frontiers in Robotics and AI

ORIGINAL RESEARCH
published: 08 March 2016
doi: 10.3389/frobt.2016.00006

Unix Philosophy and the Real World: Control Software for Humanoid Robots

Neil T. Dantam[1], Kim Bøndergaard[2], Mattias A. Johansson[3], Tobias Furuholm[3] and Lydia E. Kavraki[1]**

[1] Department of Computer Science, Rice University, Houston, TX, USA, [2] Prevas A/S, Aarhus, Denmark, [3] Rocktec Division, Altas Copco Rock Drills AB, Örebro, Sweden

Robot software combines the challenges of general purpose and real-time software, requiring complex logic and bounded resource use. Physical safety, particularly for dynamic systems such as humanoid robots, depends on correct software. General purpose computation has converged on unix-like operating systems – standardized as POSIX, the Portable Operating System Interface – for devices from cellular phones to supercomputers. The modular, multi-process design typical of POSIX applications is effective for building complex and reliable software. Absent from POSIX, however, is an interproccess communication mechanism that prioritizes newer data as typically desired for control of physical systems. We address this need in the Ach communication library which provides suitable semantics and performance for real-time robot control. Although initially designed for humanoid robots, Ach has broader applicability to complex mechatronic devices – humanoid and otherwise – that require real-time coupling of sensors, control, planning, and actuation. The initial user space implementation of Ach was limited in the ability to receive data from multiple sources. We remove this limitation by implementing Ach as a Linux kernel module, enabling Ach's high performance and latest-message-favored semantics within conventional POSIX communication pipelines. We discuss how these POSIX interfaces and design principles apply to robot software, and we present a case study using the Ach kernel module for communication on the Baxter robot.

Keywords: real-time software, middleware, robot programing, humanoid robots, software engineering

OPEN ACCESS

Edited by:
Lorenzo Natale,
Istituto Italiano di Tecnologia, Italy

Reviewed by:
Ali Paikan,
Istituto Italiano di Tecnologia, Italy
Torbjorn Semb Dahl,
Plymouth University, UK

****Correspondence:***
Neil T. Dantam
ntd@rice.edu;
Lydia E. Kavraki
kavraki@rice.edu

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *13 October 2015*
Accepted: *12 February 2016*
Published: *08 March 2016*

Citation:
Dantam NT, Bøndergaard K, Johansson MA, Furuholm T and Kavraki LE (2016) Unix Philosophy and the Real World: Control Software for Humanoid Robots. Front. Robot. AI 3:6. doi: 10.3389/frobt.2016.00006

1. INTRODUCTION

Humanoid robot software presents a broad set of requirements. Humanoids have physical dynamics, requiring fast, real-time software. Humanoids have many sensors and actuators, requiring high performance network code. Humanoids, ideally, operate autonomously, requiring complex application logic. Satisfying these three requirements is a challenging software design and development problem. Fortunately, there are existing solutions to many of these challenges. Unix-like operating systems have served over many decades as the foundation for developing complex software. The standards and design principles learned developing these operating systems and applications provide many lessons and tools for humanoids and other robots.

Humanoid robotics would benefit by building on the significant design and engineering effort employed in unix development.[1] These operating systems are codified in POSIX, the IEEE standard for a Portable Operating System Interface (POSIX, 2008). POSIX provides a vendor neutral interface for systems programing, and there are numerous high-quality implementations that run on all major computer architectures. However, POSIX is more than just a standard for accessing filesystems and networks. Many POSIX applications follow a common design approach based on composing multiple, independent, modular processes. The multiprocess design promotes rapid development of robust and flexible software by isolating errors to a single process and enabling composition of existing tools to address new requirements (Raymond, 2003; Tanenbaum and Bos, 2014), a lesson already largely adopted by the robotics community with frameworks that often, though not always, compose applications from multiple processes (Bruyninckx et al., 2003; Brugali and Scandurra, 2009; Quigley et al., 2009). Unix-like systems are also pervasive in network-intensive applications, leading to powerful communication capabilities. Moreover, the widespread use and long history of unix has bred a variety of tools and conventions to aid system integration – a major challenge for robotics – by producing and configuring software that is flexible and portable. These features and corresponding design approaches present in Unix-like systems address many, though not all, the needs of humanoid robot software.

While Unix-like operating systems have been phenomenally successful for general purpose computing, they are less prevalent in real-time control of physical processes. 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 which focus on throughput over latency and favor older data over newer data. While nearly all POSIX communication favors the older data, in robot control, the newest data are critical. However, 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. In this paper, we address the need for real-time communication within the larger context of POSIX programing. *We demonstrate a Linux kernel module for high performance, real-time communication, and discuss its use in the application of POSIX programing practice to humanoid robots.*

The Ach interprocess communication library provides fast communication that favors latest message data as typically desired for real-time control of physical systems. Ach is not a new framework that discards or duplicates the existing and significant tools for systems programing. Instead, Ach is a mechanism that integrates and builds upon the vast useful features of the POSIX and Linux ecosystem. In previous work, we presented an implementation of the Ach data structure in the POSIX user space (Dantam and Stilman, 2012; Dantam et al., 2015). User space Ach was limited in the ability to receive data from multiple sources. Now, we present an implementation of Ach as a Linux kernel module. Kernel space Ach enables applications to efficiently receive data from many sources, a crucial feature for mechatronic systems such as humanoid robots which contain many sensors, actuators, and software modules. The Ach Linux kernel module presents the conventional file descriptor interface used for communication in POSIX, enabling direct integration into existing communication systems and frameworks.

Though this work was initially developed for humanoid robots, it is broadly applicable to other complex mechantronic systems such as robot manipulators and intelligent vehicles. These evolving technologies all present similar requirements for complex software with real-time performance. The unix philosophy is effective for building complex software systems, and the Ach library grounds this approach to real-time, physical control.

2. LEARNING FROM UNIX

Humanoid robotics can learn from of the POSIX programing community. Important and challenging issues for humanoid robot software, such as high-performance communication, real-time memory allocation, and software integration, are largely addressed by existing techniques and standards. The humanoid robotics community would benefit by building on this work.

2.1. Communication and Scalability

Humanoid robotics can benefit from the strong communication capabilities of unix-like operating systems. Historically, Unix and the Internet developed in concert (Quarterman et al., 1985). POSIX provides a variety of communication and networking approaches, which largely address the performance and scalability needs of humanoid robot software. We summarize communication with many other nodes in Section 2.1.1 and service lookup in Section 2.1.2. Later, we address the unique needs of humanoid robots with the Ach library in Section 3, building on the capabilities offered by POSIX.

2.1.1. Multiplexing Approaches

Both general network servers and humanoid robots must communicate with a large number of other devices, be they network clients or hardware sensors and actuators. There are several techniques to communicate with multiple different nodes, each having trade-offs in implementation complexity and computational efficiency.

2.1.1.1. Fixed Interval Loop

A simple method to handle multiple connections is to service each connection at a fixed interval. The advantages of this approach are that it is simple to implement and it is similar to the fixed timestep commonly used in discrete-time control. However, there are

[1] Historically, the capitalized, trademarked "UNIX™" referred to operating systems based on the original code from AT&T Bell Laboratories, while the terms "unix-like," the lower-cased "unix," and the wildcards "un*x," "*nix," etc. conventionally refer to the larger family of similar, often independently developed operating systems (Raymond, 2008). Currently, UNIX™ is a trademark of The Open Group, which licenses the brand to certified, conforming operating systems (Gray v. Novell, 2011).

computational disadvantages. Messages may be delayed because the connections are only serviced once per step. Additional computation may be required to check connections that have no new messages. Furthermore, readers may block if attempting to service a connection with no data to read, and writers may block on full write buffers. While this approach can handle a small, fixed number of connections, it is not a practical consideration for network servers because it performs poorly with a large and varying number of connections.

2.1.1.2. Process-Per-Connection

One approach to handle varying numbers of connections is to create a separate worker process or thread for each connection. Creating worker processes was traditionally popular because it is easy to implement, process creation on unix-like systems is inexpensive, and separating connections in different processes provides isolation between them. The `inetd` superserver is based entirely on the approach of starting a handler process for each new connection. In addition, on modern multi-core machines, separate processes provide true concurrency. Separate handler processes also provide the unique feature of user-based access control; this is useful for low-volume and security-critical services such as SSH. The downside of using separate processes is the overhead to create and maintain the additional processes (Tanenbaum and Bos, 2014). Each connection requires memory for the process's function call stack, and context switching between processes introduces overhead. Consequently, this approach does not scale to very large numbers of connections.

2.1.1.3. Asynchronous I/O

Asynchronous I/O promises to allow applications to initiate operations, which are performed in the background with the application notified on completion. This would seem to address the scalability issues of the process-per-connection approach. However, current implementations of asynchronous I/O are not mature. The implementation on GNU/Linux uses threads to handle background I/O and scales poorly (Kerrisk, 2014).

2.1.1.4. Event-Driven I/O

Event-based methods allow efficient handling of many connections through a synchronous interface that notifies applications when a connection is ready for I/O. These methods use the traditional `select` call from System V UNIX and `poll` from BSD. The more recent `kqueue` call on FreeBSD and `epoll` on Linux reduce the overhead for very large numbers of connections. Though all these calls differ slightly in their semantics, the underlying premise is the same. The application provides the kernel with a list of file descriptors, and the kernel notifies the application when one of those descriptors is ready for a requested I/O operation. While this approach does require explicitly managing lists of active connections, it efficiently scales to large numbers of connections.

A rough benchmark for network servers is the ability to handle 10 thousand concurrent network connections (C10K) (Kegel, 2006). Though at one point this was a challenging problem, it is now easily handled through event-based methods such as `epoll` and `kqueue`. The popular and efficient Nginx[2] webserver uses event-based methods as does the libevent[3] library, which underlies communication in memcached and the Google Chrome web browser, among others. For handling many concurrent connections, event-based methods are widely used and scale on ordinary hardware to thousands of concurrent connections.

2.1.2. Name Resolution and Service Discovery

Another important issue in communication is name resolution and service discovery. Humanoid robots have many distinct software modules that need to locate the underlying mechanism for communication. Many middlewares provide their own form of service discovery: CORBA (CORBA, 2011) provides its naming service to locate remote objects, ONC RPC provides the port mapper (Srinivasan, 1995) to resolve the port numbers to connect to a desired program, and ROS resolves topic names in the `rosmaster` process (Quigley et al., 2009). However, name resolution and service discovery are addressed in a standard and general way via multicast DNS (mDNS) (Cheshire and Krochmal, 2013), a peer-to-peer variation of the traditional, hierarchical domain name system (DNS). DNS and mDNS are flexible protocols and can even store arbitrary information in TXT records (Rosenbaum, 1993). Of course, non-naming features such as connection monitoring are outside the scope of DNS. Multicast DNS is a standard protocol with existing implementations, so using mDNS instead of a specialized resolution method reduces the number of separate daemons which must run as well as separate code which must be maintained. Consequently, we use mDNS in Ach to locate communication channels on remote hosts.

2.1.3. Lesson Learned

Humanoid robots need communication that is both scalable and real-time. Event-based methods impose the lowest overhead of the various POSIX communication approaches and are the typical choice for scalability-critical network servers (Gammo et al., 2004). Communication for humanoid robots would benefit from the scalability of an event-based approach, and we discuss the real-time requirements next in Section 2.2. Event-based methods operate on kernel file descriptors (Stevens and Rago, 2013), which motivated the development of the Ach kernel module (see Section 3.2). To name and locate services, the standard mDNS protocol and implementations provide the necessary capabilities; there is no need to duplicate the features of mDNS.

2.2. Real-Time Software

Humanoid robot software requires not only the complex logic and efficient communication of general purpose software but also real-time response to handle physical dynamics. The software infrastructure for humanoids should address the need for real-time performance without unnecessarily sacrificing the capabilities of general purpose systems. While it is a challenge to develop real-time software on general purpose systems, acceptable performance can still be achieved.

[2] http://nginx.org/
[3] http://libevent.org/

2.2.1. Real-Time Communication

POSIX provides a variety of general purpose communication mechanisms; however, none are ideal for robot control. Robot control requires the latest data sample each control cycle. General purpose communication, however, gives priority to older data, which must be read or flushed before newer data can be received. This is the *Head of Line (HOL) Blocking* problem. The specific issues of each POSIX communication mechanism are discussed in Dantam et al. (2015). It was this HOL blocking challenge that motivated the initial development of Ach (Dantam and Stilman, 2012), which always provides access to the most recent data sample.

Though general network servers can handle thousands of concurrent connections (Gammo et al., 2004), there is a key difference from the needs of humanoid robots. Network servers are primarily concerned with maximizing throughput – serving as many clients as possible. Robot control, on the other hand, requires minimizing latency – handling each communication operation in minimal and bounded time. Throughput-focused methods often attempt to reduce copying, e.g., by eliding a copy to a kernel buffer for network socket communication or directly mapping a buffer into another process's addresses space via shared memory or relaying a file descriptor through a local socket. However for robots, individual real-time messages are typically small, e.g., a few floating point values read from a sensor, so the overhead of copying the data is minimal. Instead, overhead from system calls and process context-switching dominates. This shift in focus from throughput to latency is one aspect of the difference between general-purpose and real-time systems, and is a concern that we consider in the design of the Ach data structure (see Section 3.1).

Network communication uses Quality of Service (QoS) mechanisms to improve response for traffic with special requirements, for example, by reserving bandwidth or offering predictable delays (Huston, 2000). Linux provides *queuing disciplines* to prioritize sent traffic and reduce HOL blocking at the sending end (Siemon, 2013). However, HOL blocking or dropped packets may still occur at the receiving end if the receiver does not process messages as quickly as they are sent. The popular, real-time Controller Area Network (CAN) includes a dedicated priority field in messages to guarantee that higher priority messages are sent first, though messages of equal priority are still processed first-in-first-out, different senders must use unique message priorities to avoid collisions, and packet routing is not considered (ISO 11898-1:2015, 2015). Higher-level communication frameworks also employ QoS, the improve predictability of communication (DDS 1.2, 2007; Hammer and Bauml, 2013; Paikan et al., 2015). Appropriate use of QoS can improve real-time network performance, but the underlying queuing of network communication still presents challenges when one needs the most recent data sample.

The Ach library that we present in Section 3.1 is an interprocess communication mechanism rather than a network protocol, resulting in a distinct set of capabilities and challenges. Network communication must address issues such as limited bandwidth, packet loss, collisions, clock skew, and security. In contrast, processes on a single host can access a unified physical memory, which provides high bandwidth and assumed perfect reliability; still, care must be taken to ensure memory consistency between asynchronously executing processes. While network protocols use QoS to prioritize traffic, Ach maintains a specific data structure (see **Figure 1B**) to guarantee constant-time access to the most recent data sample. Furthermore, Ach communication is compatible with process priorities and priority inheritance, so higher priority processes gain first access to read from and write to an Ach channel. Overall, we view Ach is complementary to network communication. The low latency and fast latest-message-access of Ach make it well suited for real-time interprocess communication.

2.2.2. Real-Time Kernels

The trade-off between throughput and latency exists also at the level of the operating system kernel. General purpose kernels such as Linux and XNU (used in MacOSX) focus on maximizing throughput while real-time kernels focus on minimizing latency. QNX and VxWorks are POSIX kernels that focus on real-time performance, but both are proprietary. Open source kernels provide greater flexibility for the user, which is important for research where requirements are initially uncertain. There are two real-time variants of the open source Linux kernel. The Linux PREEMPT_RT patch (Dietrich, 2005) seamlessly runs Linux applications with significantly reduced latency compared to 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 runs the real-time Adeos hypervisor alongside a standard Linux kernel (Gerum, 2004). It typically offers better latency than PREEMPT_RT but is less polished (Brown, 2010; Dantam et al., 2015) and its dual kernel approach complicates development. Because of the maturity, positive roadmap, and open source code base of Linux PREEMPT_RT (Fayyad-Kazan et al., 2013), we initially implement multiplexable Ach channels within this kernel.

2.2.3. Memory Allocation

Memory allocation is a particularly critical part of software development, even more so for real-time software. The ubiquitous `malloc` and `free` pose issues for real-time performance. Typical implementations are tuned for throughput over latency. The allocator in the GNU C Library (glibc) commonly used on Linux lazily batches manipulation of free lists. Calls to glibc's `free` are usually fast but sometimes take very long to complete (Lea, 2000). In contrast, the real-time focused Two-Level Segregate Fit (TLSF) allocator (Masmano et al., 2004) promises O(1) performance, though it is not tuned for multi-threaded applications. For garbage collected languages, latency is even more severe. Collection cycles introduce pause times that are unacceptable for real-time performance on humanoid robots (Johnson et al., 2015). Research on real-time garbage collection is ongoing (Yuasa, 1990; Bacon et al., 2003; Kalibera et al., 2011), while (Smith et al., 2014) use Java for real-time control by disabling garbage collection in the real-time module. If we can constrain the ordering of allocations and frees, then the situation improves. Region-based allocators impose a last-allocated, first-freed constraint, and operate in O(1) time with low overhead (Hanson, 1990). In addition, they provide the software-engineering advantage that all objects allocated from a

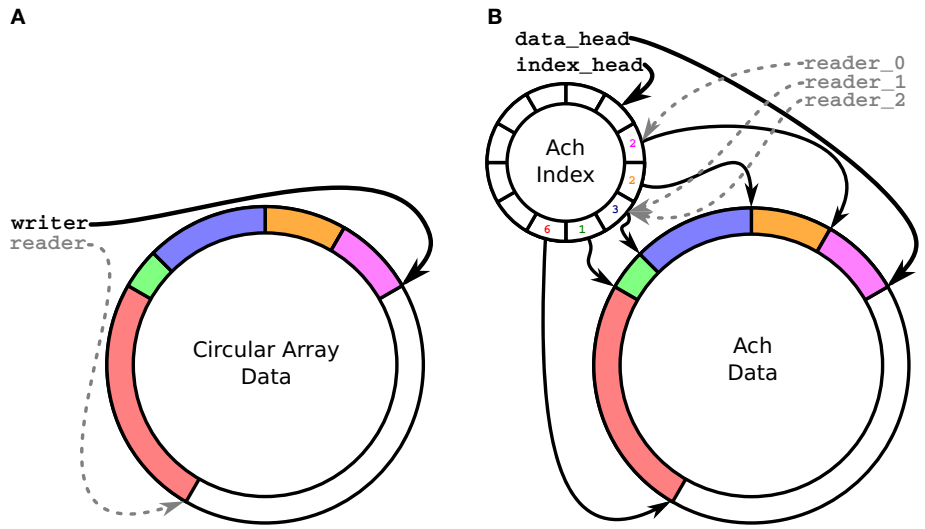

FIGURE 1 | **Logical memory structure for a conventional circular buffer (A) and an Ach channel (B)**. **(A)** shows a single writer, single reader circular buffer. **(B)** shows the Ach data structure with separate index and data buffers. The index buffer records metadata on the messages, enabling multiple readers, and detection of overwritten messages.

region can be freed with a single call, potentially reducing the bookkeeping necessary to avoid memory leaks. Though none of these memory allocation approaches are the universal solution for real-time constraints, each has advantages and is useful in the appropriate circumstances.

2.2.4. Lesson Learned

Wringing real-time performance out of a general purpose system is a careful balancing act. It requires understanding the overheads introduced by low-level calls and avoiding those with the potential to cause unacceptable resource usage. Selecting appropriate kernels and runtime support helps, but universal and guaranteed solutions are rare. 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 (Lee, 2009). This is one area where continuing research is needed.

2.3. System Integration

System integration is a major challenge in robotics (Johnson et al., 2015; Zucker et al., 2015). While this is a broad issue covering algorithms, software, hardware, and operating environments, there are still lessons from the unix community that inform integration of humanoid robot software. In this regard, the humanoid robotics community mirrors the general open source world, depending on a wide variety of software packages from globally distributed authors and running on a wide variety of underlying platforms. We discuss design issues for software extensibility and compatibility in this section. In Appendix B, we discuss how build systems and package managers help integrate the many software packages required by humanoids. These general approaches for extensible design and software management are useful for similar software integration problems on humanoids.

2.3.1. Compatibility and Extensibility

Flexible and adaptable software is crucial to humanoid robots, where requirements and platforms are continually evolving. Tools and design principles from the POSIX programing community enable software that gracefully handles both the constant churn of ongoing development and the larger shifts of evolving platforms.

2.3.1.1. Mechanism vs. Policy

A key consideration in designing flexible software is the *Separation of Mechanism and Policy* (Silberschatz et al., 2009). Flexible software helps both in development by making it easier to prototype new systems and in long-term maintenance by making it easier to adapt to changing requirements. Software is more flexible when it provides mechanisms to perform some activity, but does not dictate overly restrictive policies over when or how to execute those activities. A canonical example of this approach is the X Window System (X11), which provides a mechanism for handling the display, but defers on policies for window management and "look-and-feel" (Scheifler, 2004). Compared to other, more integrated windowing systems, X11 has been extraordinarily long-lived, surviving various alternatives, e.g., Gosling et al. (1989), Linton and Price (1993), and Thomas et al. (2003), through changing graphics platforms. For research in particular, the separation of concerns is critically important to handling requirements that evolve as understanding of the project grows. We have followed this approach also in Ach by providing a communication mechanism but not dictating policies for message encoding or event handling. This separation of *policy* from *mechanism* is important for flexibility.

2.3.1.2. Binary Compatibility

When we modify a library on the robot, it is desirable to avoid the need to modify or recompile programs using that library. This

requires maintaining *binary compatibility*. Preserving binary compatibility requires the library to export a compatible Application Binary Interface (ABI). Maintaining ABI compatibility helps users by avoiding the requirement to install multiple library versions and by reducing the need to recompile applications. Drepper (2011) provides a detailed explanation of shared libraries and compatibility. In general, preserving binary compatibility requires that symbol names not be changed or removed and that client-visible structures preserve both their total size and the offsets of their fields. In C++, changes to the class hierarchy or virtual methods break binary compatibility. These requirements present challenges as new features are added to software. Adding new functions will not break compatibility; however, changes to structures may. There are some options to change structures while still preserving the ABI. One option, used in Ach, is to reserve space in structure declarations for fields to be added in the future. Reserving space maintains the total size of the structure when new fields are added. The cost is additional memory usage for the reserved space. Another option is to encapsulate all structure allocation and access within the library, exposing structures only as opaque pointers along with functions to access their fields. Encapsulating allocation permits changes to the underlying structure. The cost is the additional indirection and function call overhead to access the structure. When breaking the ABI is necessary, it is desirable to permit multiple ABI versions of the library to be installed together. Installing multiple ABI versions can be done by changing the library name, typically by including the version number in the library name; however, this may unnecessarily create different ABI versions when the new library version actually maintains binary compatibility. The alternative is to maintain a separate ABI version from the library version number. ABI versioning is handled differently on different operating systems; however, the Libtool component of Autotools provides a uniform interface for library versioning (GNU Libtool, 2015). While preserving ABI compatibility requires care and planning, it is generally possible and benefits library users.

2.3.1.3. Source Compatibility

If we cannot maintain binary compatibility when we modify a library, it is desirable to at least require only a recompilation of programs using the library rather than modifications to the programs' source code. This requires maintaining *source compatibility*. Preserving source compatibility requires a library to export a compatible Application Programing Interface (API). Maintaining API compatibility helps users by avoiding or reducing the need for them to modify their code to accommodate API changes. API compatibility is easier to maintain than ABI compatibility, generally requiring only that symbols not be removed or renamed and that argument lists remain the same. If such changes are necessary, there are some options to reduce the burden on users. One can give users time to change their code by first *deprecating* symbols before they are removed. For example, the `gets` function, vulnerable to buffer overflows, was deprecated in ISO/IEC 9899:1999 (1999) and removed in ISO/IEC 9899:2011 (2011). When structure fields must be renamed, one can preserve API compatibility by including both names within an anonymous union field. The old name can then be marked as deprecated. If it is possible that additional arguments may at some point be needed for a function, one can pass multiple arguments as fields within a structure or as items in a bitmasked integer. This allows additional arguments to be later included as fields in the structure or bits of the integer. This approach is used by the POSIX theads API (pthreads) in their various attribute structure arguments (POSIX, 2008). Several functions in Ach also take a similar attribute structure as an argument. Taking these precautions to preserve API compatibility eases the task of software maintenance for library users.

2.3.1.4. Language Selection

Programing language selection is an important, though contentious issue, and no language is universally ideal for the diverse needs of humanoid robots. Developing complex applications is easier in high-level, garbage-collected languages, while strict real-time requirements preclude garbage-collection, leaving lower-level languages such as C and C++. Though C++ has many features over C that are sometimes useful, it comes at a cost which should be considered. C is often preferred by performance-sensitive projects, e.g., the Linux kernel, because it is easier for the programmer to understand and control important, low-level details such as error handling and memory allocations which C++ abstracts through exceptions and constructors. C++ also presents compatibility issues. Because C identifiers map directly to assembly language symbols, it is generally possible to link C code built with different compilers. C++, on the other hand, uses implementation-specific name mangling on identifiers, e.g., to handle overloaded functions, so linking C++ code built with different compilers may not be possible. Changes to operating systems ABIs for C++, though still infrequent, occur more often than for C. When performance requirements permit high-level, garbage-collected languages, binding low-level libraries written in C is generally easier than for C++. C is universally supported among high-level languages for foreign function bindings, e.g., JNI for Java, CFFI for Lisp, and ctypes for Python, whereas the ability to directly interact with C++ classes is less common. Because Ach is performance sensitive and real-time, it is implemented in C. To interface with high-level, non-real-time modules, Ach provides foreign function bindings for Common Lisp, Python, and Java. Given the trade-offs among programing languages, one should be judicious in selecting languages for implementations and interfaces.

2.3.2. Lesson Learned

The unix programing tradition provides many tools and conventions to assist with system integration of humanoid robot software. Following established conventions to preserve ABI and API compatibility makes software easier to use by reducing the system administration and software maintenance task for users. Appropriate languages ease software development and maintenance while still providing acceptable performance. Though this is far from covering the full range of system integration issues for humanoid robots, it goes a long way toward addressing software-specific system integration.

3. EXTENDING LINUX COMMUNICATION

POSIX provides a rich variety of communication methods that are well suited for general purpose information processing, but

none are ideal for real-time robot control. General computation favors throughput over latency. POSIX communication favors older data over newer. In contrast, real-time control requires low latency access to the newest data. Dantam et al. (2015) discusses the challenges of POSIX communication in detail. This gap has made it difficult to develop real-time applications in the multi-process POSIX style. To address this communication need, we developed the Ach library.

3.1. The Ach IPC Library

Ach provides a message bus or publish-subscribe style of communication between multiple writers and multiple readers (Dantam et al., 2015). Robots using Ach have multiple 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. The primary unique feature of Ach is that newer messages always supersede older messages whereas POSIX communication gives priority to older data and will block or drop newer messages when buffers are full. Ach's latest-message semantics are appropriate for continuous, time-varying signals such as reference velocities or position measurements. In other cases where reliable messaging is required, such as updating a PID gain value, Ach may provide sufficient reliability by using a separate channel with a large buffer; however, this is a secondary consideration. Ach's primary focus on latest-information, publish-subscribe messaging give it unique capabilities for real-time communication of physical data samples.

3.1.1. Relation to Robotics Middleware

There are many other communication systems developed for robotics; however, Ach provides a unique set of features and capabilities. First, many other systems operate as frameworks (Bruyninckx et al., 2003; Metta et al., 2006; Quigley et al., 2009) that impose specific structure on the application. Sometimes this structure is helpful when it fits the desired application, and other times such imposed structure may impede development if the requirements are outside the particular framework's model. In contrast, Ach strictly adheres to the idea of *mechanism, not policy* (see Section 2.3.1.1), providing a flexible communication method that is easily integrated with other approaches (see Appendix A). One could also view Ach as providing low-level capabilities that could serve as a useful building-block for such higher-level frameworks. Second, many other systems focus on network communication (Metta et al., 2006; Quigley et al., 2009; Huang et al., 2010; Hammer and Bauml, 2013). In contrast, Ach focuses on local, interprocess communication. This focus enables it to achieve superior performance in its domain (Dantam et al., 2015), and we view Ach as complementary to various network protocols. Finally, Ach provides unique semantics that make it especially suited to real-time communication of continuously varying data. Similar to multicast methods (Huang et al., 2010), Ach efficiently supports multiple senders and receivers. Ach implicitly supports *process priorities* whereas network-based methods use *QoS* (DDS 1.2, 2007; Hammer and Bauml, 2013). Crucially, Ach eliminates any possibility HOL blocking (see Section 2.2.1). Network-based methods can handle HOL blocking at the sending and receiving ends (Metta et al., 2006), but dealing with assumptions in intermediate infrastructure and code is a difficult challenge (Gettys and Nichols, 2012). Overall, the unique design decisions underlying Ach result in special advantages for local, real-time communication, and we consider Ach as a key component within a larger robot software system.

3.1.2. Design of Ach

The data structure for each channel, shown in **Figure 1B**, is a pair of 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. Ach provides additional capabilities compared to a typical circular buffers, such as in **Figure 1A**:

- Ach allows multiple receivers;
- Ach always allows access to the newest data;
- Ach drops the oldest data – instead of the newest data – when the buffer is full.

Two procedures compose the core of ach: `ach_put` and `ach_get`. Detailed pseudocode is provided in Dantam and Stilman (2012), and their use is discussed in the Ach manual (Dantam, 2015b) and programing reference (Dantam, 2015a).

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

Algorithm 1 (ach_put). There are four broad steps to the `ach_put` 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.

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.

Algorithm 2 (ach_get). There are four broad steps to the `ach_get` procedure:

1. If given the option argument 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 given the option argument to return the newest message, use the newest 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.
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.

Ach provides unique semantics compared to traditional POSIX communication. Processes on a single host can access a unified physical memory, which 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.

3.1.3. User Space Limitations

The initial implementation of Ach located the data structure shown in **Figure 1B** in POSIX shared memory and synchronized access using a mutex and a condition variable. This presented a few potential error modes and limitations: a rogue or faulty process could deadlock or corrupt a channel and each thread was limited to waiting for data on single channel at a time. We discuss these potential issues next and resolve them with the kernel space implementation described in Section 3.2.

While the formal verification of Ach (Dantam et al., 2015) guarantees that it will not deadlock with regular use of the library calls, deadlock may still occur if a reader or writer dies, e.g., with a `kill -9`, inside a library call. This is partially mitigated by the use of robust POSIX mutexes, which detect this condition and handle interrupted reads. Additional code could be added, which would rollback an interrupted write.

Because all processes accessing the channel must have read and write access to the shared memory region, a rogue process could corrupt the channel data structures. Currently, unintentional corruption is weakly detected with guard bytes. This could be improved with better sanity checks of the channel and automatic recreation of corrupted channels.

The use of POSIX threads synchronization primitives limits each thread to wait for new messages on a single channel at a time. Readers wait for new messages on a per-channel POSIX condition variable and are notified by the writer when a new message is posted. POSIX threads are limited to waiting on only a single condition variable at a time; thus, there is no way in this implementation for a thread to simultaneously wait for data on multiple channels. Alternative file-based notification, e.g., using pipes or sockets, would allow multiplexing but may cause extra context-switching and additional logic would be required to ensure that tasks run in priority order. This semantic limitation was the primary motivation for the development of the kernel space Ach implementation.

3.2. Kernel Space Ach

To address the limitations of user space Ach presented in Section 3.1.3, we develop a new kernel space implementation of the Ach data structure and procedures. This implementation runs in the Linux kernel. The channel buffers shown in **Figure 1B** are located in kernel memory, protecting channels from corruption and deadlock (see **Figure 2**). Critically, channels are accessed from user space via file descriptors, enabling efficient multiplexing through event-based `poll`/`select` style calls. This enables efficient real-time communication using established network programing idioms.

3.2.1. Kernel Module Implementation

Ach is implemented in kernel space as a Linux module that creates a device file for each channel. When the module is loaded, it creates the `/dev/achctrl` device to manage channel creation and removal. Each channel is represented with a separate virtual device, e.g., `/dev/ach-foo` for channel `foo`. These virtual devices are not accessed directly by applications but instead are accessed through the Ach library using the same API as user space channels. This provides backward source-compatibility with the user space implementation and allows applications to freely switch between user and kernel space channels. The library functions to create channels (`ach_create`) and remove channels (`ach_unlink`) operate on kernel channels through `ioctl` system calls on the `/dev/achctrl` device. The library functions to send (`ach_put`) and receive (`ach_get`) messages map to `write` and `read`, respectively. Additional parameters for receiving messages, such as timeouts or flags to retrieve the newest message, are passed to the kernel via `ioctl`s. Event-based multiplexing of ach channels – alongside sockets, pipes, and other file descriptors – is possible by passing the file descriptor for the channel device file to `poll`, `select`, etc; the kernel module performs the appropriate notification when a new message is posted to the channel. By providing this kernel-supported, file-descriptor-based interface to ach channels, we improve the ability to handle multiple data sources and to interoperate with other communication mechanisms using the standard POSIX event-based I/O functions.

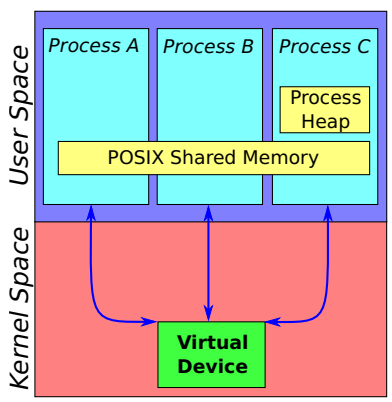

FIGURE 2 | Backing memory locations for Ach channels. Channels in the process heap are private to the process. Channels in POSIX shared memory are accessible by multiple processes. The kernel channels presented in this paper are available through per-channel virtual devices and are also accessible by multiple processes. The virtual devices are accessed from user space through file descriptors, meaning the kernel channels can be multiplexed via `select`, `poll`, etc. Because kernel channels are in protected kernel memory, they cannot be corrupted by rogue processes.

Multiplexing of Ach kernel channels is possible with the following steps:

Algorithm 3 (Ach Multiplexing).

1. Open all channels with `ach_open`.
2. Obtain the channels' file descriptors by calling `ach_channel_fd`, and record the file descriptors in the `struct pollfd`.
3. Call `poll` to wait for new data on any channel.
4. Call `ach_get` on channels with new data.

Appendix A provides a complete example of multiplexing Ach channels alongside conventional POSIX streams.

3.2.2. Advantages of Kernel Space Ach

The in-kernel Ach implementation removes the limitations of the user space implementation discussed in Section 3.1.3. Primarily, it permits multiplexing of multiple channels alongside other POSIX communication mechanisms using standard and efficient event-based I/O, e.g., `select` and `poll`. For humanoid robots, where a process may need to receive data from a large number of sources, this ability to conduct efficient I/O is a critical advantage. In addition, the potential of user space channels for corruption and deadlock from rogue processes is eliminated in the kernel implementation. Kernel channels are in kernel memory which cannot be directly accessed by user space processes. The kernel implementation eliminates the faults of user space Ach, giving it the same features and robustness as standard POSIX communication mechanisms.

3.2.3. Disadvantages of Kernel Space Ach

The in-kernel implementation does present some potential disadvantages for portability and a caveat regarding robustness.

While the user space implementation used standard POSIX calls, the in-kernel implementation is Linux-specific, running on vanilla and PREEMPT_RT kernels. This would present an issue if it is necessary to use a non-Linux kernel, requiring additional work to implement Ach within that separate kernel. However, the Ach code is modular and well-factored – the core code is largely shared between the user and Linux kernel space implementations. Adding an additional backend for another kernel should not be a major challenge, and we hope to develop a kernel module for Xenomai in the future.

Code in kernel space faces stricter correctness requirements than in user space. Software errors in the Ach kernel module – as with any kernel space code – can potentially crash the entire operating system. However, for humanoid robots, any software error, whether in user or kernel space, can potentially – and literally – crash the entire robot. Thus, moving Ach to the kernel does not significantly change the severity of potential errors. Still, it is important to understand the strict requirements on kernel space code.

3.3. Benchmarks

We provide benchmark results of message latency for Ach compared to a variety of other kernel communication methods.[4] 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 to 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 timestamped messages at the desired frequency;
4. Receivers: Receive messages and record latency of each messaged based on the timestamp.

Figure 3 shows the results of the benchmarks, run on an Intel® Core™ i7-4790 at 3.6 GHz under Linux 3.18.16-rt13 PREEMPT RT. We compare Ach with several common POSIX communication mechanisms. In contrast to Ach and these lightweight, kernel methods, heavyweight middleware such as ROS and CORBA impose several times greater communication latency (Dantam et al., 2015). All the methods shown in **Figure 3** are similar in performance, indicating that the bulk of overhead is due to the process context switch rather than the minimal time for the actual communication operation. For the single receiver case, both user and kernel space Ach provide comparable latency to POSIX communication. While the latency is similar, there are also important feature differences. Kernel space Ach can multiplex across multiple channels while user space Ach cannot, and unlike POSIX communication, Ach directly supports multiple subscribers. The results in **Figure 3** show that Ach provides strong performance, along with its ability to handle multiple subscribers and its unique latest-message-favored semantics.

3.4. Case Study: Baxter Robot

We use the new kernel space Ach implementation in our control system for the Baxter robot. The Baxter is a dual-arm manipulator. Each arm has 7 degrees of freedom and a parallel jaw gripper. The integrated electronics enable position, velocity, and torque control of the robot's axes. We implement several modes of multi-axis control using this interface.

Figure 4 shows our Ach-based control system for the Baxter robot. In this system, each driver and controller runs as a separate, isolated operating system process. Previous ach-based systems ran drivers as separate processes, but multiple controllers were combined into a single processes (Dantam et al., 2015). Because kernel space Ach channels are efficiently multiplexable, we can run the Baxter controllers in separate processes, each outputing to a distinct channel. The `ref` process receives messages from all these channels, selecting the highest priority message to communicate to the robot. This design efficiently integrates multiple control modes for the robot, each running in separate processes.

In addition to the processes shown in **Figure 4**, we also run a separate, non-real-time logging process. All the control system processes write log messages to a single event channel. In normal operation, the logger waits for new messages on the event channel, stepping through each posted message and recording it to a log file. However, if log messages are posted faster than the logger can process them, for example whether due limited CPU cycles

[4] Benchmark code available at http://github.com/golems/ach

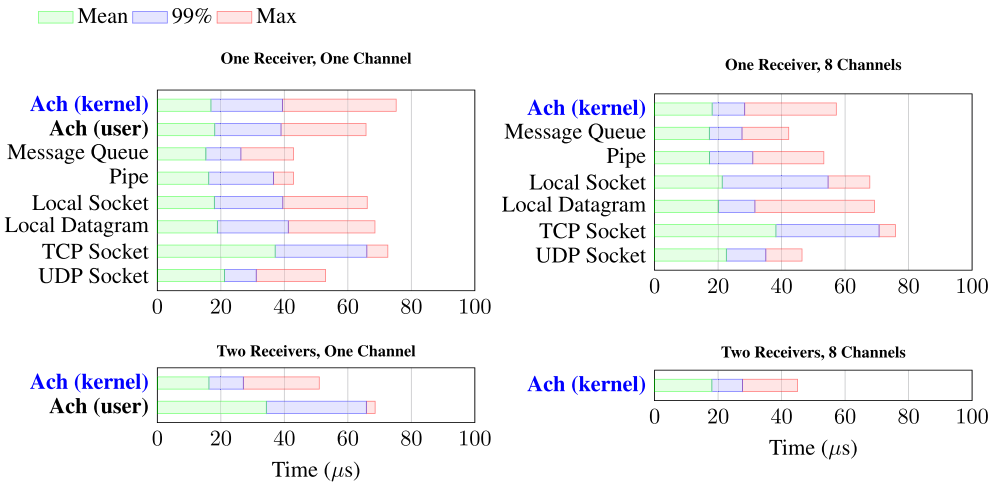

FIGURE 3 | Message Latency for Ach and POSIX communication. Ach has comparable performance to optimized POSIX communication, and unlike POSIX methods, Ach enables multiple receivers and prioritizes newer data. In the plots, "Mean" is the average latency over all messages, "99%" is the latency that 99% of messages beat, and "Max" is the maximum recorded latency. The long-tail of the worst-case behavior arises from the focus of general purpose platforms on average-case performance instead of bounded response needed by real-time systems. For example, lazy algorithms and hardware memory caches improve average performance, but still leave the worst-case cost to pay.

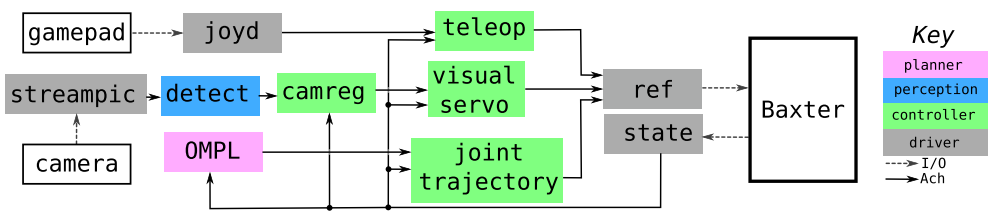

FIGURE 4 | Ach-based control system for the Baxter robot. Three control pipelines run concurrently: a teleoperation pipeline for manual control (top), a camera registration and visual servoing pipeline (middle, Dantam et al., 2016), and a joint planning (Sucan et al., 2012) and interpolation pipeline (bottom). The `ref` process multiplexes the output of these three pipelines and selects the highest priority command to send to the robot. A non-real-time logging process (not shown) also receives informational and error messages from the other control system processes over a single Ach channel and records the messages in a log file.

or a programing error, some messages will be skipped as they are overwritten in the circular buffer (see **Figure 1B**). The alternative to skipping messages would be to block the message sender until the logger can process messages or to buffer an unbounded number of unprocessed log messages. Neither blocking a real-time process nor growing a buffer without bound is desirable in a real-time control system. Instead, Ach overwrites the oldest message and the system continues, missing only the skipped log messages.

The primary advantages of this multi-process control system design are modularity and robustness to software errors. Separating drivers and controllers into different processes means they can be developed and tested independently. This separation is particularly useful to prototype new controllers, which can be developed without disturbing previously tested work. For example, we developed and tested the workspace controller described in Kingston et al. (2015) without any modification to the processes in **Figure 4**. Failures encountered while developing and testing the new controller did not affect the other running control processes. In contrast, combining multiple controllers in a single process as was necessary for the user space Ach systems presented in Dantam et al. (2015) means that errors encountered while prototyping a new controller will not interfere with existing control modes. In research applications on robot control, easing controller development and testing is a key advantage.

4. CONCLUSION

We have discussed the application of unix design principles to robot software. Among the various unix tools and conventions, the multiprocess design typical of unix applications improves modularity and robustness, critical needs for complex systems such as humanoid robots. We enable this multiprocess design for real-time control with the efficient Ach communication library. This approach is general, applying to multiple types of robots and other complex mechatronic systems that require coordination of many hardware devices and software modules. Ach fills a need in robot software unmet by POSIX, providing a communication

mechanism that supports multiple receivers and gives priority to newer messages. The kernel space implementation presented in this paper exposes a file descriptor interface, enabling multiplexing of messages from many sources and efficient integration with other communication methods. Kernel space Ach enables development of real-time robot software in the conventional modular, robust, multi-process unix style.

The approach to robot software development we have presented – advocating use of existing tools and conventions from the unix programing community – is inherently conservative. While building on unix provides a mature set of capabilities, new research in techniques to automate software development have the potential to radically improve the development process.

Formal methods for software verification and synthesis can greatly ease software development. Some tools are already in widespread use (Cimatti et al., 2002; Ball et al., 2004), and we used SPIN (Holzmann, 2004) to verify Ach in Dantam et al. (2015). Formal methods continue to be an active research area in robotics (Wang et al., 2009; Dantam and Stilman, 2013; Liu et al., 2013; Nedunuri et al., 2014; Lignos et al., 2015), bridging the fields of software engineering, automatic control, and planning and scheduling. Though limits remain on which problems admit formal reasoning, further research has the potential to broaden the scope of formal methods for humanoid robot software, changing our fundamental approach to software development.

AUTHOR CONTRIBUTIONS

ND, primary author of overall software, secondary author of additional modules described in paper. KB, primary author of additional modules described in paper. MJ, additional contributions to software. TB, supported work at Atlas Copco and Prevas, contributions to design and testing methods. LK, supported work at Rice University, contributions to testing and presentation of the work.

ACKNOWLEDGMENTS

Work at Rice University by ND and LK has been supported in part by NSF IIS 1317849, NSF CCF 1514372, and Rice University Funds. Work by KB, MJ, and TF has been supported by Atlas Copco Rock Drills AB. We thank Zachary K. Kingston for his development work and continuing maintenance of the presented Baxter software example.

REFERENCES

Bacon, D. F., Cheng, P., and Rajan, V. (2003). "A real-time garbage collector with low overhead and consistent utilization," in *Symposium on Principles of Programming Languages*, Vol. 38 (New Orleans, LA: ACM), 285–298.

Ball, T., Cook, B., Levin, V., and Rajamani, S. K. (2004). "SLAM and static driver verifier: technology transfer of formal methods inside Microsoft," in *Integrated Formal Methods, Volume 2999 of Lecture Notes in Computer Science* (Canterbury: Springer), 1–20.

Brown, J. H., and Martin, B. (2010). *How Fast Is Fast Enough? Choosing between Xenomai and Linux for Real-time Applications*. Technical Report, Rep Invariant Systems. Available at: https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf

Brugali, D., and Scandurra, P. (2009). Component-based robotic engineering (part I)[tutorial]. *IEEE Rob. Autom. Mag.* 16, 84–96. doi:10.1109/MRA.2009.934837

Bruyninckx, H., Soetens, P., and Koninckx, B. (2003). "The real-time motion control core of the Orocos project," in *International Conference on Robotics and Automation*, Vol. 2 (Taipei: IEEE), 2766–2771.

Catkin. (2015). *Catkin Conceptual Overview*. Available at: http://wiki.ros.org/catkin/conceptual_overview

Cheshire, S., and Krochmal, M. (2013). *Multicast DNS*. Internet Engineering Task Force. Available at: https://tools.ietf.org/html/rfc6762

Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., et al. (2002). "Nusmv 2: an opensource tool for symbolic model checking," in *Computer Aided Verification, CAV '02* (London: Springer-Verlag), 359–364.

CORBA. (2011). *Common Object Request Broker Architecture (CORBA/IIOP)*, 3.1.1 Edn. The Object Management Group. Available at: http://www.omg.org/spec/CORBA/3.1.1/

Dantam, N. T. (2015a). *Ach IPC Library*. Available at: http://golems.github.io/ach/api/

Dantam, N. T. (2015b). *Ach IPC User Manual*. Available at: http://golems.github.io/ach/manual/

Dantam, N. T., Amor, H. B., Christensen, H., and Stilman, M. (2016). "Online camera registration for robot manipulation," in *Experimental Robotics*. eds M. Ani Hsieh, O. Khatib, and V. Kumar (Switzerland: Springer), 179–194.

Dantam, N. T., Lofaro, D., Hereid, A., Oh, P., Ames, A., and Stilman, M. (2015). The Ach IPC library. *Rob. Autom. Mag.* 22, 76–85. doi:10.1109/MRA.2014.2356937

Dantam, N. T., and Stilman, M. (2012). "Robust and efficient communication for real-time multi-process robot software," in *International Conference on Humanoid Robots* (Osaka: IEEE), 316–322.

Dantam, N. T., and Stilman, M. (2013). The motion grammar: analysis of a linguistic method for robot control. *Trans. Rob.* 29, 704–718. doi:10.1109/TRO.2013.2239553

DDS 1.2. (2007). *Data Distribution Service for Real-time Systems*, 1.2 Edn. The Object Management Group. Available at: http://www.omg.org/spec/DDS/1.2/

Dietrich, S.-T., and Walker, D. (2005). The evolution of real-time Linux. In *7th Real Time Linux Workshop*.

Drepper, U. (2011). *How to Write Shared Libraries*. Technical Report, Red Hat. Available at: http://www.akkadia.org/drepper/dsohowto.pdf

Fayyad-Kazan, H., Perneel, L., and Timmerman, M. (2013). Linux PREEMPT-RT vs. commercial RTOSs: how big is the performance gap? *J. Comput.* 3, 135–142. doi:10.5176/2251-3043_3.1.244

Gammo, L., Brecht, T., Shukla, A., and Pariag, D. (2004). "Comparing and evaluating epoll, select, and poll event mechanisms," in *Proceedings of the 6th Annual Ottawa Linux Symposium*. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.215.7953

Gerum, P. (2004). *Xenomai – Implementing a RTOS Emulation Framework on Gnu/Linux*. Technical Report, Xenomai.

Gettys, J., and Nichols, K. (2012). Bufferbloat: dark buffers in the internet. *Commun. ACM* 55, 57–65. doi:10.1145/2063176.2063196

GNU Libtool. (2015). *GNU Libtool – Portable Dynamic Shared Object Management*. Free Software Foundation. Available at: https://www.gnu.org/software/libtool/manual/

GNU Standards. (2015). *GNU Coding Standards*. Free Software Foundation. Available at: https://www.gnu.org/prep/standards/

Gosling, J., Rosenthal, D. S., and Arden, M. J. (1989). *The NeWS Book: An Introduction to the Network/Extensible Window System*. New York, NY: Springer Science & Business Media.

Gray v. Novell, Inc. (2011). *United States Court of Appeals, Eleventh Circuit, NO. 09-11374*. Florida.

Hammer, T., and Bauml, B. (2013). "The highly performant and realtime deterministic communication layer of the ardx software framework," in *2013 16th International Conference on Advanced Robotics (ICAR)* (Montevideo: IEEE), 1–8.

Hanson, D. R. (1990). Fast allocation and deallocation of memory based on object lifetimes. *Software Pract. Exp.* 20, 5–12.

Holzmann, G. (2004). *The Spin Model Checker*. Boston, MA: Addison Wesley.

Huang, A. S., Olson, E., and Moore, D. C. (2010). "LCM: Lightweight communications and marshalling," in *Intelligent Robots and Systems* (Taipei: IEEE), 4057–4062.

Huston, G. (2000). *Next Steps for the IP QoS Architecture*. Internet Engineering Task Force. Available at: https://tools.ietf.org/html/rfc2990

ISO 11898-1:2015. (2015). *Road Vehicles – Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling*, 2 Edn. ISO. Available at: http://www.iso.org/iso/catalogue_detail.htm?csnumber=63648

ISO/IEC 9899:1999. (1999). *Programming Languages – C*, 2 Edn. ISO/IEC. Available at: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=29237

ISO/IEC 9899:2011. (2011). *Programming Languages – C*, 3 Edn. ISO/IEC. Available at: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=57853

Johnson, M., Shrewsbury, B., Bertrand, S., Wu, T., Duran, D., Floyd, M., et al. (2015). Team IHMC's lessons learned from the DARPA robotics challenge trials. *J. Field Rob.* 32, 192–208. doi:10.1002/rob.21571

Kalibera, T., Pizlo, F., Hosking, A. L., and Vitek, J. (2011). Scheduling real-time garbage collection on uniprocessors. *ACM Trans. Comput. Syst.* 29, 8. doi:10.1145/2003690.2003692

Kegel, D. (2006). *The C10K Problem*. Available at: http://www.kegel.com/c10k.html

Kerrisk, M. (2014). *aio – POSIX Asynchronous I/O Overview*, 3.74 Edn. Linux Man-Pages. Available at: https://www.kernel.org/doc/man-pages/

Kingston, Z. E., Dantam, N. T., and Kavraki, L. E. (2015). "Kinematically constrained workspace control via linear optimization," in *International Conference on Humanoid Robots* (Seoul: IEEE), 758–764. doi:10.1109/HUMANOIDS.2015.7363455

Lea, D. (2000). *A Memory Allocator*. Available at: http://g.oswego.edu/dl/html/malloc.html

Lee, E. A. (2009). Computing needs time. *Commun. ACM* 52, 70–79. doi:10.1145/1506409.1506426

Lignos, C., Raman, V., Finucane, C., Marcus, M., and Kress-Gazit, H. (2015). Provably correct reactive control from natural language. *Auton. Robots* 38, 89–105. doi:10.1007/s10514-014-9418-8

Linton, M., and Price, C. (1993). "Building distributed user interfaces with fresco," in *Proceedings of the 7th X Technical Conference* (Boston, MA: O'Reilly), 77–87.

Liu, J., Ozay, N., Topcu, U., and Murray, R. M. (2013). Synthesis of reactive switching protocols from temporal logic specifications. *Trans. Autom. Control* 58, 1771–1785. doi:10.1109/TAC.2013.2246095

Masmano, M., Ripoll, I., Crespo, A., and Real, J. (2004). "TLSF: a new dynamic memory allocator for real-time systems," in *Euromicro Conference on Real-Time Systems* (Catania: IEEE), 79–88.

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: yet another robot platform. *Int. J. Adv. Rob. Syst.* 3, 43–48. doi:10.5772/5761

Nedunuri, S., Prabhu, S., Moll, M., Chaudhuri, S., and Kavraki, L. E. (2014). "SMT-based synthesis of integrated task and motion plans from plan outlines," in *International Conference on Robotics and Automation* (Hong Kong: IEEE), 655–662.

Paikan, A., Pattacini, U., Domenichelli, D., Randazzo, M., Metta, G., and Natale, L. (2015). "A best-effort approach for run-time channel prioritization in real-time robotic application," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Hamburg: IEEE), 1799–1805.

pkg-config. (2013). *pkg-config*. Available at: http://www.freedesktop.org/wiki/Software/pkg-config/

Poettering, L. (2014). *Revisiting How We Put Together Linux Systems*. Available at: http://0pointer.net/blog/revisiting-how-we-put-together-linux-systems.html

POSIX. (2008). *IEEE Std 1003.1-2008*. The IEEE and The Open Group. Available at: http://pubs.opengroup.org/onlinepubs/9699919799/

Quarterman, J. S., Silberschatz, A., and Peterson, J. L. (1985). 4.2 BSD and 4.3 BSD as examples of the UNIX system. *ACM Comput. Surv. (CSUR)* 17, 379–418. doi:10.1145/6041.6043

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *International Conference on Robotics and Automation, Workshop on Open Source Robotics* (Kobe: IEEE).

Raymond, E. S. (2003). *The art of Unix Programming*. Boston, MA: Addison-Wesley Professional.

Raymond, E. S. and Landley, R. (2008). *OSI Position Paper on the SCO vs. IBM Complaint*. Available at: http://www.catb.org/~esr/hackerlore/sco-vs-ibm.html

Rosenbaum, R. (1993). *Using the Domain Name System To Store Arbitrary String Attributes*. Internet Engineering Task Force. Available at: https://tools.ietf.org/html/rfc1464

Scheifler, R. W. (2004). *X Window System Protocol*. X Consortium, Inc. Available at: http://www.x.org/archive/X11R7.5/doc/x11proto/proto.pdf

Siemon, D. (2013). Queueing in the linux network stack. *Linux J.* Available at: http://www.linuxjournal.com/content/queueing-linux-network-stack

Silberschatz, A., Galvin, P. B., and Gagne, G. (2009). *Operating System Concepts*. Danvers, MA: J. Wiley & Sons.

Smith, J., Stephen, D., Lesman, A., and Pratt, J. (2014). "Real-time control of humanoid robots using OpenJDK," in *International Workshop on Java Technologies for Real-time and Embedded Systems* (Niagara Falls, NY: ACM), 29.

Srinivasan, R. (1995). *Binding Protocols for ONC RPC Version 2*. Internet Engineering Task Force. Available at: https://tools.ietf.org/html/rfc1833

Stevens, W. R., and Rago, S. A. (2013). *Advanced Programming in the UNIX Environment*. Indianapolis: Addison-Wesley.

Sucan, I., Moll, M., and Kavraki, L. E. (2012). The open motion planning library. *Rob. Autom. Mag.* 19, 72–82. doi:10.1109/MRA.2012.2205651

Tanenbaum, A. S., and Bos, H. (2014). *Modern Operating Systems*. Upper Saddle River, NJ: Prentice Hall Press.

Thomas, M., Lupu, E., and Rükert, D. (2003). *Y: A Successor to the x Window System*. Technical Report, Imperial College London. Available at: http://www3.imperial.ac.uk/pls/portallive/docs/1/18619743.PDF

Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., and Mahlke, S. (2009). "The theory of deadlock avoidance via discrete control," in *SIGPLAN Notices*, Vol. 44 (Savannah: ACM), 252–263.

Yuasa, T. (1990). Real-time garbage collection on general-purpose machines. *J. Syst. Software* 11, 181–198. doi:10.1016/0164-1212(90)90084-Y

Zucker, M., Joo, S., Grey, M. X., Rasmussen, C., Huang, E., Stilman, M., et al. (2015). A general-purpose system for teleoperation of the DRC-HUBO humanoid robot. *J. Field Rob.* 32, 336–351. doi:10.1002/rob.21570

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

The reviewer AP and the handling editor LN declared their shared affiliation, and the handling editor states that the process nevertheless met the standards of a fair and objective review.

APPENDIX

A. Ach Multiplexing Example

Ach kernel channels can be multiplexed using the conventional `select`, `poll`, etc. functions. We provide an example program that multiplexes two Ach channels along with the program's standard input and echoes all data to standard output. This example demonstrates Ach's efficient handling of multiple data sources and compatibility with other forms of POSIX communication.

```c
#include <stdlib.h>
#include <pthread.h>
#include <inttypes.h>
#include <stdio.h>
#include <poll.h>
#include <unistd.h>
#include <ach.h>

int main(int argc, char **argv)
{
    const char *names[] = {"channel-0", "channel-1"};
    const size_t n_channels = sizeof(names) / sizeof(names[0]);
    const size_t n_pfd = n_channels + 1;
    struct ach_channel channel[n_channels];
    struct pollfd pfd[n_pfd];

    /*****************************/
    /* Initialize pollfd structs */
    /*****************************/
    for( size_t i = 0; i < n_channels; i ++ ) {
        /* Open Channel */
        enum ach_status r = ach_open( &channel[i], names[i], NULL );
        if( ACH_OK != r ) {
            fprintf(stderr, "could not open channel '%s': %s\n",
                    names[i], ach_result_to_string(r));
            exit(EXIT_FAILURE);
        }
        /* Get channel file descriptor */
        r = ach_channel_fd( &channel[i], &pfd[i].fd );
        if( ACH_OK != r ) {
            fprintf(stderr, "could not get file descriptor for channel '%s': %s\n",
                    names[i], ach_result_to_string(r));
            exit(EXIT_FAILURE);
        }
        /* Set events to poll for */
        pfd[i].events = POLLIN;
    }
    /* Also, poll() standard input */
    pfd[n_channels].fd = STDIN_FILENO;
    pfd[n_channels].events = POLLIN;


    /***************/
    /* poll() loop */
    /***************/
    for(;;) {
        /* poll() for new data */
        int r_poll = poll( pfd, n_pfd + 1, -1 );
        if( r_poll < 0 ) {
            perror("poll");
            exit(EXIT_FAILURE);
        }
        /* Find file descriptors with new data */
        for( size_t i = 0; i < n_pfd && r_poll > 0; i++ ) {
            if( (pfd[i].revents & POLLIN) ) {
                char buf[512];
                size_t data_size = 0;
                if( i < n_channels ) {
                    /* Get new data on an Ach channel */
                    enum ach_status r = ach_get( &channel[i], buf, sizeof(buf), &data_size,
```

```
                                                NULL, ACH_O_NONBLOCK | ACH_O_FIRST );
                    switch(r) {
                    case ACH_OK:
                    case ACH_MISSED_FRAME:
                        break;
                    default:
                        fprintf( stderr, "Error getting data from '%s': %s\n",
                                 names[i], ach_result_to_string(r));
                        exit(EXIT_FAILURE);
                    }
                } else {
                    /* Read new data on a file descriptor */
                    ssize_t r = read(pfd[i].fd, buf, sizeof(buf));
                    if( r < 0 ) {
                        perror("read()");
                        exit(EXIT_FAILURE);
                    } else {
                        data_size = r;
                    }
                }
                /* Echo the data to standard output */
                ssize_t wr = write(STDOUT_FILENO, buf, data_size);
                if( wr < 0 ) {
                    perror("write");
                    exit(EXIT_FAILURE);
                }
                r_poll--;
            }
        }
    }
    return 0;
}
```

B. Configuration, Building, and Packaging

Software build systems and package managers are useful tools to address the system integration needs of humanoid robot software. Open source software distributions such as Debian and FreeBSD have developed approaches for integrating and maintaining enormous numbers of software packages. Their goal is to make software that is portable, that builds robustly, and that is easy to install, upgrade, and remove. These tools are general and suited to the needs of humanoids as well.

Different humanoid robots provide varying hardware capabilities and software environments, and it is important that humanoid software be adaptable across these different robots. A key step to achieving this portability is the *configuration* step of the build process, where the software adapts to conditions of the environment in which it must run. For example, configuration may determine whether to use the previously mentioned `epoll` or `kqueue` calls depending on whether it must run on Linux or FreeBSD, or it may determine how to interface with the fieldbus linking the robot's embedded electronics. In general, configuration chooses alternate implementations or optional components to build based on the available features of the host system. This adaptability is vital to building software that is portable and that builds robustly.

The two predominant build systems are the GNU Autotools and CMake. Overall, both offer similar capabilities with a number of superficial differences. There is, however, a difference in design philosophy that influences the use of these systems. Autotools assumes little about the host platform beyond a POSIX shell, testing at compile-time for essentially every other feature, an approach that is robust to new and changing platforms but requires additional time for the feature tests. In contrast, CMake maintains a database of modules for platforms and libraries, which can reduce compilation time by omitting feature tests but is unhelpful for differing platforms and dependencies. One can also maintain a build-system agnostic database of available libraries using pkg-config (pkg-config, 2013), which works with CMake, Autotools, and other build systems. Additionally, projects using Autotools generally follow a strict set of conventions such that all can be configured, built, and installed by the same procedure (GNU Standards, 2015). There are fewer established conventions for CMake so it is common for different CMake projects to require different steps in the build process. One should consider the need for adaptability, conformity, and configuration performance when selecting a build system.

We use Autotools to build Ach due to their maturity and strict conventions compared to CMake. In addition, Autotools enable more direct integration with the Make-based build system of the Linux kernel, which simplifies building and installing the Ach Linux kernel module (see Section 3.2).

To manage the large number of software packages on humanoid robots, package managers are an invaluable tool. Package managers handle the details of installation, cross-package dependencies,

and package versioning. The two main styles of package managers are binary-based and source-based. Binary-based package managers download and install pre-compiled packages. Examples include Redhat's RPM, Debian's APT, and – if viewed broadly – "App Stores" such as that of Apple's iOS and Google Play. Source-based package managers download package source code and build it on the local machine. Examples include FreeBSD ports, Gentoo Portage, and Homebrew for MacOSX. The advantage of binary packages is that no time must be spent to compile the package on the local machine. The advantage of source-based package managers is that packages can be custom-configured with optional features based on the users preferences and fewer server resources are required to store the compiled binaries. The choice of a package manager is typically dictated by the operating system distribution of the user. For handling software deployment, these package managers are mature and useful tools.

Two new data storage tools offer the potential to improve existing build systems and package managers: distributed revision control – e.g., git and mercurial – and copy-on-write (COW) filesystems – e.g., ZFS and BTRFS. Existing build systems and package managers were developed at a time when source code was typically downloaded as tarballs from a scattered collection of servers. Today, source code is often downloaded via a distributed version control system. Directly accessing the revision controlled files is a poor fit for Autotools' approach of generating a portable configuration script, and it makes some features of current package managers redundant, such as hosting and distributing multiple tarball versions and applying patches before building. Second, COW filesystems provide the capability to make cheap, writable snapshots. This could be used to maintain multiple concurrent images of the operating system, for example to install different versions of one package or two different conflicting packages. One view on how these new tools could change build systems and packages managers is given by Poettering (2014). These new developments show that while existing build systems and package managers are useful and mature, there is room for ongoing development and new innovation.

ROS (Quigley et al., 2009) provides a different view on build systems and package management, focusing on robots. ROS combines the build system and package manager into a single framework based on CMake (Catkin, 2015); however, binary packages are still distributed using the existing APT package manager. This approach eliminates some duplication of metadata – i.e., which files must be installed – necessary with the traditional distinction between package managers and build systems. However, this is unhelpful if packages are to be installed on a non-ROS system. Additionally, packages are also constrained to use the given CMake-based build system which may not always be the best fit, e.g., for non-C/C++ packages, or provide necessary capabilities, e.g., the site-based configuration feature of Autotools. Still, ROS presents an interesting take on how we build and package software.