IPC methods should be used for relevant system components.

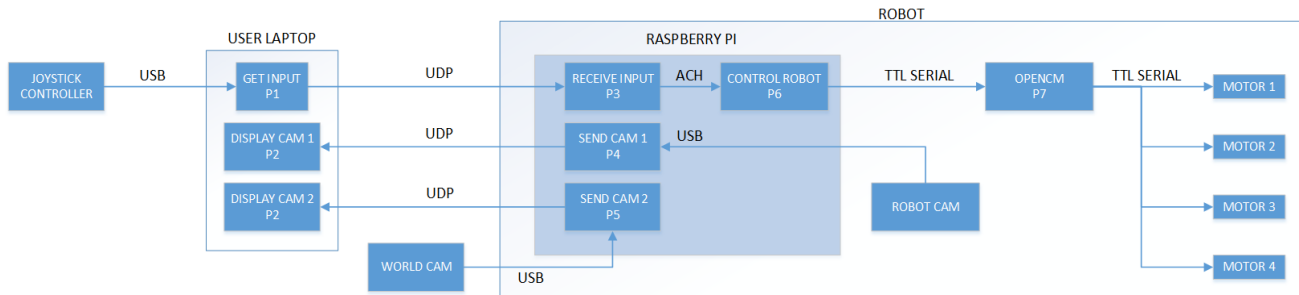


Fig. 16. Crane Robot Communication Diagram

too taxing on the system. This led to errors and corruption of video information.

We needed to utilize the non head of line blocking aspect of the ACH IPC to access the most recent command message at a lower frequency. We incorporated an additional process that would receive UDP message and fill an ACH channel with data. This allowed the command process to update at a rate that matched the video information processed and sent back to the user without overloading the resources of the Pi. Figure 16 shows a block diagram of the processes and communication methods for the robot.

By evaluating the conceptual design and objective of this robot, we were able to narrow down the list of potential IPC methods to use. ZMQ would not be a viable candidate for use as the communication delay would be detrimental to the robots operation. There was no need for a complex communication topology as all of the process communication

was between a single client/server pair. The Raspberry Pi and the OpenCM modules are single threaded processors and no additional synchronization features of ZMQ would be needed. ROS can be installed on the Pi and would have been possible candidate for use in this system except that the entire system would communicate with TCP at its core. The aforementioned head of line issue with command packets eliminates a sockets based approach for the entire system. Utilizing ACH and ACHD for network communications would allow us to achieve all of our design goals, but would require the user terminal to also have the ACH libraries to be installed. Utilizing sockets for the network communication was essential to build the most flexible system possible and allow for a modular input solution. This led to the combined solution of implementing both UDP and ACH for the robot.

VII. CONCLUSIONS

Multiprocess robotics are the preferred method of structuring a modern system. The selection of a communication protocol for these processes to co-exist is a non trivial decision. In this paper, we have reviewed the latency most popular options of TCP, UDP, ZeroMQ, Shared Memory, and ROS.

Tests were performed to evaluate round trip messaging time on a sample system incorporating a laptop computer, wired network connection, SBC and embedded microprocessor. The findings of the tests were applied to a functional system with detailed analysis of choices behind the evaluated ideal IPC system.

The results found through this paper indicate that Shared Memory (specifically ACH) offers the most flexibility of design when implemented on a contained SBC. The ability to switch between FIFO and LIFO without loss of data is ideal for robotics. For network communications the findings support UDP as the primary messaging IPC. UDP allows for messages to be sent with minimal latency from a wide variety of off the shelf hardware. Utilizing this with ACH allows removes the head of line blocking issue of sockets. Designing robotics with a combination of UDP and shared memory creates the optimal solution for robotics.

REFERENCES

- [1] N. Dantam, D. Lofaro, A. Hereid, P. Oh, A. Ames, and M. Stilman, "The ach library: A new framework for real-time communication," *Robotics Automation Magazine, IEEE*, vol. 22, no. 1, pp. 76–85, March 2015.
- [2] N. Dantam and M. Stilman, "Robust and efficient communication for real-time multi-process robot software," in *Humanoid Robots (Humanoids), 2012 12th IEEE-RAS International Conference on*, Nov 2012, pp. 316–322.
- [3] P. Bouchier, "Embedded ros [ros topics]," *Robotics Automation Magazine, IEEE*, vol. 20, no. 2, pp. 17–19, June 2013.
- [4] M. Grey, N. Dantam, D. Lofaro, A. Bobick, M. Egerstedt, P. Oh, and M. Stilman, "Multi-process control software for hubo2 plus robot," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, April 2013, pp. 1–6.
- [5] A. Shakhimardanov, N. Hochgeschwender, M. Reckhaus, and G. K. Kraetzschmar, "Analysis of software connectors in robotics," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, Sept 2011, pp. 1030–1035.
- [6] J. Vella and S. Zammit, "Packet losses of multicast over 802.11n heterogeneous wireless local area network," in *Systems, Signals and Image Processing (IWSSIP), 2012 19th International Conference on*, April 2012, pp. 300–303.
- [7] D. K. Cho, S. H. Chun, J. G. Ahn, Y. S. Kwon, J. H. Eom, and Y. I. Kim, "A udp-based protocol for mobile robot control over wireless internet," in *Robot Communication and Coordination, 2009. ROBOCOMM '09. Second International Conference on*, March 2009, pp. 1–4.
- [8] A. Harris and J. Conrad, "Improving and designing sensors for the simulation environment for autonomous robots (sear)," in *Southeastcon, 2013 Proceedings of IEEE*, April 2013, pp. 1–5.
- [9] M. Sstrik, "mq: The theoretical foundation," 250bpm, Tech. Rep., 09 2011.
- [10] P. Hintjens and M. Sustrik, "Multithreaded magic with zmq," iMatrix Corporation, Tech. Rep., 09 2010.
- [11] S. Cousins, "Is ros good for robotics? [ros topics]," *Robotics Automation Magazine, IEEE*, vol. 19, no. 2, pp. 13–14, June 2012.
- [12] A. Araujo, D. Portugal, M. Couceiro, and R. Rocha, "Integrating arduino-based educational mobile robots in ros," in *Autonomous Robot Systems (Robotica), 2013 13th International Conference on*, April 2013, pp. 1–6.