Georgia Institute of Technology, Technical Report, GT-GOLEM-2011-001, 2011

Ach: IPC for Real-Time Robot Control

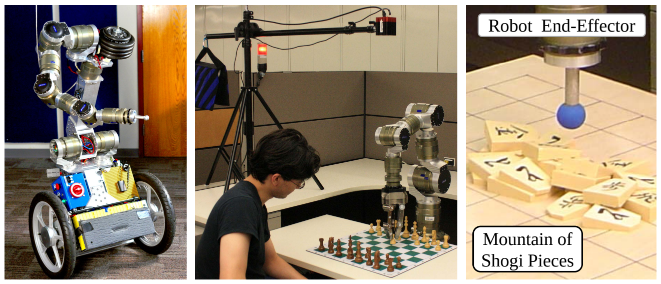
Neil Dantam and Mike Stilman

***Abstract*—We present a new Inter-Process Communication (IPC) mechanism and library. Ach is uniquely suited for coordinating perception, control drivers, and algorithms in real-time systems that sample data from physical processes. Ach eliminates the Head-of-Line Blocking problem for applications that always require access to the newest message. Ach is efficient, robust, and formally verified. It has been tested and demonstrated on a variety of physical robotic systems. Finally, the source code for Ach is available under an Open Source BSD-style license.**

I. Introduction

Real-time control of physical processes presents a special set of requirements and constraints on computerized control systems. Typically, scientists and engineers view physical processes 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 which focus on throughput over latency and favor prior data over latter data. To address these concerns, our library, *Ach*[1], provides robust and efficient communication between software components such as userspace device drivers and control algorithms in real-time.

There are three goals and assumptions that guide the design of Ach. First, to utilize decades of prior development and engineering, we choose to implement our real-time system on top of a POSIX-like Operating System (OS) [7]. 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 a critical issue for physical processes, we must make our system robust. Therefore, we assume that a multiple process approach will be more robust than a single-process or multi-threaded application. Processes include device drivers and algorithms for perception and control. This implies sampled data must be passed between OS processes using some form of Interprocess Communication (IPC). The penalty for this choice is the overhead of additional context-switching which is justified by increased robustness and modularity. Our results with Ach verify that the overhead is acceptable for robot control applications [2], [3]. Finally, we favor Open Source Software since it maximizes flexibility and control in applying the IPC to new systems. This is important both in research and development of novel devices where some requirements may be unknown from the start. These initial considerations motivate our development of an open source IPC to efficiently pass sampled data.



(a) Golem Krang (b) Chess Motion Grammar (c) Yama Grammar

Fig. 1. Robotic Systems where Ach provided all communications between hardware drivers, perception, planning and control algorithms. [2], [3], [14].

POSIX provides a rich variety of IPC mechanisms, but none of them fully satisfy our requirements. An overview of these mechanism is given in [13]. The fundamental difference is that as soon as a new sample of the signal is produced, nearly everything in our system no longer cares about the old sample. Thus, we want to always 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, it is difficult for programmers to manually manage synchronization and guarantee performance. This makes 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 standardized and open source implementation satisfied our requirements for low-latency exchange of most-recent samples, we have developed a new open source IPC library.

The contribution of this paper is a POSIX Interprocess Communication library for the real-time control of physical processes such as robotic systems. This library, called *Ach*, provides a message-bus or publish-subscribe communication semantics, an approach taken by other real-time middleware and robotics programming systems [10], [11], [8], [9]. Ach provides numerous advantages making it suitable for real-time control of physical systems. In particular, Ach is formally verified, it is efficient, and it always provides processes with the most recent data sample. To our knowledge, these benefits are unique among existing communications software.

This paper is organized as follows: Sect. II reviews the various types of POSIX IPC and explains why they do not meet our needs. Sect. III explains the ach algorithm and implementation. Sect. IV discusses the numerous advantages and few faults of ach, and provides some quantitative performance benchmarks. Finally Sect. V summarizes the paper and describes some possible future directions.

The authors are with the Robotics and Intelligent Machines Center in the Department of Interactive Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA. email: ntd@gatech.edu, mstilman@cc.gatech.edu

[1]Ach is available at http://www.golems.org/node/1526. The name "Ach" comes from the common abbreviation for the motor neurotransmitter Acetylcholine and the computer networking term "ACK."

II. Review of POSIX IPC

POSIX provides three main types of general IPC: streams, datagrams, and shared memory. We review each of these types and consider why they do not satisfy our requirements for real-time control of physical processes. A thorough survey of POSIX IPC is provided in [13].

A. Streams

Stream IPC includes pipes, fifos, local-domain stream sockets, and TCP sockets. These IPC mechanisms all 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 the old bytes before we see any new bytes. Furthermore, if we do not wish either the reading or writing process to block, then we must resort to rather more complicated Nonblocking or Asynchronous IO approaches.

B. Datagrams

1) Datagram Sockets: Datagram sockets perform somewhat better than streams in that they are less likely to block the sender. However, they 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.

2) POSIX Message Queues: While similar to Datagram sockets, POSIX Message Queues include the feature of message priorities. The downside of this is that it is possible to block if the kernel runs out of space to buffer messages. In Linux, this is a global rather than a per-queue limit. Consider a process that gets stuck and stops processing its message queue. When it starts again, the process must still read/flush old messages before getting the most recent sample.

C. Shared Memory

POSIX shared memory is very fast and we could, by simply overwriting a variable, always have the latest data. However, this provides no recourse for recovering older data that may have been missed. In addition, general use of shared memory presents synchronization issues which are notoriously difficult to solve. For these reasons, we consider direct use of shared memory inappropriate.

D. Further Considerations

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

2) Priorities: To 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 non real-time process, such as a logger, does anything.

E. Real-Time and Robotics Middlewares

In addition to the core POSIX IPC mechanisms, there exist various messaging middlewares; however, these are either not Open Source or not suitable for our real-time domain. Data Distribution Service [10] lacks compatible open source implementations. Aware2.0 [8] is not open source. Microsoft Robotics Studio is not open source and does not run on POSIX systems [9]. NAOqi [1] is a behavior-based architecture which does not meet our requirements for flexible IPC. ROS [11] provides open source TCP and UDP message transports, which suffer from the aforementioned HOL blocking problem.

III. 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. The messages sent on a channel are simple byte arrays, so arbitrary data may be transmitted such as 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 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 -C channel_name`, before the reader or writer start. This is analogous to the creation of FIFOs with `mkfifo` called either as a shell command or as a C function and is another example of Ach's flexibility. After the channel is created, each reader or writer must open the channel before it can get or put messages.

A. Channel Structure

The core data structure of an Ach channel is a pair of circular arrays located in the POSIX shared memory file, Fig. 2. The data array contains variable sized elements which store the actual message frames sent through the Ach channel. The index array contains fixed size elements where each element

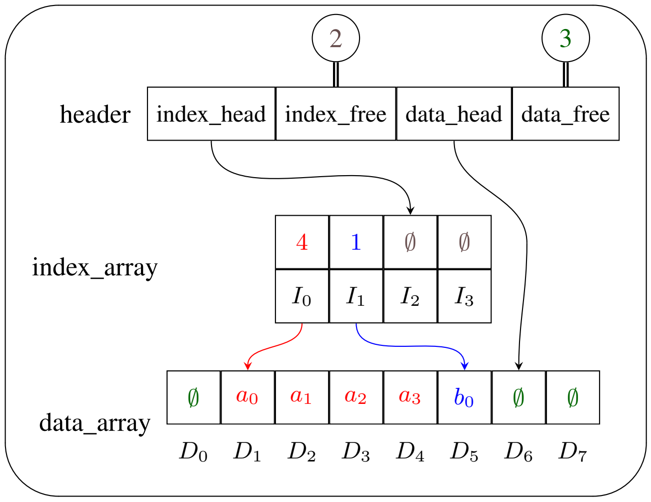


Fig. 2. Logical Memory Structure for an Ach shared memory file. In this example, I_0 points to a four byte message starting at D_1, and I_2 points to a one byte message starting at D_5. The next inserted message will use index cell I_2 and start at D_6. There are two free index cells and three free data bytes. Both arrays are circular and wrap around when the end is reached.

contains both an offset into the data array and the length of that element in the data array. A head offset into each array indicates both the place to insert the next data and the location of the most recent message frame. This pair of circular arrays allows us to find the variable sized message frames by first looking at a known offset in the fixed-sized index array.

B. Core Procedures

Two procedures compose the core of ach: `ach_put` and `ach_get` which we describe in pseudocode.

1) ach_put: The procedure `ach_put` inserts new messages into the channel. Its function 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, lines 2-5. 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, lines 6-10. 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, lines 11-16.
(4) Update the offset and free counts in the channel structure, lines 16-22.

2) ach_get: The procedure `ach_get` receives a message from the channel. Its function 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. 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.
(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.

```
Procedure achput (c,b,n)
  Input: c : ach channel ;          // shared memory file
  Input: b : byte array ;              // message buffer
  Input: n : integer ;             // length of message
  Output: status : integer ;             // status code
1  if $n > length(c.data\_array)$ then return $OVERFLOW$;
2  LOCK($c$); // take the mutex
   /* Get a index entry                               */
3  if $0 = c.index\_free$ then
4      $c.data\_free \mathrel{+}= c.index\_array[c.index\_head].size$;
5      $c.index\_free \leftarrow 1$;
   /* Make room in data array                         */
6  $i \leftarrow (c.index\_head + c.index\_free)\ \%\ c.index\_cnt$;
7  while $c.data\_free < n$ do
8      $c.data\_free \mathrel{+}= c.index\_array[i].size$;
9      $c.index\_free$ ++;
10     $i \leftarrow (i + 1)\ \%\ c.index\_cnt$;
   /* Copy Buffer                                     */
11 if $c.data\_size$ - $c.data\_head \geq n$ then
       /* Simple Copy                                 */
12     MEMCPY($c.data\_array + c.data\_head$, $b$, $n$);
13 else
       /* Wraparound Copy                             */
14     $e \leftarrow c.data\_size - c.data\_head$;
15     MEMCPY($c.data\_array + c.data\_head$, $b$, $e$);
16     MEMCPY($c.data\_array$, $b + e$, $n - e$);
   /* Modify Counts                                   */
17 $c.index\_array[c.index\_head].size = n$;
18 $c.index\_array[c.index\_head].offset = c.data\_head$;
19 $c.data\_head \leftarrow$
   $(c.data\_head + n)\ \%\ length(c.data\_array)$;
20 $c.data\_free \mathrel{-}= n$;
21 $c.index\_head \leftarrow (c.index\_head + 1)\ \%\ c.index\_cnt$;
22 $c.index\_free$ −−;
23 UNLOCK($c$); // release the mutex
24 NOTIFY($c$); // wake readers on cond. var.
25 return $OK$;
```

C. Formal Verification

We have formally verified the `ach_put` and `ach_get` procedures using the SPIN Model Checker [6]. SPIN models the operation of a computer program using the Promela language, which is based on the Guarded Command Language [4] and Communicating Sequential Processes [5]. Our model for Ach checks the consistency of channel data structures, ensures proper transmission of message data, and verifies freedom from deadlock. Because model checking enumerates all possible world states, we can verify these properties for all

Procedure achget (c,b,n,s,i,o_w,o_l)

```
Input: $c$ : ach channel ;   // shared memory file
Input: $b$ : byte array ;   // storage for message
Input: $n$ : integer ;                  // size of b
Input: $s$ : integer ;      // last seq. num. seen
Input: $i$ : integer ;       // next index to read
Input: $o_w$ : boolean ; // wait for new message?
Input: $o_l$ : boolean ;        // get newest msg.?
Output: integer $\times$ integer ;       // size, status
Output: $s$ : integer ;      // new last seq. num.
Output: $i$ : integer ;          // new next index
LOCK($c$); // take the mutex
if $c.seq\_num = s \wedge o_w$ then
    WAIT($c$); // condition variable wait
if $c.last\_seq = s \vee 0 = c.last\_seq$ then
    UNLOCK(c);
    return $(0 \times STALE)$; // no entries
/* Find index array offset, j        */
if $o_l$ then
    /* newest index                   */
    $j \leftarrow$
    $(c.index\_head + c.index\_cnt - 1)$ % $c.index\_cnt$;
else if $\neg o_l \wedge c.index\_array[i].seq\_num = s + 1$ then
    $j \leftarrow i$; // next index
else
    /* oldest index                   */
    $j \leftarrow (c.index\_head + c.index\_free)$ % $c.index\_cnt$;
/* Now read frame from data array    */
$x = c.index\_array[j]$;
if $x.size > n$ then
    UNLOCK(c);
    return $(x.size \times OVERFLOW)$;
if $x.offset + x.size < c.data\_size$ then
    MEMCPY($b$, $c.data\_array + x.offset$, $x.size$);
else
    $e = c.data\_size - x.offset$;
    MEMCPY($b$, $c.data\_array + x.offset$, $e$);
    MEMCPY($b + e$, $c.data\_array$, $x.size - e$);
$s' \leftarrow s$;
$s \leftarrow x.seq\_num$;
UNLOCK(c);
$i \leftarrow (i + 1)$ % $c.index\_cnt$;
if $x.seq\_num > s' + 1$ then
    return $(x.size \times MISSED)$;
else
    return $(x.size \times OK)$;
```

possible interleavings of ach_put and ach_get, something that is 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. Verification enhanced both the robustness and simplicity of Ach.

D. Other Design Considerations

An important consideration in the design of Ach is the idea of *Mechanism, not Policy* [12]. Ach provides a mechanism to move bytes between processes and a mechanism to notify callers should something go awry. It does not specify a policy for serializing arbitrary data structures or a policy for how to handle all types of errors. Such policies are application dependent and even within our own research group have changed across different applications and over time. Thus, by adopting the *mechanism* design approach, we maximize the flexibility and utility of our software.

IV. Discussion of Ach

Ach provides many advantages for real-time applications and some potential faults. Here, we summarize our observations.

A. Advantages of Ach

1) Formally Verified: Ach is formally verified: we have produced a Promela Modela of the core Ach functions and verified it using the SPIN model-checker. Users of Ach do not need to worry about the synchronization or data consistency issues which are all handled by the library.

2) No HOL Blocking: Ach never has HOL blocking; it can always give you the newest data. We can always compute the newest message in the channel in O(1) time. Any process which wants the latest data will always get it without having to look at older messages.

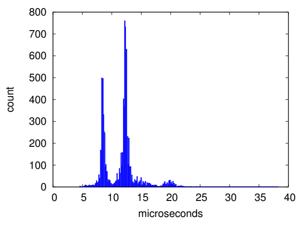
3) Read Older Data: Ach will give you older data as best it can. Any messages in the circular buffer which have not been overwritten can still be read. Each reader tracks the offset of the last message it read, so it can find its next message in O(1) time. If the next message it wanted has been overwritten, then we compute the oldest message in the buffer in O(1) time and give that to the reader instead.

4) Efficiency: Ach is algorithmically fast. Each read and write operation is O(n) where n is the number of bytes in the message.

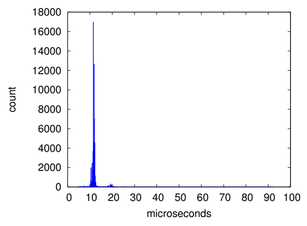
5) Multiple Senders and Receivers: Ach supports all combinations of communications between M senders and N receivers with only $M + N$ file descriptors. Typical communication methods open sockets between every reader and writer. They require $M \times N$ file descriptors. In Ach, we achieve $M \times N$ communication lines with only $M + N$ file descriptors. Instead of representing these communications explicitly, each Ach channel can have M readers and N writers. Reads and writes can be arbitrarily interleaved by all processes. Each process reading an Ach channel needs only to open one single file descriptor for that channel's shared memory area, resulting in $M + N$ file descriptors.

6) Priorities: Ach obeys real-time priorities. Each channel is protected by a mutex and condition variable, thus the kernel decides which process gets the next access to a given channel. The kernel decides based on process priority, and therefore higher priority processes gain first access to read and write to an Ach channel.

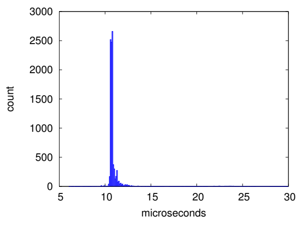
Additionally, Ach properly performs priority inheritance so that if, for example, the logger is reading some channel and



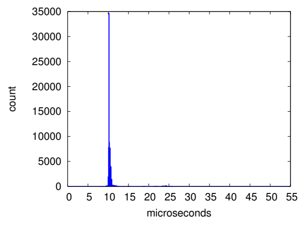
(a) daneel 1kHz



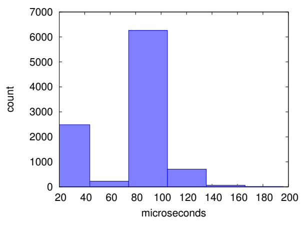
(b) daneel 8kHz



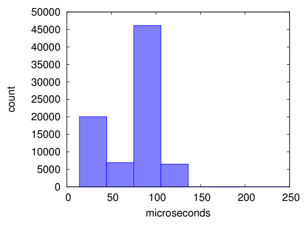
(c) daneel PREEMPT 1kHz



(d) daneel PREEMPT 8kHz



(e) nibbler 1kHz



(f) nibbler 8kHz

Fig. 3. Histograms of Ach messaging latencies. Daneel is a Core 2 Duo running Ubuntu 10.4. The PREEMPT version used the RT_PREEMPT kernel patch. Nibbler is an 800MHz ARM CPU running Debian 6.0.

the motor driver starts a read, the logger will temporarily run at the motor drivers priority until it exits the critical section surrounding the channel read.

7) Access Control: Because Ach is implemented on top of POSIX shared memory files, access to channels can be controlled via the unix permission bits. This allows channel access to be restricted on a per-user and per-group basis, though callers of `ach_get` will still need the write bit enabled in order to use the channel mutex and condition variable.

8) Portability: The Ach library is implimented in C using portable POSIX functions. It has been tested on GNU/Linux and MacOSX operating systems, and on IA-32, AMD64, and ARM CPUs.

9) Open Source: Ach is Open Source, available under a BSD-style license. This includes the formal model, benchmark code, and an example application.

B. Benchmark Results and Discussion

We provide benchmark results for Ach message latencies in Fig. 3.

1) Benchmark Platforms: The benchmark data was collected on the following platforms:

- daneel: Intel Core 2 Duo E7300, Ubuntu Linux 10.04 i386, Kernel 2.6.32-33-generic
- daneel PREEMPT: Intel Core 2 Duo E7300, Ubuntu Linux 10.04 i386, Kernel 2.6.31-11-rt
- nibbler: Qualcomm MSM7230 Snapdragon 800 MHz, Debian GNU/Linux 6.0 armel, Kernel 2.6.32.28-cyanogenmod-g4f4ee2e

2) Benchmark Procedure and Results: The benchmark application performs the following steps.

1) Create and open an Ach channel
2) Fork a receiver process
3) Parent: post timestamped messages to the Ach channel at the desired frequency
4) Child: Receive Ach messages from the channel and compute the delay based on the message timestamp.

The plots in Fig. 3 show the message latencies for each of the tested configurations. The daneel and daneel PREEMPT configurations ran on a IA-32 CPU. For both these configurations and at both frequencies, we see a typical latency of $10\mu\text{s}$. The worst case latency at 1kHz for these systems was $30 - 40\mu\text{s}$. For the standard kernel, worst case latency at 8kHz was much worse, totaling $100\mu\text{s}$. By switching the fully preemptible Linux kernel, we reduced that latency to $50\mu\text{s}$. Nibbler, which used a much slower ARM CPU, had a typical latency of $100\mu\text{s}$ with worst case latency of $200 - 250\mu\text{s}$. These results show the latency imposed by Ach still allows us to operate robots at our desired rate of 1kHz.

C. Potential Error Modes

There are two theoretical failure modes we are aware of which may arise with Ach. However, in our three years of active Ach use, neither of these failure modes have occurred in practice. They exist as theoretical vulnerabilities and we mention them as areas for potential improvement.

1) Deadlock: Use of a mutex may result in deadlock if reader or write dies, ie. with a `kill -9`, while holding the lock. This may be mitigated by the use of robust POSIX mutexes which will detect this condition. Additional code could be added which would either reverse an interrupted write or pass through on an interrupted read as reads do not modify the channel data structures.

2) Corruption: Because all processes accessing the channel must have read/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. To make Ach truly impervious to this failure mode, though, would likely require moving the channels into the kernel.

V. Conclusions and Future Work

We have presented Ach, a new IPC mechanism specially suited to real-time systems that sample physical processes. Compared to standard POSIX IPC and other robotics middleware [13], [11], [8], [9], [1], Ach provides unique message-passing semantics which always allow the latest data sample to be read. The algorithms and data structures are formally verified, increasing both the robustness and simplicity of our implementation. Ach has been demonstrated to work effectively for a variety of robot control applications over three years.

There remain a number of ways to improve the performance and robustness of Ach. One shortcoming which may particularly affect some is that Ach focuses on efficient communication between proceses on a single host; network communication is not addressed in any reasonable fashion. It would be appropriate and desirable to pair Ach with some suitable network transport such as SCTP or RDS. The synchronization used by Ach is very simple and could undoubtedly be improved, though care would need to be taken to maintain proper priority inheritance. It should be possible to `mmap` the data array twice sequentially into the process address space to eliminate the double `memcpy` in the Ach procedures. This would, more importantly, allow serialization formats such as XDR and Protocol Buffers, to serialize directly to and from the Ach channel, eliminating a redundant copy operation. Finally, to maximize robustness against corruption, it may be appropriate to move the channels into the kernel.

Ach and sample code can be downloaded at http://www.golems.org/node/1526. By providing this IPC open source to the science and engineering community we hope that it will be a useful tool to expedite the development of new robust systems such as robot platforms. We aim to continue improving the efficiency, robustness, and generality of Ach.

References

[1] Agüero, C., Cañas, J., Martín, F., and Perdices, E. Behavior-based iterative component architecture for soccer applications with the nao humanoid. In *5th Workshop on Humanoids Soccer Robots. Nashville, TN, USA* (2010).

[2] Dantam, N., Kolhe, P., and Stilman, M. The motion grammar for physical human-robot games. In *IEEE Intl. Conf. on Robotics and Automation* (2011), IEEE.

[3] Dantam, N., and Stilman, M. The motion grammar: Linguistic planning and control. In *Robotics: Science and Systems* (2011), IEEE.

[4] Dijkstra, E. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*, 8 (1975), 453–457.

[5] Hoare, C. Communicating sequential processes. *Communications of the ACM 21*, 8 (1978), 666–677.

[6] Holtzman, G. *The Spin Model Checker*. Addison Wesley, Boston, MA, 2004.

[7] The IEEE and The Open Group. *IEEE Std 1003.1-2008*, 2008. http://pubs.opengroup.org/onlinepubs/9699919799/.

[8] iRobot Corporation. Aware 2.0. http://www.irobot.com/gi/developers/Aware/.

[9] Microsoft Corporation. Microsoft robotics studio. http://www.microsoft.com/robotics/.

[10] The Object Management Group. *Data Distribution Service for Real-time Systems*, 1.2 ed., January 2007. http://www.omg.org/spec/DDS/1.2/.

[11] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics* (Kobe, Japan, May 2009).

[12] Scheifler, R., Carver, D., Gerovac, B., Gettys, J., Karlton, P., McGregor, S., Rao, R., Sun, D., Winchell, D., Angebrandt, S., et al. X window system protocol, version 11. *Network Working Group RFC 1013* (1987).

[13] Stevens, W. R., and Rago, S. A. *Advanced Programming in the UNIX Environment*, 2 ed. Addison Wesley, Boston, MA, 2005.

[14] Stilman, M., Olson, J., and Gloss, W. Golem krang: Dynamically stable humanoid robot for mobile manipulation. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on* (2010), IEEE, pp. 3304–3309.