2012 IEEE-RAS International Conference on Humanoid Robots

Robust and Efficient Communication for Real-Time Multi-Process Robot Software

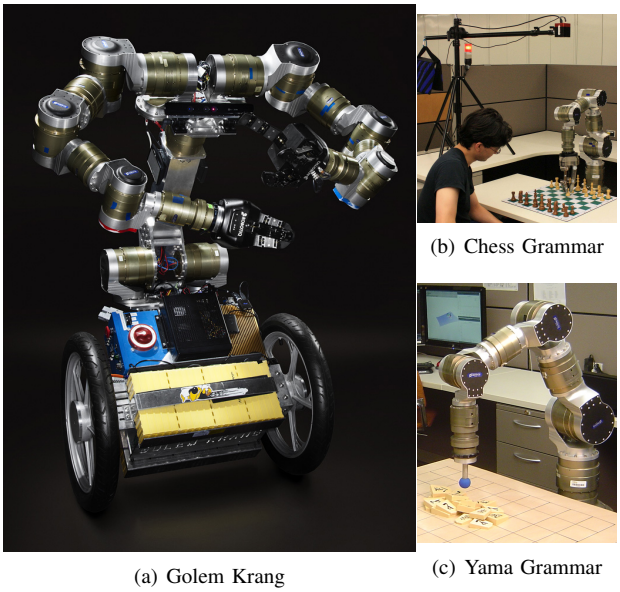
Neil Dantam Mike Stilman

***Abstract*—We present a new Interprocess Communication (IPC) mechanism and library. Ach is uniquely suited for coordinating drivers, controllers, and algorithms in complex robotic systems such as humanoid robots. 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, and we discuss the implementation on our humanoid robot Golem Krang. Finally, the source code for Ach is available under an Open Source permissive license.**

I. Introduction

Correct real-time software is critical to the safe and reliable operation of complex robotic systems such as humanoid robots. These systems depend on software for dynamic balance, object manipulation, navigation, and even seemly innocuous tasks such as safe regulation of battery voltage. A multi-process software design increases robustness by isolating errors to a single process, allowing the rest of the system to continue operating and reducing the incidence of catastrophic failure. This approach additionally assists with modularity and concurrency. However, traditional methods of Interprocess Communication (IPC) are not well suited to robotics applications. In particular, standard POSIX IPC mechanisms such as pipes favor older data over newer and can block on or drop newer messages. In a real-time control loop, we always prefer to have the most recent data sample. In addition, it is critical to minimize message latency for real-time tasks such as dynamic balance and force control of manipulators. To address these concerns and produce robust control software for our humanoid robot Golem Krang, Fig. 1(a), we introduce the *Ach*[1] Interprocess Communication (IPC) library which enables efficient multi-process real-time control, is more suited to robotics applications than typical POSIX IPC, and is formally verified to ensure correctness.

There are three goals and assumptions that guide the design of our control software and the Ach library. 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) [1]. 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 humanoid robots, we must make our system robust. Therefore, we adopt a multiple process approach as more robust than a single-process or multi-threaded application. This implies sampled data must be passed between OS processes using some form of Interprocess Communication (IPC). Finally, we favor Open Source Software since it maximizes flexibility and control in development. This is important both in research and for 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 Grammar (c) Yama Grammar

Fig. 1. Robotic Systems where Ach provided all communications between hardware drivers, perception, planning, and control algorithms. [2], [3], [4].

POSIX provides a rich variety of IPC mechanisms, but none of them fully meet our requirements. An overview of these mechanisms is given in [5]. The fundamental difference is that as soon as a new sample of the signal is produced, nearly everything no longer cares about older samples. 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, synchronization of shared memory is a difficult programming problem, making the typical and direct use of POSIX shared memory unfavorable for developing robust systems. Furthermore, some parts of the system, such as logging, may need to access older samples, so this also should be permitted at least on a best-effort basis. Since no existing standardized and open source implementation satisfied our requirements

The authors are with the Robotics and Intelligent Machines Center in the Department of Interactive Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA. 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."

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 robots. This library, called *Ach*, provides a message-bus or publish-subscribe communication semantics – similar to other real-time middleware and robotics programming systems [6], [7], [8], [9] – but with the distinguishing feature of favoring newer data over old. Ach provides certain advantages making it suitable for real-time control of physical robotic 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.

Ach has been used to implement communication for several robotic systems, shown in Fig. 1 and the video attachment to this paper [2], [3], [4]. The performance of these systems verify that the context-switch overhead of a multi-process real-time application is acceptable for robot control, including manipulation and dynamic balancing. Our experience in developing these systems confirms the benefits in robustness and flexibility from a multi-process application.

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 these general-purpose IPC mechanisms are not ideal for real-time robot control. A thorough survey of POSIX IPC is provided in [5].

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, to prevent blocking of the reading or writing process, we must resort to 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 is that it is possible to block if the queue fills up. Consider a process that gets stuck and stops processing its message queue. When it starts again, the process must still read/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.

The data structure which Ach most closely resembles is the circular array. Circular arrays or ring buffers are common data stuctures in device drivers and real-time programs, and the implimentation in Ach provides unique features to satisfy our requirements for a multi-process real-time system. Primarily, typical circular buffers allow only one producer and one consumer with the view that the producer inserts data and the consumer removes it. The Bip Buffer [10] is an efficient circular buffer that minimizes copying, but it is still designed around a single producer and consumer. The MCRingBuffer [11] is a cache-efficient design, but it again focuses on the single producer and consumer model. Our system has multiple producers and multiple consumers writing and reading a single sequence of messages. A message reader cannot *remove* a message, because some other process may still need to read it. Because of this different design requirement, we developed our own data structure and algorithm for real-time IPC among multiple processes.

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. General, Real-Time and, Robotics Middleware

In addition to the core POSIX IPC mechanisms, there exist various messaging middlewares and robot software architectures. However, these are either not Open Source or not suitable for our multi-process real-time domain.

The Message Passing Interface (MPI) [12] is ubiquitous in high-performance computing, but its focus is on maximizing message throughput for networked clusters. Our domain centers around minimizing sample latency on a single host.

The Advanced Message Queuing Protocol (AMPQ) [13] is a network message distribution middleware focused on business applications; it does not address low-latency real-time systems. ZeroMQ [14] provides IPC based on TCP and local-domain sockets which have the HOL blocking condition. Remote Procedure Call (RPC) methods such as ONC RPC [15] and CORBA [16] allow synchronous point-to-point communication but they do not directly allow efficient communication between multiple senders and receivers and also do not address HOL blocking. In contrast, Data Distribution Service [6] is a publish-subscribe network protocol which may be complementary to the efficient and formally verified IPC we present here.

The Orocos Real-Time Toolkit [17] and NAOqi [18] are two architectures for robot control, but they do not meet our requirements for flexible IPC. Aware2.0 [8] is not open source. Microsoft Robotics Studio is not open source and does not run on POSIX systems [9]. The RT-Middleware [19] framework is based on CORBA, which does not provide a suitably lightweight IPC mechanism. ROS [7] provides open source TCP and UDP message transports, which suffer from the aforementioned HOL blocking problem. Rosbridge [20] is an HTTP and Javascript interface to ROS messaging. This convenience is appropriate for certain use cases but is unsuitable for high-speed real-time control. In conclusion, none of these middlewares met our needs for an open source, light-weight, and non-HOL blocking IPC.

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 channel counters
- `ach_put`: Insert a new message into the channel
- `ach_get`: Receive a message from the channel
- `ach_close`: Close the shared memory file

Channels must be created before they can be opened. Creation may be done directly by either the reading or writing process, or it may be done via the shell command, `ach -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. After the channel is created, each reader or writer must open the channel before it can get or put messages.

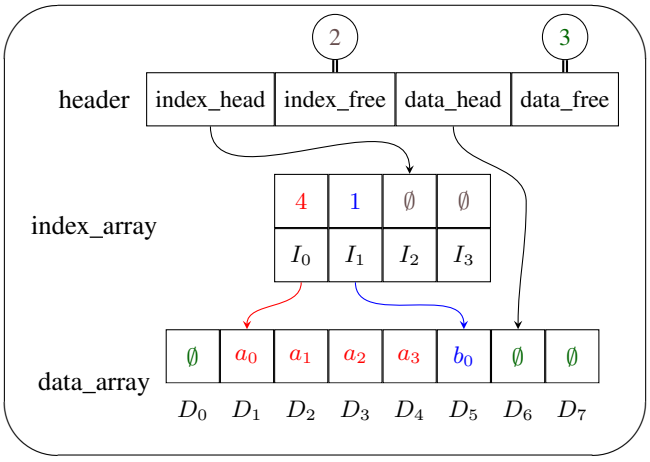


Fig. 2. Logical Memory Structure for an Ach shared memory file. In this example, I_0 points to a four byte message starting at D_1, and I_1 points to a one byte message starting at D_5. The next inserted message will use index cell I_2 and start at D_6. There are two free index cells and three free data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

A. Channel Data 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 contains both an offset into the data array and the length of that element. 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.

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

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.

Procedure ach_put

```
   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 += c.index_array[c.index_head].size;
5  |  c.index_free ← 1;
   /* Make room in data array                            */
6  i ← (c.index_head + c.index_free) % c.index_cnt;
7  while c.data_free < n do
8  |  c.data_free += c.index_array[i].size;
9  |  c.index_free ++;
10 |  i ← (i + 1) % c.index_cnt;
   /* Copy Buffer                                        */
11 if c.data_size - c.data_head ≥ n then
   |  /* Simple Copy                                     */
12 |  MEMCPY(c.data_array + c.data_head, b, n);
13 else
   |  /* Wraparound Copy                                 */
14 |  e ← 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 ←
   (c.data_head + n) % length(c.data_array);
20 c.data_free −= n;
21 c.index_head ← (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;
```

(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, lines 1-3. Otherwise, if there are no new messages, return a status code indicating this fact, lines 4-6.
(2) Find the index entry to use, lines 7-12. 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, lines 13-22.
(4) Update the sequence number count and next index entry offset for this receiver, lines 23-25.

Procedure ach_get

```
   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 × integer ;          // size, status
   Output: s : integer ;          // new last seq. num.
   Output: i : integer ;             // new next index
1  LOCK(c); // take the mutex
2  if c.seq_num = s ∧ o_w then
3  |  WAIT(c); // condition variable wait
4  if c.last_seq = s ∨ 0 = c.last_seq then
5  |  UNLOCK(c);
6  |  return (0 × STALE); // no entries
   /* Find index array offset, j                         */
7  if o_l then
   |  /* newest index                                    */
8  |  j ← (c.index_head + c.index_cnt − 1) % c.index_cnt;
9  else if ¬o_l ∧ c.index_array[i].seq_num = s + 1 then
10 |  j ← i; // next index
11 else
   |  /* oldest index                                    */
12 |  j ← (c.index_head + c.index_free) % c.index_cnt;
   /* Now read frame from data array                     */
13 x = c.index_array[j];
14 if x.size > n then
15 |  UNLOCK(c);
16 |  return (x.size × OVERFLOW);
17 if x.offset + x.size < c.data_size then
18 |  MEMCPY(b, c.data_array + x.offset, x.size);
19 else
20 |  e = c.data_size − x.offset;
21 |  MEMCPY(b, c.data_array + x.offset, e);
22 |  MEMCPY(b + e, c.data_array, x.size − e);
23 s' ← s;
24 s ← x.seq_num;
25 UNLOCK(c);
26 i ← (i + 1) % c.index_cnt;
27 if x.seq_num > s' + 1 then
28 |  return (x.size × MISSED);
29 else
30 |  return (x.size × OK);
```

IV. Case Study: Golem Krang

Golem Krang [4] is a dynamically balancing bi-manual mobile manipulator designed and built at the Georgia Tech Humanoid Robotics Lab. All the real-time control for Krang is implemented through the Ach IPC library. This approach has produced software that is both robust and modular, minimizing system failures and allowing significant code reuse both within Krang with other projects [2], [3] sharing the same hardware components.

The electronic components of Golem Krang are summarized in the block diagram of Fig. 3. The real-time control software runs on the Pentium-M Control PC under Ubuntu Linux. Krang also contains a secondary Intel i7 PC for

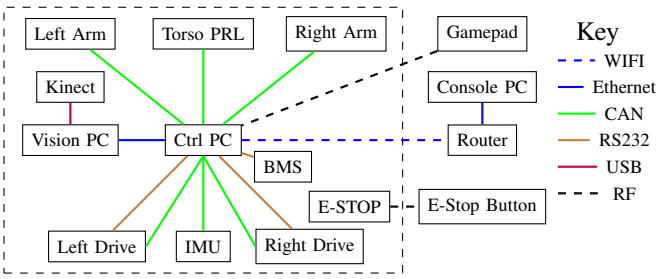


Fig. 3. Block diagram of electronic components on Golem Krang. Blocks inside the dashed line are onboard and blocks outside are offboard.

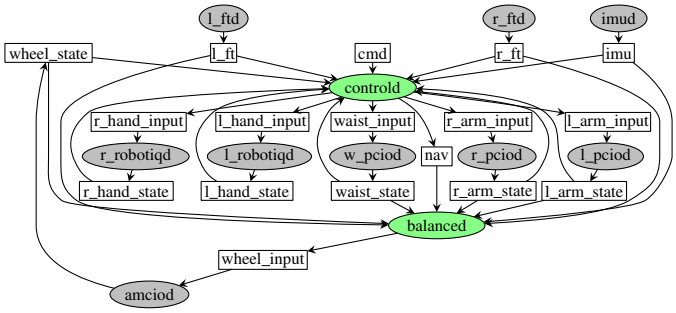


Fig. 4. Block diagram of primary software components on Golem Krang. Gray ovals are user-space driver processes, green ovals are controller processes, and rectangles are Ach channels.

vision processing which runs outside of the real-time control system discussed in this paper. The Control PC communicates over eight Controller Area Network buses to embedded microcontrollers for the wheels, Inertial Measurement Unit (IMU), torso, and arms. Each wheel is controlled by an AMC servocontroller. The torso is actuated using three Schunk PRL motor modules. Each arm is a Schunk LWA3 with a wrist-mounted ATI force-torque sensor and Robotiq Adaptive Gripper. The battery management system (BMS) monitors voltage of the lithium battery cells. To achieve dynamic balance and manipulation, software on the Control PC must gather state updates from the arms, wheels, and IMU, then compute the inputs for arms and wheels, all at the desired rate of one kilohertz.

The software for Krang is implemented as a collection of processes communicating over Ach channels, Fig. 4. Each hardware device, such as the IMU or an LWA3 arm, is managed by a userspace driver daemon. These drivers are independent operating system processes that initialize the hardware, then read state and send commanded control inputs to hardware device. These state and input messages flow through a variety of ach channels. State messages include values such as positions, velocities, and forces. Input messages include values such as reference velocities or voltages. Each driver daemon has one channel for the device state messages, ensuring that the current state of the robot can always be accessed as the newest message in each of these channels. Devices that take an input command have a second channel for that input. In addition to the driver daemons, there are two controller daemon processes. The `balanced` process implements the stable balancing controller for Krang. Control of the arms and reference forward and rotational speeds are handled in the separate `controld` process. Dividing these controllers promotes robustness by isolating the highly-critical balance control from other faults. This collection of driver and controller daemons communicating over Ach channels implements the real-time, kilohertz control loop for Golem Krang.

This design produces a system that is efficient, modular, and robust. The low overhead and suitable semantics of Ach communication permits real-time control under Linux using multiple processes. In several cases, Krang contains multiple identical hardware devices. The message-passing, multi-process design aids code reuse by allowing access to duplicated devices with multiple instances of the same daemon binary – two instances of the ftd daemon for the F/T sensors, two instances of the robotiqd daemon for the grippers, and three instances of the pciod daemon for two arms and torso. The relative independence of each running process makes this system robust to failures in non-critical components. For example, an electrical failure in a waist motor may stall the w_pciod process, but the balanced controller and amciod driver daemons continue running independently, ensuring that the robot does not fall. Thus, Ach helps enhance the safety of this potentially dangerous robotic system.

V. Verification, Benchmarks, and Discussion

A. Formal Verification

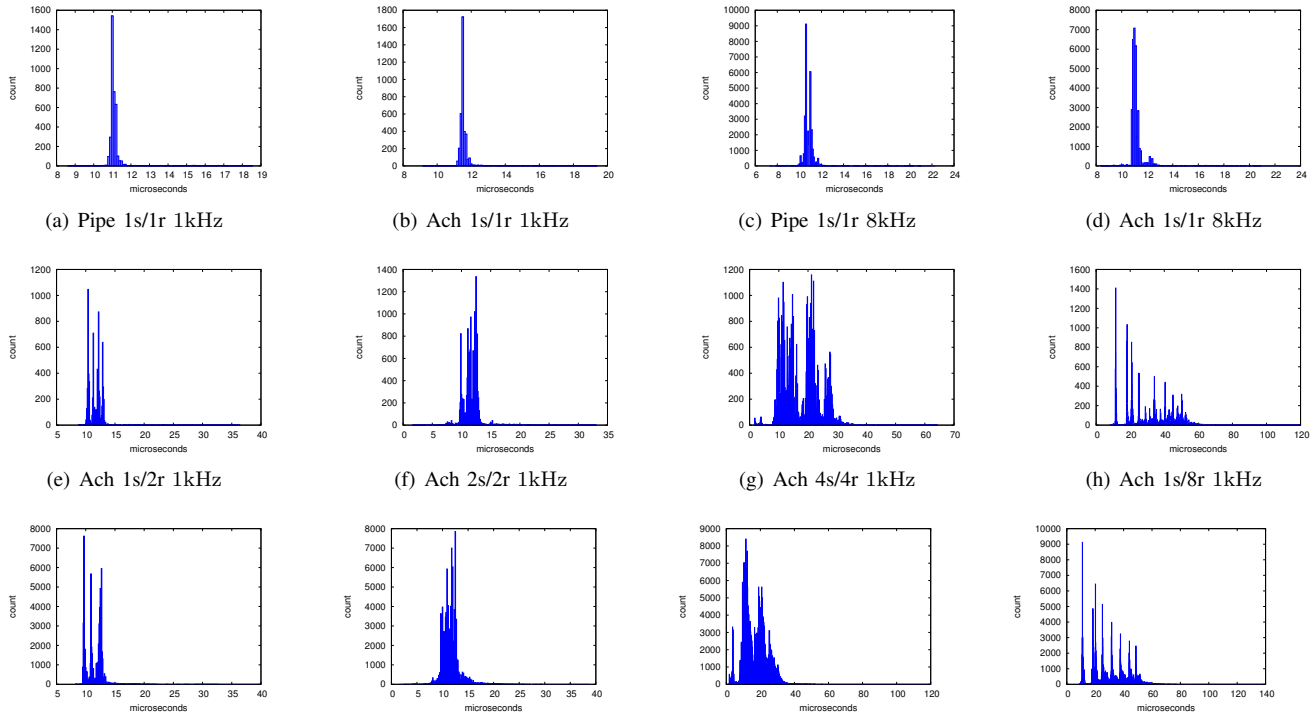
We used the SPIN Model Checker [21] to formally verify Ach. Formal verification is a method to enhance the reliability of software by first *modeling* the operation of that software and then *checking* that the model adheres to some specification for performance [22]. SPIN models the operation of a computer program using the Promela language, which is based on the Guarded Command Language [23] and Communicating Sequential Processes [24]. SPIN then enumerates all possible world states of that model and ensures that each state satisfies the given specification.

We verified the `ach_put` and `ach_get` procedures using SPIN. Our model for Ach checks the consistency of channel data structures, ensures proper transmission of message data, and verifies freedom from deadlock. Because model checking enumerates all possible world states, we can verify these properties for all 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.

B. Benchmarks

We provide benchmark results for Ach message latencies in Fig. 5 and include latencies for message sent overs POSIX pipes as a comparison.

1) Benchmark Procedure: We perform the benchmarks on an Intel Core 2 Duo E7300 running Ubuntu Linux 10.04 i386, Kernel 2.6.31-11-rt. The benchmark application performs the following steps.



(a) Pipe 1s/1r 1kHz (b) Ach 1s/1r 1kHz (c) Pipe 1s/1r 8kHz (d) Ach 1s/1r 8kHz

(e) Ach 1s/2r 1kHz (f) Ach 2s/2r 1kHz (g) Ach 4s/4r 1kHz (h) Ach 1s/8r 1kHz

(i) Ach 1s/2r 8kHz (j) Ach 2s/2r 8kHz (k) Ach 4s/4r 8kHz (l) Ach 1s/8r 8kHz

Fig. 5. Histograms of Ach and Pipe messaging latencies. Benchmarking performed on a Core 2 Duo running Ubuntu Linux 10.04 with PREEMPT kernel. The labels αs/βr indicate a test run with α sending processes and β receiving processes.

1) Create and open an Ach channel
2) Fork one or more receiver processes
3) Fork one or more sender processes
4) Senders: Post timestamped messages at the desired frequency
5) Receivers: Receive messages and record latency of each messaged based on the timestamp.

We repeat this procedure, varying the frequency and number of senders and receivers. The benchmark code is included with the Ach source distribution.

2) Benchmark Results: Our benchmark results in Fig. 5 show that Ach first matches the performance of POSIX pipes for the single sender/receiver case while also providing non-HOL-blocking semantics, and then additionally scales linearly to multiple senders and receivers. The first row of Fig. 5 shows essentially identical performance between POSIX pipes and single sender and receiver Ach. This is expected because the majority of latency should come from the process context-switch which must occur with both pipes and Ach. This also indicates that the cost of the context switch is significantly greater than the cost of the data copy for the small messages sizes typical of real-time applications. The next two rows show Ach performance for multiple senders and receivers. The additional processes increase latency because channel access is restricted to one process at a time. This serialization of access gives a linear increase in the worst-case latency, resulting in 20μs per receiver worst-case latency increase for the 1kHz rate on our benchmark platform. These results show that the latency imposed by Ach still allows us to operate robots at our desired rate of 1kHz

C. Discussion

An important consideration in the design of Ach is the idea of *Mechanism, not Policy* [25]. 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.

There is a trade-off between single-process, multi-process, and multi-threaded approaches that influenced our choice of a multi-process system design and motivated the development of Ach. Software components in a single process can communicate with a function call whereas components in different kernel threads or different processes require a CPU context-switch which is orders of magnitude slower. The context-switch cost bounds the granularity at which real-time components may be divided between threads or processes. On the other hand, when the application can be parallelized, multiple threads and processes permit true concurrency, a crucial performance benefit on modern multi-core CPUs. Multi-threaded approaches generally provide a slight performance advantage over multi-process programs, and this advantage may be more substantial if data can be cleverly shared between the threads. However, the synchronization of multi-threaded programs is a notoriously difficult task.

New formal verification methods applied to even mature real-world multi-threaded programs often find numerous programming errors [26], [27]. In comparison, multi-process programs cannot have these memory consistency errors. Furthermore, multi-process programs are inherently robust against failures in individual software components, as each process can be stopped, started, and modified independently. Thus, while none of these approaches are universally ideal, the multi-process design we have adopted does have key benefits in concurrency and robustness. However, the lack of appropriate real-time IPC has previously made development of multi-process real-time applications difficult. We address this challenge with Ach.

Networked real-time systems are another related area. However, network protocols pose a different set of requirements and challenges from inter-process communication. Processes on a single host can access a single physical memory which provides high bandwidth and assumed perfect reliability; still, care must be taken to ensure consistency of the memory 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, and clock skew. These differences in requirements imply that different mechanisms should be used to implement IPC and network communication. Thus, we intend Ach to be both complementary to and compatible with networked communication.

VI. Conclusions

We have presented Ach, a new IPC mechanism for real-time robotic systems. Compared to standard POSIX IPC and other robotics middleware [5], [7], [8], [18], 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 the implementation. Ach has been validated in the core of a variety of robot control applications for over three years and has enabled development of efficient and reliable control software for our robot Golem Krang. In addition, we are collaborating with the Drexel Autonomous Systems Lab to implement Linux control software for the KAIST Hubo using Ach.

The Ach library and sample code can be downloaded at http://www.golems.org/node/1526. We hope that this open source IPC will be a useful tool to expedite the development of new robust robotic systems.

References

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

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

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

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

[5] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment*. Addison Wesley, Boston, MA, 2 edition, 2005.

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

[7] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. 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.

[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] Simon Cooke. The bip buffer - the circular buffer with a twist, May 2003. http://www.codeproject.com/KB/IP/bipbuffer.aspx.

[11] P.P.C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 78–79. ACM, 2009.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message passing interface*. MIT Press, Cambridge, MA, 1999.

[13] S. Vinoski. Advanced message queuing protocol. *Internet Computing, IEEE*, 10(6):87–89, 2006.

[14] Martin Sústrik. *∅MQ: The Theoretical Foundation*, July 2011. http://www.250bpm.com/concepts.

[15] R. Srinivasan. *RPC: Remote Procedure Call Protocol Specification Version 2*. Internet Engineering Task Force, August 1995. http://www.ietf.org/rfc/rfc1831.txt.

[16] The Object Management Group. *Common Object Request Broker Architecture (CORBA/IIOP)*, 3.1.1 edition, August 2011. http://www.omg.org/spec/CORBA/3.1.1/.

[17] H. Bruyninckx, P. Soetens, and B. Koninckx. The real-time motion control core of the orocos project. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 2, pages 2766–2771. IEEE, 2003.

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

[19] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.K. Yoon. Rt-middleware: distributed component middleware for rt (robot technology). In *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 3933–3938. IEEE, 2005.

[20] Christopher Crick, Graylin Trevor Jay, Sarah Osentoski, Benjamin Pitzer, and Odest Chadwicke Jenkins. rosbridge: Ros for non-ros users. In *International Symposium on Robotics Research (ISRR 2011)*, Flagstaff, AZ, USA, August 2011.

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

[22] C. Baier, J.P. Katoen, et al. *Principles of Model Checking*. MIT Press, Cambridge, MA, 2008.

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

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

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

[26] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 267–280. USENIX Association, 2008.

[27] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 51–64. ACM, 2011.