Interprocess Communication for Robotics Applications

Edward M. Roderick

Abstract— **Many modern robotic systems are being developed as a collection of independent processes. In order for these processes to function together, one must employ an interprocess communication (IPC) system. There exist several different prebuilt options, each with their own strengths and weaknesses. This paper presents an analysis the capabilities of four popular system (Sockets, ZeroMQ, ACH and ROS) and evaluates their performance based on latency and set of design goals in a real world application.**

I. INTRODUCTION

As modern robotic systems grow in both size and complexity, the benefits of using a multi-process approach to control the system becomes more and more apparent. The modularity that arises from each susbsystem being broken down into individual processes allows for work to be spread across large teams of designers. By defining set inputs and outputs of each process, programmers can utlize whatever programming language is most appropriate for the required task. Additionally, the overall system can be distrubted across a variety of hardware (full computers, single board computers, microcontrollers) allowing for high levels of optimization and protection from a total system crash if a single subsystem fails. In order for all of these different processes running on different pieces of hardware to communicate effectivley, one must put careful consideration to the method in which these processes communicate.

While many different Interprocess controllers (IPC) exist, this paper will focus on a few mainstream options and how they apply directly to the design constraints applied to robotics. As robotics operate in real time, the latency and data integrity of each communication step is of critical importance. For highly complex systems (humanoids, robots operating in proximity with humans, etc) with a multitude of processes running simulatenously, any disruption in communication can result in information from sensors being processed late and actuators responding to data that is no longer valid. The result can range from the robot not accomplishing its desired task to catastrohpic system failures that can put humans working in the vicinity of the robot in danger.

Each process that will be evaluated has its own set of strenghts and weaknesses. There will be not be a 'one size fits all' solution to every system and the use of each IPC must be weighed against the design metrics of the system. In order to do so effectively, this paper serves to provide a broad understanding of capabilites of Sockets, ACH, ZeroMQ and ROS and how they apply to a real world example.

II. BACKGROUND

A. ACH (Shared Memory)

POSIX IPC offers three types of messages: streams, datagrams and shared memory. For the real time robotics application, it is important to ensure that the IPC system is non head of line blocking as the most recent sensor data will take priority over previously acquired data. Streams suffers from head of line blocking and is thus not a desirable method for robotics. Datagrams utilize a data buffer that when full, loses any new data. This is particularly problematic as the robot would routinely lose important data. Shared memory is the fastest method of the POSIX suite and will provide the most current data by overwriting any old information stored. Unfortunately this method is susceptible to synchronization issues and further considerations must be taken in code which would reduce its reusability and complexity. Furthermore, for a realtime system, none of the POSIX IPC methods allow any methods of priority inheritance which is critical to allow high priority systems to access data first.

The ACH system allows the system to create multiple data channels to send data between processes. Each channel consists of two circular buffers: a data buffer and an index buffer. The two buffers are written in a channel specific POSIX shared memory file. By storing the data buffers in shared memory, the synchronization issue can be solved once for the entire channel via a mutex, allowing each reader process to poll the channel or wait for new data to be posted. This also prevents starvation and allows for priority inheritance between the real time processes.

A case study is presented [1] in which a balancing bipedal robot was developed with many different processes and ACH channels. The study details how the multiprocess approach allows for high priority processes (like balancing in this case) to stay alive in the event that other processes fail (ie computer vision). This approach also allowed processes to be distributed between smaller microcontrollers to drive servomotors to higher powered computers to process computer vision. SPIN simulations were run to benchmark the ACH system and it was shown that performance is identical to POSIX for single readers and receivers and the majority of the latency is a product of context switches.

B. Sockets: TCP and UDP

placeholder for tcp and udp backgrounds

C. ZeroMQ

ZeroMQ (MQ) is a messaging system that extends on the foundations of sockets. It provides an additional abstraction

layer over TCP by sending messages to a system topology instead of specific IP addresses. MQ prevents sending to specific IP addresses by design. The available messaging patterns is open ended, popular systems include publish/subscribe and request/reply. Only one messaging pattern is allowed for a topology and they cannot be interconnected. This is essential to providing guarantees that data will arrive at its intended location. MQ separates its stack into end to end and hop by hop layers. Unlike TCP/IP, each MQ end to end protocol has its own hop by hop protocol[2]. This allows each specific messaging pattern to have their own routing functionality. This allows bidirectional messages to 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.

With multicore processors being commonplace in todays market and the growing availability of multicore processors in commercially available embedded systems there is a need for efficiently written multithreaded code. Current methods of dealing with the issues that arise from multithreading (synchronization, starvation, priority inversion) are difficult to learn and do not scale well as the number of available cores increases. By creating code with too many locks, threads can block one another and the system can perform inconsistently depending on what system the program is running on. Writing lock free code can become susceptible to processors reordering instructions and the code again performing inconsistently. These multithreading approaches are API based and incorrect usage can often result in multiple threads blocking each other and the program essentially running as a single threaded process. ZeroMQ provides a solution to these issues by abstracting the more difficult lock free multithreading code behind its native message passing techniques and achieves simple multithreading by sending messages between threads. Being based on BSD Sockets, the code implementation is simple and intuitive for existing programmers, as well as being available for a multitude of programming languages. ZeroMQ is infinitely scalable as the number of available cores increases without any rewriting of existing code.

D. ROS

With the growing availability of cheap ARM processor systems, versions of ROS are being developed to provide a lower cost design than the traditional full-fledged Linux pc. ROS subsystems can be distributed over multiple ARM systems, allowing for more design flexibility, lower power consumption, and improved scalability. ROS allows for higher level systems to be separated from real time subsystems and other system critical functions. This allows for high priority systems to be simplified to reduce risk of failure. ROS was designed to support higher level robot functions instead of the control of individual motors/sensors/etc. Much of the integration of subsystems is left to the design as ROS does not support fieldbuses.

Three architecture styles exist for the designing to embed the ROS system: Embedded PC, Proprietary system with interface, or use of ROS messaging and APIs. The embedded PC would require all control hardware of subsystems to be fitted to the PC itself. This provides smooth integration with ROS, but fails to offer real hard time support. Entire existing robots can be managed from ROS by configuring a translation function between the proprietary system and the embedded system. This abstracts the lower level programming and allows for the existing systems to operate in real time. The third architecture is focused on the IPC system for ROS, which operates on remote procedure calls and publish/subscribe support.

The ROS IPC system allows for custom messages to be used and provides flexibility in configuring the embedded systems. Rosserial, rosc, and Rosbridge are common methods of passing these messages. Rosserial provides a proxy over a C++ client that can be ported easily to any system that supports the language, not required an OS. Rosserial provides a ROS-like API to publish, subscribe, offer and consume RPC services. Potential issues arise with the proxy becoming a bottleneck. Rosc allows for direct connection to the Ethernet and handles native ROS connections and messages. However, for embedded systems, the TCP/IP overhead may prove too overwhelming for low powered systems. Rosbridge provides dynamic socket and websocket access to the full capabilities of ROS

APPENDIX

Placeholder for appendicies

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. Sstrik, "mq: The theoretical foundation," 250bpm, Tech. Rep., 09 2011.

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

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