The Ach Library

A New Framework for Real-Time Communication

By Neil T. Dantam, Daniel M. Lofaro, Ayonga Hereid, Paul Y. Oh, Aaron D. Ames, and Mike Stilman

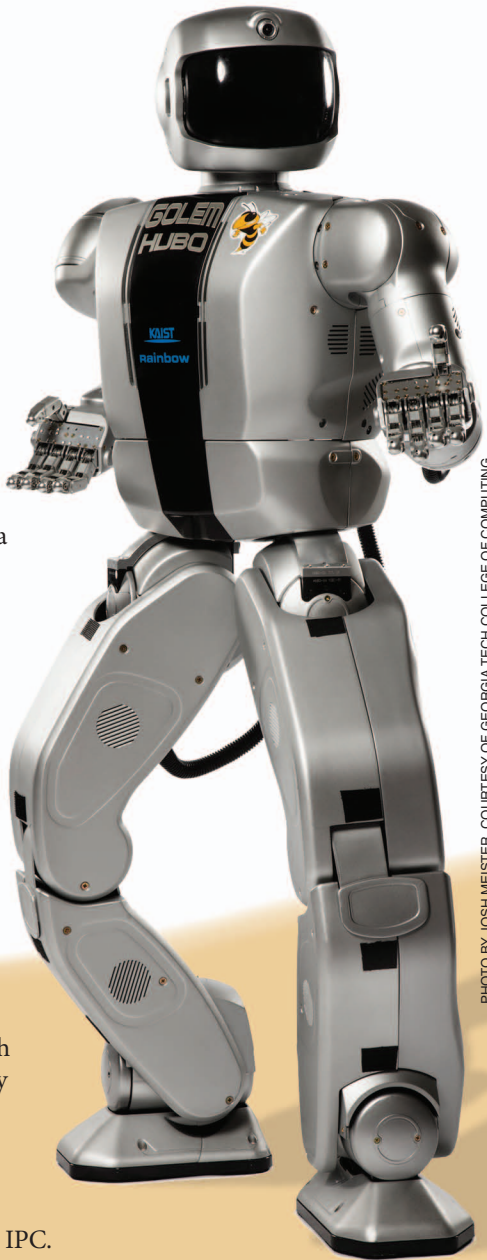

PHOTO BY JOSH MEISTER, COURTESY OF GEORGIA TECH COLLEGE OF COMPUTING

Correct real-time software is vital for robots in safety-critical roles such as service and disaster response. These systems depend on software for locomotion, navigation, manipulation, and even seemingly innocuous tasks such as safely regulating battery voltage. A multiprocess software design increases robustness by isolating errors to a single process, allowing the rest of the system to continue operation. This approach also assists with modularity and concurrency. For real-time tasks, such as dynamic balance and force control of manipulators, it is critical to communicate the latest data sample with minimum latency. There are many communication approaches intended for both general-purpose and real-time needs [9], [13], [15], [17], [19]. Typical methods focus on reliable communication or network transparency and accept a tradeoff of increased message latency or the potential to discard newer data. By focusing instead on the specific case of real-time communication on a single host, we reduce communication latency and guarantee access to the latest sample. We present a new interprocess communication (IPC) library, Ach which addresses this need, and discuss its application for real-time multiprocess control on three humanoid robots (Figure 1). (Ach is available at http://www.golems.org/projects/ach.html. The name Ach comes from the common abbreviation for the motor neurotransmitter Acetylcholine and the computer networking term ACK.)

There are several design decisions that influenced this robot software and motivated the development of the Ach library. First, to utilize decades of prior development and engineering, we implement our real-time system on top of a portable operating system interface (POSIX)-like operating system (OS). (POSIX is IEEE Standard 1003.1 for a portable OS interface. It enables software portability among supporting OSs, such as GNU/Linux, MacOSX, Solaris, and QNX. This provides us with high-quality, open-source platforms, such as GNU/Linux and a wide variety of compatible hardware and software. Second, because safety is critical for these robots, the software must be robust. Therefore, we adopt a multiple-process approach over a single-process or multithreaded application to limit the potential scope of errors [18]. This implies that the sampled data must be passed between the OS processes using some form of IPC. Since general-purpose IPC favors older data [19] (see the "Review of POSIX IPC" section), while real-time control needs the latest data, we have developed a new IPC library.

This article discusses a POSIX IPC library for the real-time control of physical processes such as robots, describes its application on three different humanoid platforms, and compares this IPC library with a variety of other communication methods. This library, called Ach, provides a message-bus or publish-subscribe communication

Digital Object Identifier 10.1109/MRA.2014.2356937
Date of publication: 15 January 2015

1070-9932/15©2015IEEE

semantics, similar to other real-time middleware and robotics frameworks [13], [15], but with the distinguishing feature of favoring newer data over old. Ach is formally verified and efficient, and it always provides access to the most recent data sample. To the best of our knowledge, these benefits are unique among existing communications software.

Review of POSIX IPC

POSIX provides a rich variety of IPC that is well suited for general-purpose information processing, but none are ideal for real-time robot control. Typically, a physical process, such as a robot, is viewed as a set of continuous, time-varying signals. To control this physical process with a digital computer, one must sample the signal at discrete time intervals and perform control calculations using the sampled value. To achieve high-performance control of a physical system, we must process the latest sample with minimum latency. This differs from the requirements of general computing systems that focus on throughput over latency and favor prior data over latter data. Thus, for robot control, it is better to favor new data over old data whereas nearly all POSIX IPC favors the old data. This problem is typically referred to as *head-of-line* (*HOL*) *blocking*. The exception to this is POSIX shared memory. However, the synchronization of shared memory is a difficult programming problem, making the typical and direct use of POSIX shared memory unfavorable for developing robust systems. Furthermore, some parts of the system, such as logging, may need to access older samples; so this also should be permitted at least on a best-effort basis. Since no existing implementation satisfied our requirements for low-latency exchange of most-recent samples, we have developed a new open-source IPC library.

The three main types of POSIX IPC are streams, datagrams, and shared memory. We review each of these types and consider why these general-purpose IPC mechanisms are not ideal for real-time robot control. Table 1 contrasts the response of each method to a full buffer, and Table 2 summarizes the pros and cons of each method. A thorough survey of POSIX IPC is provided in [19].

Streams

Stream IPC includes pipes, firstIn, firstouts (FIFOs), local-domain stream sockets, and transmission control protocol (TCP) sockets. All of these IPC mechanisms expose the file abstraction, a sequence of bytes accessed with `read` and `write`. All stream-based IPC suffers from the HOL blocking problem; we must read all of the old bytes before we see any new bytes. Furthermore, to prevent blocking of the reading or writing process, we must resort to more complicated nonblocking or asynchronous I/O.

Datagrams

Datagram Sockets

Datagram sockets perform better than streams in that they are less likely to block the sender. In addition, some types of datagram sockets can multicast packets, efficiently transmitting them to multiple receivers. However, datagram sockets give a variation on the HOL blocking problem where newer messages are simply lost if a buffer fills up. This is unacceptable since we require access to the most recent data.

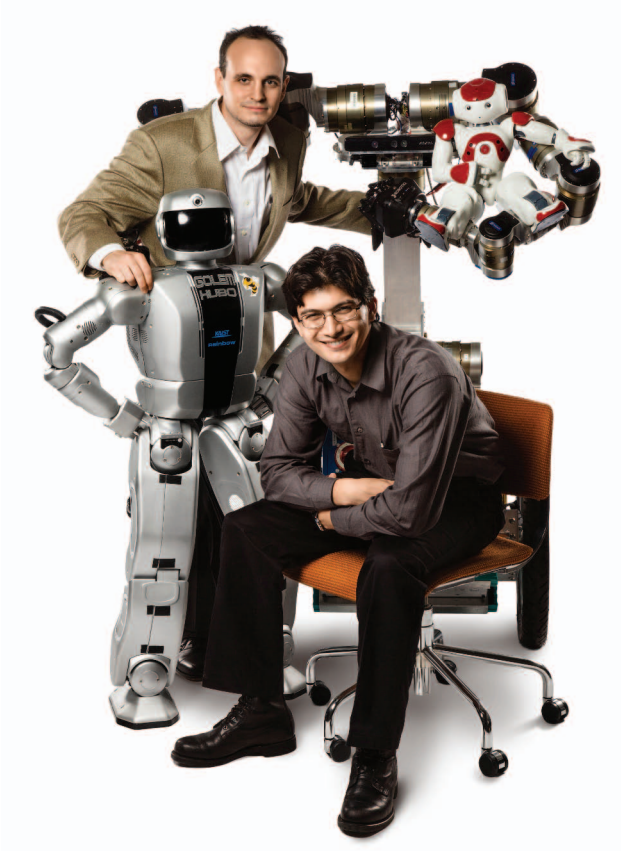

Figure 1. Hubo, Golem Krang, and NAO are existing robotic systems where Ach provides communications between the hardware drivers, perception, planning, and control algorithms. (Photo courtesy of Josh Meister and the Georgia Tech College of Computing.)

POSIX Message Queues

POSIX message queues are similar to datagram sockets and also include the feature of message priorities. The downside is that it is possible to block if the queue fills up. Consider a process that gets stuck and stops processing its message queue. When it starts again, the process must still read or flush old messages before getting the most recent sample.

Table 1. Full buffer semantics.

Method	Action on Full Buffer
Stream	Block sender, or error
Datagram	Drop newest message, or error
Message queue	Block sender, or error
Ach	Drop oldest message

Table 2. POSIX IPC summary, pros, and cons for real time.

Method	Pro	Con	Examples
Streams	Reliable, ordered	HOL blocking	Pipes, TCP, local socket
Datagrams	Multicast, does not block sender	Full buffer blocks or discards new data	UDP, local socket
Message queues	Can avoid blocking sender	Full buffer blocks or discards new data	POSIX message queues
Shared memory	Fast	Last only, synchronization issues	POSIX shared memory, mmap
Asynchronous I/O	No blocking	Immature, favors old data	POSIX asynchronous I/O
Nonblocking I/O	No blocking	Must retry, favors old data	O_NONBLOCK
Multiplexed I/O	Handles many connections	Receiver must read/discard old data	Select, poll, epoll, kqueue

Shared Memory

POSIX shared memory is very fast, and one could always have the latest data by simply overwriting a variable. However, this provides no recourse for recovering older data that may have been missed. In addition, shared memory presents synchronization issues that are notoriously difficult to solve [10], making the use of direct shared memory less suitable for safety-critical real-time control.

The data structure that Ach most closely resembles is the circular array. Circular arrays or ring buffers are common data structures in device drivers and real-time programs, and the implementation in Ach provides unique features to satisfy our requirements for a multiprocess real-time system. Typical circular buffers allow only one producer and one consumer with the view that the producer inserts data and the consumer removes it. Our robots have multiple producers and multiple consumers writing and reading a single sequence of messages. A message reader cannot remove a message because some other process may still need to read it. Because of this different design requirement, Ach uses a different data structure and algorithm to perform real-time IPC among multiple processes.

Further Considerations

Nonblocking and Asynchronous I/O Approaches

There are several approaches that allow a single process or thread to perform I/O operations across several file descriptions. Asynchronous I/O (AIO) may seem to be the most appropriate for this application. However, the current implementation under Linux is not as mature as other IPC mechanisms. Methods using select/poll/epoll/kqueue are widely used for network servers. Yet, both AIO and select-based methods mitigate the HOL problem, but do not eliminate it. Specifically, the sender will not block, but the receiver must read or flush the old data from the stream before it can see the most recent sample.

Priorities

To the best of our knowledge, none of the stream or datagram forms of IPC consider the issue of process priorities. Priorities are critical for real-time systems. When there are two readers that want the next sample, we want the real-time process, such as a motor driver, to get the data and process it before a nonreal-time process, such as a logger, does anything.

General Real-Time Robotics Middleware

In addition to the core POSIX IPC mechanisms, there are many messaging middlewares and robot software architectures. However, these are either not open source or not ideal for our multiprocess real-time domain. Many of these approaches build on an underlying POSIX IPC method, inheriting that method's strengths and weaknesses. Furthermore, our benchmark results for some of these methods (see the "Benchmarks" section) show that they impose noticeable overhead compared to the underlying kernel IPC.

Several frameworks and middleware focus on real-time control or robotics. The Orocos real-time tool kit [3] and NAOqi [1] are two architectures for robot control, but they do not meet our requirements for flexible IPC. iRobot's Aware2.0 is not open source, and Microsoft Robotics Developer Studio is not open source and does not run on POSIX systems. The robot operating system (ROS) [15] provides open-source TCP and user datagram protocol (UDP) message transports, which suffer from the aforementioned HOL blocking problem. Common object request broker architecture (CORBA) provides object-oriented remote procedure calls, an event notification service, and underlies the OpenRTM middleware. Our benchmark results (see the "Benchmarks" section) show that TAO CORBA [17], a popular implementation, gives poor messaging performance compared with the alternatives.

In contrast, the data distribution service (DDS) [13] and lightweight communications and marshalling (LCM) [9] are publish-subscribe network protocols. LCM is based on UDP multicast, which efficiently uses network bandwidth to communicate with multiple subscribers. However, UDP does drop newer packets when the receiving socket buffer is full. These protocols may be complementary to the efficient and formally verified IPC we present here.

In conclusion, none of these middlewares met our needs for an open-source, lightweight, and non-HOL-blocking IPC. However, the design of Ach facilitates integration with some of these other frameworks (see the "Speed Regulation on NAO" and "Reliable Software for the Hubo2+" sections).

The Ach IPC Library

Ach provides a message bus or publish-subscribe style of communication between multiple writers and multiple readers. A real-time system has multiple Ach channels across which individual data samples are published. Messages are sent as byte arrays, so arbitrary data may be transmitted, such as floating point vectors, text, images, and binary control messages. Each channel is implemented as two circular buffers: 1) a data buffer with variable-sized entries and 2) an index buffer with fixed-size elements indicating the offsets into the data buffer. These two circular buffers are written in a channel-specific POSIX shared memory file. Using this formulation, we solve and formally verify the synchronization problem exactly once and contain it entirely within the Ach library.

The Ach interface consists of the following procedures.

- `ach_create`: Create the shared memory region and initialize its data structures.
- `ach_open`: Open the shared memory file and initialize process local channel counters.
- `ach_put`: Insert a new message into the channel.
- `ach_get`: Receive a message from the channel.
- `ach_close`: Close the shared memory file.

Channels must be created before they can be opened. Creation may be done directly by either the reading or writing process, or it may be done via the shell command, `ach mk channel_name`, before the reader or writer start. This is analogous to the creation of FIFOs with either the `mkfifo` shell command or C function. After the channel is created, each reader or writer must open the channel before it can get or put messages.

Channel Data Structure

The core data structure of an Ach channel is a pair of circular arrays located in the POSIX shared memory file (Figure 2). It differs from typical circular buffers by permitting multiple consumers to access the same message from the channel. The data array contains variable-sized elements that store the actual message frames sent through the Ach channel. The index array contains fixed-size elements where each element contains both an offset into the data array and the length of that data element. A head offset into each array indicates the place to insert the next data and the location of the most recent message frame. Each reader maintains its own offset into the index array, indicating the last message seen by that reader. This pair of circular arrays allows readers to find the variable-size message frames based on the index array offset and the corresponding entry in the data array.

Access to the channel is synchronized using a mutex and condition variable. This allows readers to either periodically poll the channel for new data or wait on the condition variable until a writer has posted a new message. Using a read/write lock instead would have allowed only polling. In addition, synchronization using a mutex prevents starvation and enables proper priority inheritance between processes, which is important to maintaining real-time performance.

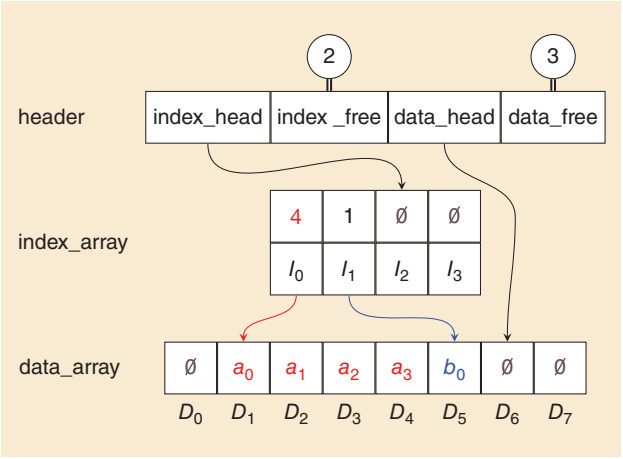

Figure 2. A logical memory structure for an Ach shared memory file. In this example, I_0 points to a 4-byte message starting at D_1, and I_1 points to a 1-byte message starting at D_5. The next inserted message will use index cell I_2 and start at D_6. There are two free index cells and three free data bytes. Both arrays are circular and wrap around when the end is reached.

Core Procedures

Two procedures compose the core of Ach: `ach_put` and `ach_get`. Detailed pseudocode is provided in [6].

ach_put

The procedure `ach_put` inserts new messages into the channel. It is analogous to `write`, `sendmsg`, and `mq_send`. The procedure is given a pointer to the shared memory region for the channel and a byte array containing the message to post. There are four broad steps to the procedure.

1) Get an index entry. If there is at least one free index entry, use it. Otherwise, clear the oldest index entry and its corresponding message in the data array.
2) Make room in the data array. If there is enough room already, continue. Otherwise, repeatedly free the oldest message until there is enough room.
3) Copy the message into data array.
4) Update the offset and free counts in the channel structure.

ach_get

The procedure `ach_get` receives a message from the channel. It is analogous to `read`, `recvmsg`, and `mq_receive`. The procedure takes a pointer to the shared memory region, a storage buffer to copy the message to, the last message sequence number received, the next index offset to check for a message, and option flags indicating whether to block waiting for a new message and whether to return the newest message bypassing any older unseen messages. There are four broad steps to the procedure.

1) If we are to wait for a new message and there is no new message, then wait. Otherwise, if there are no new messages, return a status code indicating this fact.
2) Find the index entry to use. If we are to return the newest message, use that entry. Otherwise, if the next entry we expected to use contains the next sequence number we expect to see, use that entry. Otherwise, use the oldest entry.

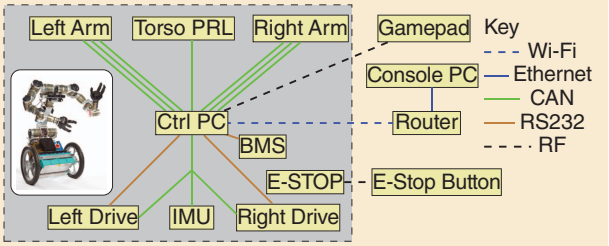

Figure 3. A block diagram of the electronic components on Golem Krang. The blocks inside the dashed line are onboard, and the blocks outside are offboard. Control software runs on the Pentium-M control PC under Ubuntu Linux, which communicates over eight CAN buses to the embedded hardware. The arms are Schunk light weight arm, 3 (LWA3s) with ATI wrist force-torque sensors and Robotiq adaptive grippers. The torso is actuated using three Schunk PRL motor modules. The wheels are controlled using AMC servo drives. The battery management system (BMS) monitors the lithium cells. (Photo courtesy of Josh Meister and the Georgia Tech College of Computing.)

3) According to the offset and size from the selected index entry, copy the message from the data array into the provided storage buffer.
4) Update the sequence number count and next index entry offset for this receiver.

Case Studies

Dynamic Balance on Golem Krang

Golem Krang (Figure 3) is a dynamically balancing, bimanual mobile manipulator designed and built at the Georgia Tech Humanoid Robotics Lab [20]. All of the real-time control for Krang is implemented through the Ach IPC library. This approach has improved software robustness and modularity, minimizing system failures and allowing code reuse, both within Krang and with other projects [7] sharing the same hardware components. The software for Krang is implemented as a collection of processes communicating over Ach channels (Figure 4). In this design, providing a separate state Ach channel for each hardware device ensures that the current state of the robot can always be accessed through the newest messages in each of these channels. In addition, splitting the control into separate `balanced`, for stable balancing, and `controlled`, for arm control, processes promotes robustness by isolating the highly critical balance control from other faults. This collection of driver and controller daemons communicating over Ach channels implements the real-time, kilohertz control loop for Golem Krang.

This design provides several advantages for control on Krang. The low overhead and suitable semantics of Ach communication permits real-time control under Linux using multiple processes. In several cases, Krang contains multiple identical hardware devices. The message-passing, multiprocess design aids code reuse by allowing access to duplicated devices with multiple instances of the same daemon binary—two instances of the `ftd` daemon for the F/T sensors, two instances of the `robotiqd` daemon for the grippers, and three instances of the `pciod` daemon for two arms and torso.

The relative independence of each running process makes this system robust to failures in noncritical components. For example, an electrical failure in a waist motor may stall the `w_pciod` process, but, without any additional code, the `balanced` controller and `amciod` driver daemons continue running independently, ensuring that the robot does

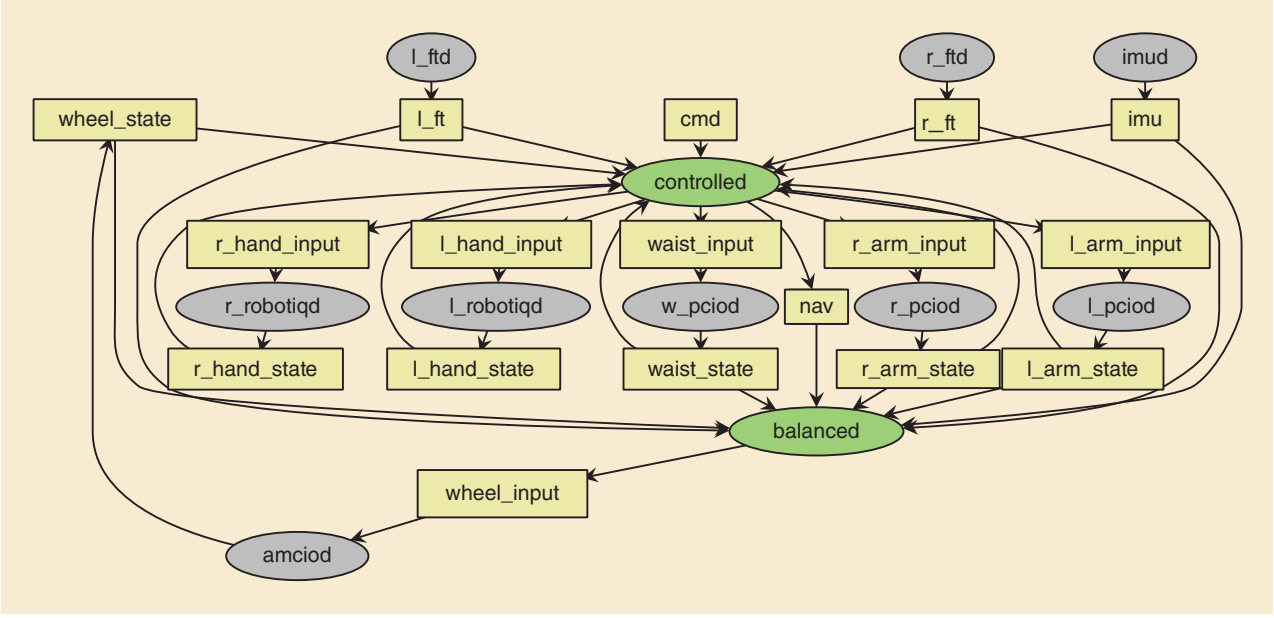

Figure 4. A block diagram of the primary software components on Golem Krang. The gray ovals are user-space driver processes, the green ovals are controller processes, and the rectangles are Ach channels. Each hardware device, such as the IMU or LWA3, is managed by a separate driver process. Each driver process sends state messages, such as positions or forces, over a separate state channel. Devices that take input, such as the reference velocity, have a separate input channel.

not fall. Thus, Ach helps enhance the safety of this potentially dangerous robot.

Speed Regulation on NAO

The Aldebaran NAO is a 0.5-m, 5-kg bipedal robot with 25 degrees of freedom (DOF). It contains an onboard Intel Atom PC running a GNU/Linux distribution with the NAOqi framework to control the robot. The user code is loaded into the NAOqi process as dynamic library modules. We used Ach to implement human-inspired control [14] on the NAO [5]. The human-inspired control approach achieves provably stable, human-like walking on robots by identifying key parameters in human gaits and transferring these to the robot through an optimization process. To implement this approach, real-time control software to produce the desired joint angles must run on the NAO's internal computer.

The NAOqi framework provides an interface to the robot's hardware; however, it presents some specific challenges for application development and for the implementation of human-inspired control in particular. NAOqi is slow and memory-intensive, consuming at idle 15% of the available CPU time and 20% of the available memory. In addition, real-time user code must run as a callback function, which is awkward for the desired controller implementation. Using Ach to move the controller to a separate process improves the implementation.

A multiprocess software design (Figure 5) addresses these challenges with NAOqi and enhances the robustness and efficiency of human-inspired control on the NAO. Each process runs independently; so an error in a noncritical process, such as `logger/debugger`, cannot affect other processes, eliminating a potential failure. The user processes can be stopped and started within only a few seconds. In contrast, NAOqi takes about 15 s to start. The independence of processes means that NAOqi need not be restarted so long as `libamber` is unchanged. Since `libamber` is a minimal module, only interfacing with the Ach channels and accessing the NAO's hardware, it can be reused unmodified for different applications on the NAO. Different projects can run different controller processes, using Ach and `libamber` to access the NAO's hardware, all without restarting the NAOqi process. In addition, using standard debugging tools, such as GNU debugger (GDB), is much easier since the user code can be executed within the debugger independently of the NAOqi framework. Thus, converting the NAO's control software to a multiprocess design simplified development and improved reliability.

Reliable Software for the Hubo2+

The Hubo2+ is a 1.3-m-tall, 42-kg full-size humanoid robot, produced by the Korean Advanced Institute of Science and Technology (KAIST) and spinoff company Rainbow Inc. [4]. It has 38 DOF: six per arm and leg, five per hand, three in the neck, and one in the waist. The sensors include three-axis force-torque sensors in the wrists and ankles, accelerometers in the feet, and an inertial measurement unit (IMU). The sensors and embedded motor controllers are connected via a controller area network to a pair of Intel Atom PC104+ computers.

Hubo-Ach (available under permissive license, http://github.com/hubo/hubo-ach) is an Ach-based interface to Hubo's sensors and motor controllers [12]. This provides a conventional GNU/Linux programming environment, with the variety of tools available therein, for developing applications on the Hubo. It also links the embedded electronics and real-time control to popular frameworks for robotics software: ROS [15], OpenRAVE, and MATLAB.

Reliability is a critical issue for software on the Hubo. As a bipedal robot, Hubo must constantly maintain a dynamic balance; if the software fails, it will fall and break. A multiprocess software design improves Hubo's reliability by isolating the critical balance code from other noncritical functions, such as control of the neck or arms. For the high-speed, low-latency communications and priority access to latest sensor feedback, Ach provides the underlying IPC.

Hubo-Ach handles controller area network (CAN) bus communication between the PC and embedded electronics. Because the motor controllers synchronize to the control period in a phase lock loop, the single `hubo-daemon` process runs at a fixed control rate. The embedded controllers lock to this rate and linearly interpolate between the commanded positions, providing smoother trajectories in the face of limited communication bandwidth. This communication process also avoids bus saturation; with a CAN bandwidth of 1 Mbit/s and a 200-Hz control rate, `hubo-daemon` utilizes 78% of the bus. `Hubo-daemon` receives position targets

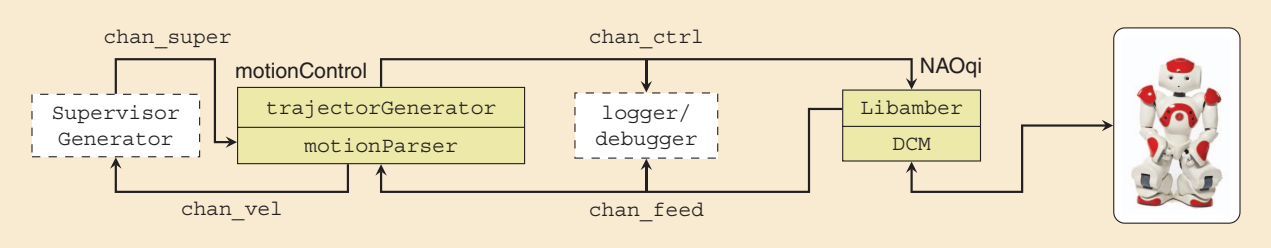

Figure 5. A block diagram of the primary software components on NAO. The solid-lined blocks are real-time processes, and the dashed-lined blocks are nonreal-time processes. `NAOqi` loads the `libamber` module to communicate over Ach channels. The `motionControl` process performs feedback control, while the `logger/debugger` process records data from the Ach channels. The `supervisor generator` process performs high-level policy generation for speed control. (Photo courtesy of Josh Meister and the Georgia Tech College of Computing.)

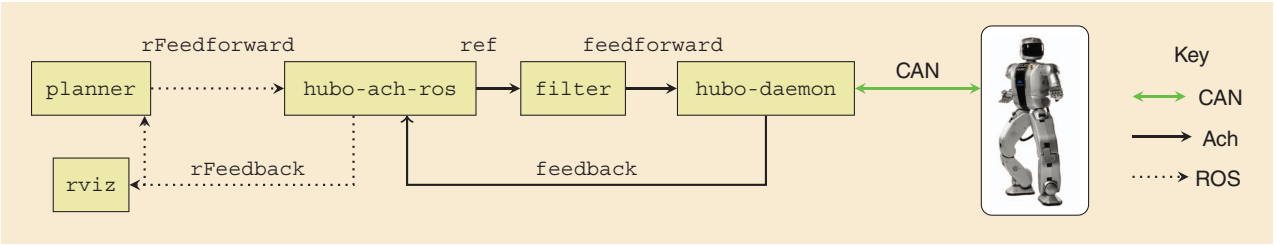

Figure 6. A block diagram of the feedback loop integrating Hubo-Ach and ROS. The `planner` process computes trajectories, and the `rviz` process displays a three-dimensional model of Hubo's current state. The `Hubo-ach-ros` process bridges the Ach channels with ROS topics. The `filter` process smooths trajectories to reduce jerk. `Hubo-daemon` communicates with the embedded motor controllers. (Photo courtesy of Josh Meister and the Georgia Tech College of Computing.)

from a `feedforward` channel and publishes sensor data to the `feedback` channel, providing the direct software interface to the embedded electronics. Figure 6 shows an example control loop integrating Hubo-Ach and ROS.

Hubo-Ach is being used for numerous projects at several research labs. Users include groups at the Massachusetts Institute of Technology, Worcester Polytechnic Institute (WPI), The Ohio State University, Purdue, Swarthmore College, Georgia Tech, and Drexel University. These projects primarily revolve around the Defense Advanced Research Projects Agency (DARPA) robotics challenge (DRC) (http://www.theroboticschallenge.org/) team DRC-Hubo (http://drc-hubo.com/). The DRC includes rough terrain walking, ladder climbing, valve turning, vehicle ingress/egress, and more. Figure 7 shows the Hubo using the Hubo-Ach system to turn a valve.

Hubo-Ach helps the development of reliable, real-time applications on the Hubo. Separating software modules into different processes increases system reliability. A failed process can be independently restarted, minimizing the chance of damage to the robot. In addition, the controllers can run at fast rates because Ach provides high-speed, low-latency communication with `hubo-daemon`. Hubo-Ach provides a C API callable from high-level programming languages, and it integrates with popular platforms for robot software, such as ROS and MATLAB, providing additional development flexibility. Hubo-Ach is a validated and effective interface between the mechatronics and the software control algorithms of the Hubo full-size humanoid robot.

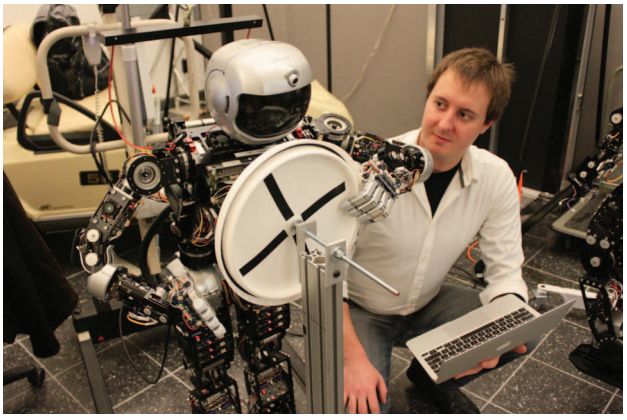

Figure 7. Hubo (left) turning a valve via Hubo-Ach alongside Daniel M. Lofaro (right). Valve turning was developed in conjunction with Dmitry Berenson at WPI for the DRC. (Photo courtesy of Daniel M. Lofaro and Kenneth Chaney.)

Performance and Discussion

Formal Verification

We used the SPIN model checker [8] to formally verify Ach. Formal verification is a method to enhance the reliability of software by first modeling the operation of that software and then checking that the model adheres to a specification for performance. SPIN models concurrent programs using the Promela language. Then, it enumerates all possible world states of that model and ensures that each state satisfies the given specification. This can detect errors that are difficult to find in testing. Because process scheduling is nondeterministic, testing may not reveal errors due to concurrent access, which could later manifest in the field. However, because model checking enumerates all possible process interleavings, it is guaranteed to detect concurrency errors in the model.

We verified the `ach_put` and `ach_get` procedures using SPIN. Our model for Ach checks the consistency of channel data structures, ensures proper transmission of message data, and verifies freedom from deadlock. Model checking verifies these properties for all possible interleavings of `ach_put` and `ach_get`, which would be practically impossible to achieve through testing alone. By modeling the behavior of Ach in Promela and verifying its performance with SPIN, we eliminated errors in the returned status codes and simplified our implementation, improving the robustness and simplicity of Ach.

Benchmarks

We provide benchmark results of message latency for Ach and a variety of other kernel IPC methods as well as the LCM, ROS, and TAO CORBA middleware (benchmark code available at http://github.com/ndantam/ipcbench). Latency is often more critical than bandwidth for real-time control as the amount of data per sample is generally small, e.g., state and reference values for several joint axes. Consequently, the actual time to copy the data is negligible compared with other sources of overhead such as process scheduling. The benchmark application performs the following steps.

1) Initialize communication structures.
2) `Fork` sending and receiving processes.

3) Sender: Post time-stamped messages at the desired frequency.
4) Receivers: Receive messages and record latency of each messaged based on the time stamp.

We ran the benchmarks under two kernels: Linux PREEMPT_RT and Xenomai. PREEMPT_RT is a patch to the Linux kernel that reduces latency by making the kernel fully preemptible. Any Linux application can request real-time priority. Xenomai runs the real-time Adeos hypervisor alongside a standard Linux kernel. Real-time applications communicate through Adeos via an API skin, such as RTDM, µITRON, or POSIX; these applications are not binary compatible with Linux applications, although the POSIX skin is largely source compatible.

Figure 8 shows the results of the benchmarks run on an Intel Xeon 1270v2 under both Linux PREEMPT_RT and Xenomai's POSIX skin. We used Linux 3.4.18 PREEMPT_RT, Xenomai 2.6.2.1/RTnet 0.9.13/Linux 3.2.21 Ach 1.2.0, LCM 1.0.0, ROSCPP 1.9.50, and TAO 2.2.1 with ACE 6.2.1. (While we were able to test RTnet's loopback performance, the RTnet driver for our Ethernet card caused a kernel panic. Similar stability issues with Xenomai were noted in [2].) We benchmarked one and two receivers, corresponding to the communication cases in the "Case Studies" section. Each test lasted for 600 s, giving approximately 6×10^5 data points per receiver. These results show that for the use cases in the "Case Studies" section, where communication is between a small number of processes, Ach offers a good balance of performance in addition to its unique latest-message-favored semantics.

As an approximate measure of programmer effort required for each of these methods, Figure 9 summarizes the source lines of code (measured using http://www.dwheeler.com/sloccount/) for the method-specific code in the benchmark program. The counts include message and interface declarations and exclude generated code. To give a more fair

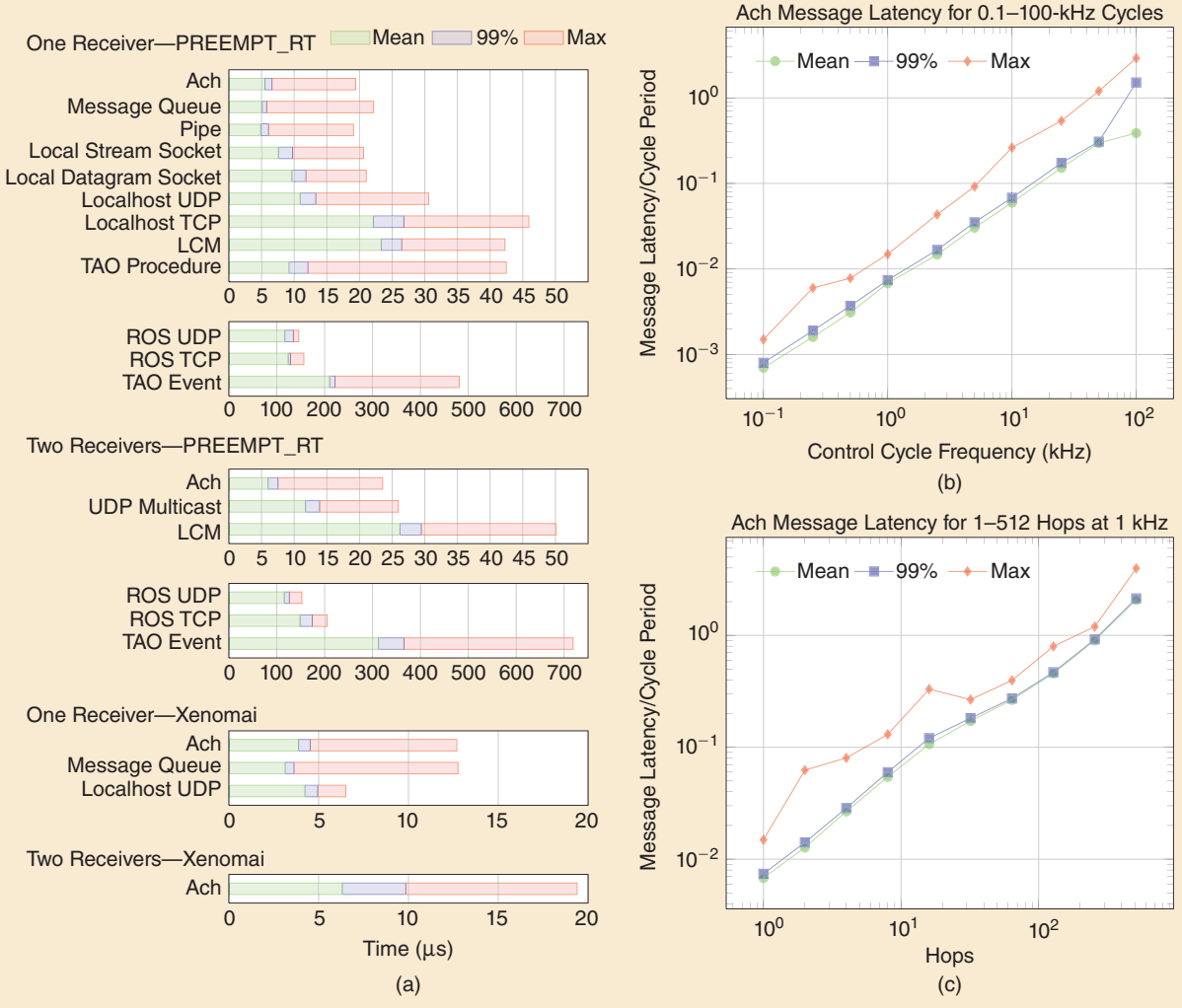

Figure 8. The message latency for Ach, POSIX IPC, and common middleware. "Mean" is the average over all messages, "99%" is the latency that 99% of the messages beat, and "Max" is the maximum recorded latency. (b) and (c) show the limits of Ach performance on Linux PREEMPT RT, with a $10^0 = 1$ latency ratio indicating latency of an entire cycle. (b) shows the latency ratio for various control cycle frequencies. The discontinuity above 50 kHz occurs due to transmission time exceeding the cycle period and consequent missed messages. (c) shows the latency ratio resulting from passing the message through multiple intermediate processes.

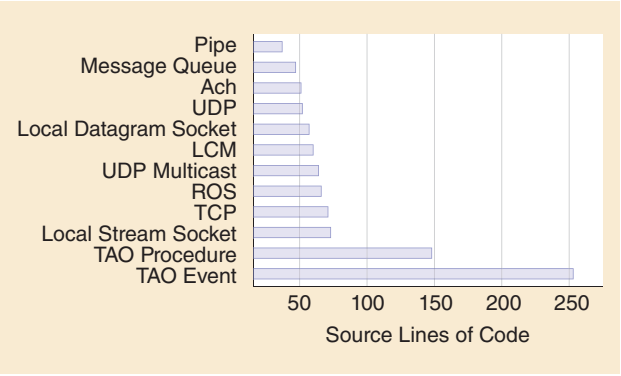

Figure 9. Source lines of code for each benchmarked method.

comparison, we attempted to consistently check errors across all methods. Most methods have similar line counts, with sockets usually requiring a small amount of extra code to set up the connection. The pipe code is especially short because the file descriptors are passed through `fork`; this would not work for unrelated processes. The networked methods in the test do not consider security, which would necessarily increase the complexity of networked real-world applications, while Ach, message queues, and local domain sockets implicitly control local data access based on user IDs. TAO CORBA stands out with several times more code than the other methods. It is also notable that the higher-level frameworks in this test did not result in significantly shorter communication code than direct use of kernel IPC.

Discussion

The performance limits illustrated in Figure 8 indicate the potential applicability of Ach. The latency ratio compared with the hop count is particularly important because it bounds the minimum granularity at which the control system can be divided between processes. On our test platform, significant overhead, i.e., exceeding 25% of the 1-kHz control cycle, is incurred when information must flow serially through approximately 32 processes. This cost is important to consider when dividing computation among different processes. For higher control rates, our test platform reaches 25% messaging overhead at approximately 10 kHz. For the robots in the "Case Studies" section, the embedded components, particularly the CAN buses, effectively limit the control rates to 1 kHz or lower; Ach is not the bottleneck for these systems. However, implementing systems that do require 10 kHz or greater control rates would be difficult with Ach on Linux PREEMPT_RT. These performance considerations show the range of systems for which this software design approach is suitable.

In addition to performance considerations, it is also critical to note the semantic differences between communication methods. The primary unique feature of Ach is that newer messages always supersede older messages. The other message-passing methods give priority to older data, and will block or drop newer messages when buffers are full. CORBA also differs from the other methods by exposing a remote procedure call rather than a message-passing interface; however, the CORBA event service layers message passing on top of the remote procedure call. Selecting the appropriate communication semantics for an application simplifies implementation.

Some of the benchmarked methods also operate transparently across networks. This can simplify distributing an application across multiple machines, though this process is not seamless due to differences between local and network communication [16]. Processes on a single host can access a unified physical memory that provides high bandwidth and assumed perfect reliability; still, care must be taken to ensure memory consistency between asynchronously executing processes. In contrast, real-time communication across a network need not worry about memory consistency, but must address issues such as limited bandwidth, packet loss, collisions, clock skew, and security. With Ach, we have focused on efficient, latest-message-favored communication between a few processes on a single host. We intend the Ach double-circular-buffer implementation to be complementary to, and its message-passing interface compatible with, networked communication. This meets the communication requirements for systems such as those in the "Case Studies" section.

An important consideration in the design of Ach is the idea of mechanism, not policy [18]. Ach provides a mechanism to move bytes between processes and to notify callers of errors. It does not specify a policy for serializing-arbitrary data structures or handling all types of errors. Such policies are application dependent and, even within our own research groups, have changed across different applications and over time. This separation of policy from mechanism is important for flexibility.

This flexibility is helpful when integrating with other communication methods or frameworks. To integrate with ROS on Hubo (see the "Reliable Software for the Hubo2+" section), we created a separate process to translate between real-time Ach messages and nonreal-time ROS messages. This approach is straightforward since both Ach and ROS expose a publish/subscribe message passing interface. On the other hand, NAOqi exposes a callback interface. Still, we can integrate with this (see the "Speed Regulation on NAO" section) by relaying Ach messages within the NAOqi callback. In general, integrating Ach with other frameworks requires serializing framework data structures to send over an Ach channel. However, since Ach works with raw byte arrays, it is possible to directly use existing serialization methods such as XDR, Boost.Serialization, ROS Genmsg, Google Protocol Buffers, or contiguous C structures.

Achieving real-time bounds on general-purpose computing systems presents an overall challenge. The Linux PREEMPT_RT patch seamlessly runs Linux applications with significantly reduced latency compared with vanilla Linux, and work is ongoing to integrate it into the mainline kernel. However, it is far from providing formally guaranteed bounds on latency. Xenomai typically offers better latency than PREEMPT_RT [2], but it is less polished and its dual kernel approach complicates development. There are many other OSs with a dedicated focus on real time, e.g., VxWorks, QNX, and TRON. In addition to

OS selection, the underlying hardware can present challenges. CPU frequency scaling, which reduces power usage, can significantly increase latency. On x86/AMD64 processors, system management interrupts (SMI) (http://www.intel.com/design/processor/manuals/253669.pdf) preempt all software, including the OS, potentially leading to latencies of hundreds of microseconds. A fundamental challenge is that general-purpose computation considers time, not in terms of correctness, but only as a quality metric—faster is better—whereas real-time computation depends on timing for correctness [11]. These issues are important in the overall real-time system design.

Conclusions

Ach is a new IPC method for real-time communication, demonstrated on multiple robotic systems. Compared with standard POSIX IPC and the communication mechanisms employed by popular robotics middleware [1], [15], [19], Ach's unique message-passing semantics always allow the latest data sample to be read. It provides good performance for typical communication needs on humanoid robots. The algorithms and data structures are formally verified. Ach has been validated in the core of a variety of robot control applications and has aided the development of efficient and reliable control software for our robots, Golem Krang, Hubo, and NAO.

The Ach library and sample code can be downloaded at http://www.golems.org/projects/ach.html. By providing this open-source IPC library to the robotics community, we hope that it will be a useful tool to expedite the development of new robust systems.

Acknowledgments

This work is dedicated to the memory of M. Stilman, whose perpetual enthusiasm will always inspire us. We would like to thank M. Grey and M. Zucker for their major development of Hubo-Ach.

References

[1] C. E. Agüero, J. M. Cañas, F. Martín, and E. Perdices, "Behavior-based iterative component architecture for soccer applications with the Nao humanoid," in *Proc. 5th Workshop Humanoids Soccer Robots*, Nashville, TN, 2010, pp. 29–34.

[2] J. H. Brown and B. Martin. (2010). How fast is fast enough? Choosing between Xenomai and Linux for real-time applications. Invariant Systems Inc., Cambridge, MA. Tech. Rep. [Online]. Available: https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf

[3] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the Orocos project," in *Proc. IEEE Int. Conf. Robotics Automation*, 2003, vol. 2, pp. 2766–2771.

[4] B.-K. Cho, S.-S. Park, and J.-H. Oh, "Controllers for running in the humanoid robot," in *Proc. 9th IEEE-RAS Int. Conf. Humanoid Robots*, Paris, France, 2009, pp. 385–390.

[5] N. Dantam, A. Hereid, A. Ames, and M. Stilman, "Correct software synthesis for stable speed-controlled robotic walking," in *Proc. Robotics: Science Systems*, June 2013.

[6] N. Dantam and M. Stilman, "Robust and efficient communication for real-time multi-process robot software," in *Proc. 12th IEEE-RAS Int. Conf. Humanoid Robots*, Osaka, Japan, 2012, pp. 316–322.

[7] N. Dantam and M. Stilman, "The motion grammar: Analysis of a linguistic method for robot control," *IEEE/RAS Trans. Robot.*, vol. 29, no. 3, pp. 704–718, 2013.

[8] G. Holtzman, *The spin model checker*. Reading, MA: Addison-Wesley, 2004.

[9] A. S. Huang, E. Olson, and D. C. Moore, "LCM: Lightweight communications and marshalling," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots Systems*, Taipei, Taiwan, 2010, pp. 4057–4062.

[10] T. Kelly, Y. Wang, S. Lafortune, and S. Mahlke, "Eliminating concurrency bugs with control engineering," *Computer*, vol. 42, no. 12, pp. 52–60, 2009.

[11] E. A. Lee, "Computing needs time," *Commun. ACM*, vol. 52, no. 5, pp. 70–79, May 2009.

[12] D. Lofaro, "Unified algorithmic framework for high degree of freedom complex systems and humanoid robots," Ph.D. dissertation, Dept. Elect. Eng., Drexel Univ., Philadelphia, PA, May 2013.

[13] (2007, Jan.). *Data Distribution Service for Real-time Systems*. Object Management Group Inc., Needham, MA. [Online]. Available: http://www.omg.org/spec/DDS/1.2/

[14] M. J. Powell, A. Hereid, and A. D. Ames, "Speed regulation in 3D robotic walking through motion transitions between human-inspired partial hybrid zero dynamics," in *Proc. IEEE Int. Conf. Robotics Automation*, Karlsruhe, Baden-Württemberg, 2013, pp. 4803–4810.

[15] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: An open-source robot operating system," in *Proc. IEEE Int. Conf. Robotics Automation, Workshop Open Source Robotics*, 2009.

[16] A. Rotem-Gal-Oz. (2006). Fallacies of distributed computing explained. Sun Microsystems Inc., Santa Clara, CA. Tech. Rep. [Online]. Available: http://www.rgoarchitects.com/Files/fallacies.pdf

[17] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time object request broker," *Comput. Commun.*, vol. 21, no. 4, pp. 294–324, 1998.

[18] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Hoboken, NJ: Wiley, 2009.

[19] W. R. Stevens and S. A. Rago, *Advanced Programming in the UNIX Environment*, 2nd ed. Reading, MA: Addison-Wesley, 2005.

[20] M. Stilman, J. Olson, and W. Gloss, "Golem Krang: Dynamically stable humanoid robot for mobile manipulation," in *Proc. IEEE Int. Conf. Robotics Automation*, Anchorage, AK, 2010, pp. 3304–3309.

Neil T. Dantam, Department of Computer Science, Rice University, Houston, Texas, United States. E-mail: ntd@rice.edu.

Daniel M. Lofaro, Electrical and Computer Engineering, George Mason University, Fairfax, Virginia, United States. E-mail: dlofaro@gmu.edu.

Ayonga Hereid, Mechanical Engineering, Texas A&M University, College Station, United States. E-mail: ayonga@tamu.edu.

Paul Y. Oh, Department of Mechanical Engineering, University of Nevada, Las Vegas, United States. E-mail: paul.oh@unlv.edu.

Aaron D. Ames, Mechanical Engineering, Texas A&M University, College Station, United States. E-mail: aames@tamu.edu.

Mike Stilman, Institute for Robotics and Intelligent Machines, Georgia Institute of Technology, Atlanta, United States.