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 interprocess communication system (IPC) must be implemented. Many prebuilt systems are available and this paper presents four popular options (Sockets, ZeroMQ, ACH, and ROS) as candiates 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. 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 distrubted across a variety of hardware (full computers, single board computers, microcontrollers).

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. For complex systems (humanoids, robots operating in proximity with humans, etc) with multiple processes running concurrently, communication delay can results in actuators responding to obsolete data. This paper serves to provide a guide to evaluating the capabilites of Sockets, ACH, ZeroMQ and ROS.

II. BACKGROUND

A. ACH (Shared Memory)

Shared memory offers the fastest read/write speeds for communication between processes within a single computer. As part of the POSIX[1] IPC library, shared memory is available to virtually every robotic system. Real time robotics require that the IPC system be able to prioiritize the most recent data over data received sequentially (non head of line blocking). Basic shared memory provides this by overwriting variables but is susceptable to synchronization issues and all previous data is now lost[1].

ACH[2] builds on basic shared memory and provides a message bus (channel) between mutliple writers and multiple readers. 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 inhereitence and prevents starvation[1]. The ACH library is open source and formally verified[1] making it suitable for widespread use.

ACH channels provide high levels of flexibilty in communication and programming. They can be configured to read the most recent data or in order it was received. 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.

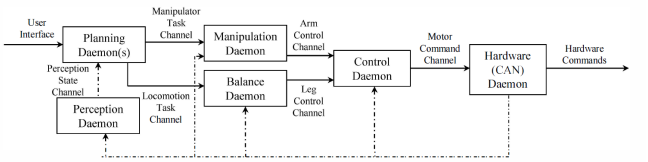

Fig. 1. ACH Communication Flow Chart for HUBO Robot

ACH provides LAN based communications by utilzing 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[2]. This allows for CPU intensive processes to be offloaded from the robot to a local high performance computer.

B. Sockets: TCP and UDP

Sockets provides interprocess communication over a network via the TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). It is built it to most operating systems which makes code highly portable. Sockets communicate with a client/server architecture with either point to point or multicast messages[3]. Utilzing 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. Modern

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

[1]POSIX: Portable operating system interface. A set of standards for maintaining compatability between opearting systems.

[2]Available at https://github.com/golems/ach

newtork routers offer gigabit connections enabling sockets to provide fast communication between distrubted processes.

Sockets message buffers are head of line blocking and disregard new messages when full. 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[4]. Each message transmission triggers a response from the recipient. This allows TCP to identify dropped messages and retransmit. 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 mutltiple 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. This method favors transmission speed and has less latency than TCP.

C. ZeroMQ

ZeroMQ[4] (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. ZMQ prevents sending to specific IP addresses by design. The available messaging patterns is open ended. Common patterns include publish/subscribe and request/reply. Figure 2 below shows examples of how toplogies can be arranged.

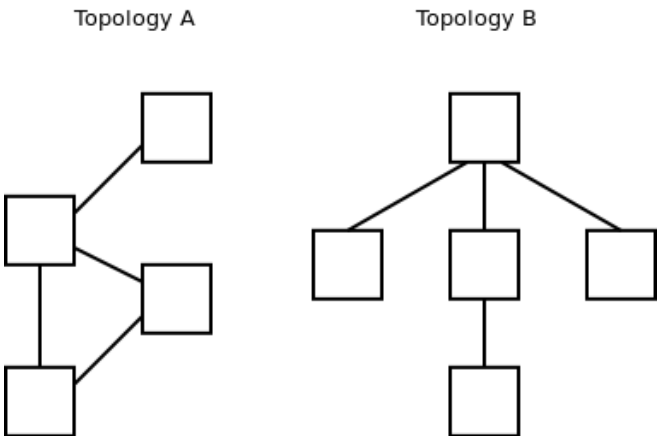

Fig. 2. ZeroMQ Example Topologies

Only one messaging pattern is allowed for a topology and they cannot be interconnected. This guarentees 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[5]. 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 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. This functionality is hidden from the developer and allows for complex, scalable messaging in multithreaded systems. Common methods for dealing with synchronization (mutex locks, etc) can block threads when scaled to high number of processes. ZMQ absracts complex lock free synchronization methods behind the familiar interface of sockets[6].

D. ROS

ROS[5] (Robot Operating System) is a flexible framwork for developing robotic systems. Offering more features than an IPC, ROS utilizes TCP as its core transport mechanism for communication between processes. 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. This reduces development time and amount of low level code that needs to be written for each piece of hardware. Figure 3 below shows an example ROS network diagram.

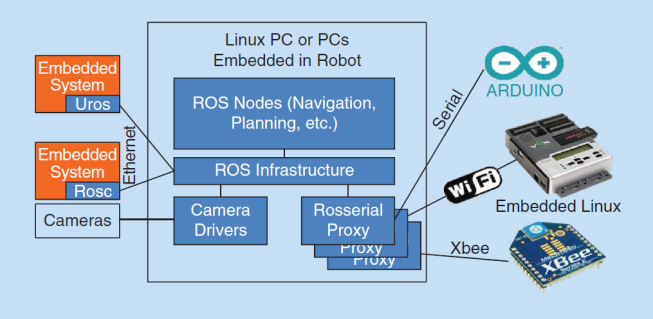

Fig. 3. ROS Example Network Diagram

Commerically available embedded systems can interface with the ROS system by adding interface processes (ROS topics) 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[7]. ROS can be implemented as a combination of an embedded PC, properietary systems with custom interfaces, and ROS Messaging. Embedded PCs provide smooth ROS integration but fail to offer hard real time support. 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. The IPC system for ROS operates on remote procedure calls and publish/subscribe protocols for messaging.

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

[4]Available at www.zeromq.org

[5]Available at www.ros.org

The ROS IPC system allows for custom messages interface with embedded systems. Rosserial, rosc, and Rosbridge are methods of passing these messages. 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 capabiliies of ROS[7].

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 designed for command and control of dynamixel 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.

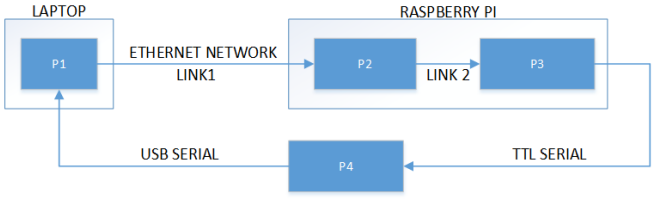

Fig. 4. Test Setup Block Diagram

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. 10000 samples were processed for each test.

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[1], 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.

IV. TEST RESULTS

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 inherenit reliability. The introduction of a wireless networks or lower quality routers may cause more of a differentiation to arise. The fastest overall system consisted of UDP network messages and ACH messages for the internal communication through the Pi. There was negligable 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 internall Raspberry Pi link in Table II.

TABLE I
RESULTS SUMMARY: NETWORK LINK

	Network: Link 1				
Item	UDP	TCP	ZMQ	ACH	ROS
Average (ms)	3.834	3.829	ZMQ	N/A	ROS
Min (ms)	3.359	3.399	ZMQ	N/A	ROS
Max (ms)	95.063	42.006	ZMQ	N/A	ROS
Std Deviation (ms)	1.441	0.768	ZMQ	N/A	ROS

TABLE II
RESULTS SUMMARY: RASPBERRY PI LINK

	Raspberry Pi: Link 2				
Item	UDP	TCP	ZMQ	ACH	ROS
Average (ms)	3.834	4.774	ZMQ	3.752	ROS
Min (ms)	3.359	4.079	ZMQ	3.219	ROS
Max (ms)	95.063	43.974	ZMQ	73.533	ROS
Std Deviation (ms)	1.441	1.160	ZMQ	1.705	ROS

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 occured despite UDP being the least reliable IPC method tested. Consistent results occured with a standard deviation of 1.441 ms and the average transit was 3.834 ms. This was the second fastest configuration tested. Figure 5 below shows a summary of collected data with outliers removed for clarity.

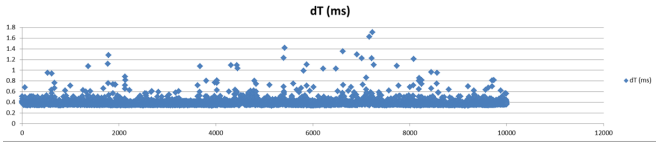

Fig. 5. UDP Control Test Results

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 negligable 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 negligable. Figure 6 below shows a summary of collected data with outliers removed for clarity.

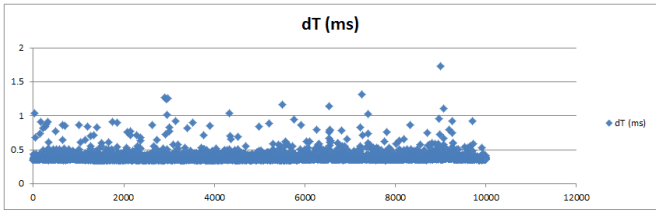

Fig. 6. TCP Link 1 Test Results

Differentiation arises when TCP is utilized for the Link 2 loopback transport. Additional latency was recorded for an averate 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 candiate for an internal IPC. Figure 7 below summarizes the Link 2 test results.

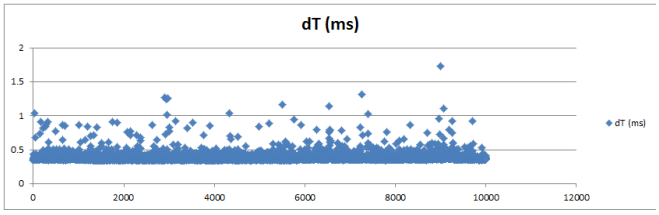

Fig. 7. TCP Link 2 Test Results

D. ZMQ Results

placeholder for zmq results

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 8 below shows a summary of collected data with outliers removed for clarity.

F. ROS Results

placeholder for ros results

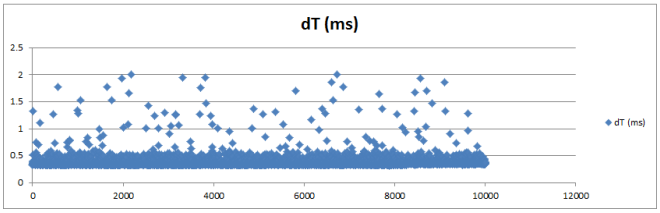

Fig. 8. ACH L2 Test Results

V. 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 recieve 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 proprogate 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

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 quantites 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 incomming messages was not an acceptable solution as processing two USB video streams at higher rates was 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 resouces of the Pi. Figure 9 shows a block diagram of the processes and communication methods for the robot.

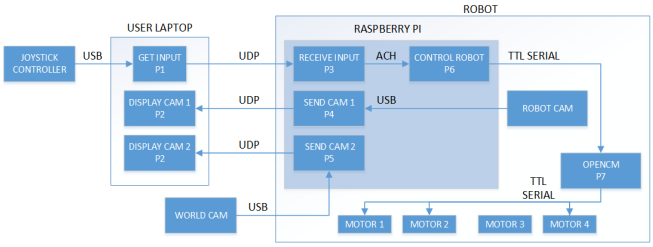

Fig. 9. Crane Robot Communication Diagram

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 candiate 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. Utilzing 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 implemeneting both UDP and ACH for the robot.

VI. CONCLUSIONS

conclusions placeholder

ACKNOWLEDGMENT

placeholder for acknowledgements

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] M. Sstrik, "mq: The theoretical foundation," 250bpm, Tech. Rep., 09 2011.

[6] P. Hintjens and M. Sustrik, "Multithreaded magic with mq," iMatrix Corporation, Tech. Rep., 09 2010.

[7] P. Bouchier, "Embedded ros [ros topics]," *Robotics Automation Magazine, IEEE*, vol. 20, no. 2, pp. 17–19, June 2013.