

Multi-Process Control Software for HUBO2 Plus Robot

M.X. Grey[†], Neil Dantam[†], Daniel M. Lofaro[‡],
Aaron Bobick[†], Magnus Egerstedt[†], Paul Oh[‡], Mike Stilman[†]

Abstract—Humanoid robots require greater software reliability than traditional mechatronic systems if they are to perform useful tasks in typical human-oriented environments. This paper covers a software architecture which distributes the load of computation and control tasks over multiple processes, enabling fail-safes within the software. These fail-safes ensure that unexpected crashes or latency do not produce damaging behavior in the robot. The distribution also offers benefits for future software development by making the architecture modular and extensible. Utilizing a low-latency inter-process communication protocol (Ach), processes are able to communicate with high control frequencies. The key motivation of this software architecture is to provide a practical framework for safe and reliable humanoid robot software development. The authors test and verify this framework on a HUBO2 Plus humanoid robot.

I. INTRODUCTION

Due to the critical role of software in robot operation, a great deal of consideration needs to be put into the underlying design of a robot's software architecture. The simplest approach to developing robot software is to integrate all functionality, from hardware drivers to planning algorithms, into a single executable program. This single-process approach has the benefit of being free from inter-process communication latency, which might be good for performance depending on the application. It also simplifies the design considerations required on the part of the programmer which can make the initial stages of software development easier.

Despite these benefits, the single-process design increases the possibility critical system failure by making all operations interdependent. For example, a single segmentation fault in a single component in the software would immediately crash all operations on the robot. For a humanoid, which must constantly maintain dynamic balance and control, this behavior is unacceptable. If a planning algorithm or a perception process crashes or stalls for any reason, this should not result in the robot losing stability and damaging itself. Rather, the robot should maintain stationary balance while it waits for the higher level processes to restart.

The software architecture described in this paper addresses this concern and more. It improves software reliability by distributing functionality between independent processes (daemons) which use an efficient and flexible communication interface called Ach. Before describing how these are used, we will introduce basic concepts in multi-process operation and inter-process communication, and then describe the particular robotic platform into which this design has been integrated.

[†]Authors are affiliated with Georgia Institute of Technology, Atlanta, GA, 30332, USA. {mxgrey, ndt, magnus}@gatech.edu and {afb, mstilman}@cc.gatech.edu.

[‡]Author is affiliated with Drexel University, Philadelphia, PA, 19104, USA. paul@coe.drexel.edu and dan@danLofaro.com.

A. Real-Time Robot Control with Multiple Processes

In general, a process is simply a self-contained computer program which runs on an operating system. Processes do not necessarily require any interaction from the user (i.e. they can run invisibly in the background of the operating system), and these particular processes are commonly referred to as *daemons*. The processes discussed in this paper are *daemonized*, so they run in the background of the operating system and manage themselves, without requiring user interaction. They also save output or error messages to logs which can be viewed at any time.

The ability to perform real-time tasks is critical in robot applications which require dynamic balancing or quick response time. For a multi-process architecture, this means that certain processes need a higher *priority* to ensure that they perform their tasks at a consistent frequency which corresponds to their real-time needs. This means that the operating system will temporarily stop lower priority tasks to let the real-time process run whenever needed. A higher priority process always runs first, ensuring that the computer's resources are focused on handling the most important tasks.

Using a multi-process approach provides modularity in the software architecture. Stable and previously developed processes can remain untouched even as new features are added into the software. For example, a process which performs motor control calculations would not need to be modified in any way in order to implement a new planning or perception algorithm, because these new features could simply go into their own process. This makes the architecture extensible, allowing for stability and consistency as software development progresses in the future.

B. Inter-process Communication & Ach

Multi-process systems require Inter-process Communication (IPC). Robotic systems have particular needs in this regard, which differ from those of general-purpose computing systems. General purpose IPC such as pipes and sockets favor older data over newer and can block or drop newer messages (known as Head-of-line Blocking). 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 the special needs of real-time systems and produce reliable control software for our robots we use *Ach*¹ IPC library, which enables efficient multi-process real-time control, is more suited to robotics applications than traditional IPC mechanisms, and is formally verified to ensure correctness [1].

¹Ach is available at <http://www.golems.org/node/1526>

Ach provides a publish-subscribe or message-bus interface for multiple processes simultaneously. Typically, one process publishes information to an Ach channel while other processes read from the channel as needed. ROS [2] inter-process communication is not suitable for our needs because it does not operate fast enough to satisfy our real-time application and suffers from the aforementioned Head-of-line Blocking. OROCOS [3] and NAOqi [4] are not used in our architecture because they do not provide the necessary multi-process publish/subscribe system.

In addition to excellent local inter-process communication performance, Ach also enables networked communication. A remote computer can push messages to an Ach channel on the robot's on-board computer. This means that daemons which require extra processing power, such as the Planning Daemon, can be run on an external high-performance computer and their output transferred via TCP or UDP to an Ach channel in HUBO's on-board computer. This feature can also be used in reverse where external computers can pull messages from HUBO's on-board Ach channels. This networking feature of Ach provides seamless integration between on-board real-time processes, such as hardware and control daemons, and off-board non-realtime processes, such as planners and loggers.

C. HUBO2 Plus

The particular software design described in this paper has been implemented on a HUBO2 Plus humanoid robot. HUBO2 Plus is a 130 cm (4' 3") tall, 42 kg (93 lb) full-size humanoid robot commonly referred to as HUBO. It was designed and constructed by Prof Jun-ho Oh at the HUBO Lab in the Korean Advanced Institute of Science and Technology (KAIST) [5]. HUBO is anthropomorphic, meaning it has 2 arms, 2 legs and a head. There are 6 degrees of freedom (DOF) in each leg, 6 in each arm, 5 in each hand, 3 in the neck, and 1 in the waist; all totalling 38 DOF. All of the major joints are high gain PD position controlled with the exception of the fingers. The fingers are open-loop PWM controlled. The sensing capability consists of a three axis force-torque (FT) sensor on each leg between the end of the ankle and the foot as well as between where the arm connects to the hand. Additionally it has an inertial measurement unit (IMU) at the center of mass and accelerometers on each foot. The reference commands for all of the joints are sent from the primary control computer (x86) to the individual motor controllers via two Controller Area Network (CAN) buses. There are currently eight HUBO's functioning in the United States as of December 2012. Four reside at Drexel University and one at Georgia Tech, Purdue, Ohio State, and MIT. Jaemi HUBO is the oldest of the HUBOs in America and has been at the Drexel Autonomous Systems Lab² (DASL) since 2008 [6]. Fig. 1 shows the major dimensions of HUBO.

II. HUBO CONTROL DESIGN

This section will describe the particular software design which has been (and is continuing to be) developed for the HUBO platform. It will discuss the hierarchy of daemons, their role in the operation of the robot, and finally how a user or programmer can interact with the daemons as well as develop

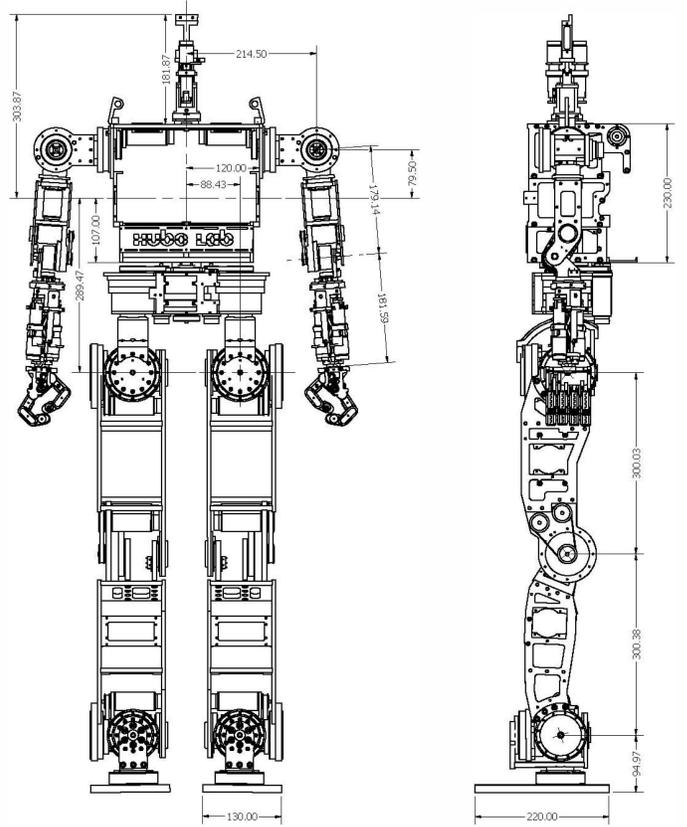


Fig. 1. HUBO2 Plus platform: 38 DOF, 130 cm tall full-size humanoid robot weighing 37 kg.

daemons of their own (for purposes such as planning and perception).

A. Daemon & Communication Structure

In this design, daemons are placed into a hierarchy based on how critical or how low-level their functionality is, as seen in Figure 2. The lower-level daemons run with a higher process priority, ensuring that when they need to run, they are not interrupted by a less important process, such as the user interface. Each box in Figure 2 represents an independent process, all of which are daemons running the background, and each column represents a set of equal priority processes.

In this design, commands flow cleanly from the left-most tier (planning and perception) to the right-most tier (the hardware interface), and then feedback data flows back over to all processes. Each arrow signifies an independent Ach channel. Solid lines represent command channels while the dotted lines represent a feedback channel. Notice that all the dotted lines fork off of a single arrow; this intentionally represents that all feedback is contained on a single channel which all daemons read from simultaneously (except for the CAN/Hardware Daemon which exclusively publishes to it).

The left-most arrow represents the user interface. This is where the user sends high-level instructions or commands into the system. These instructions are passed into the Planning Daemon which determines the how to fulfill the user's instructions. The Planning Daemon takes in pre-processed

²Drexel Autonomous Systems Lab: <http://dasl.mem.drexel.edu/>

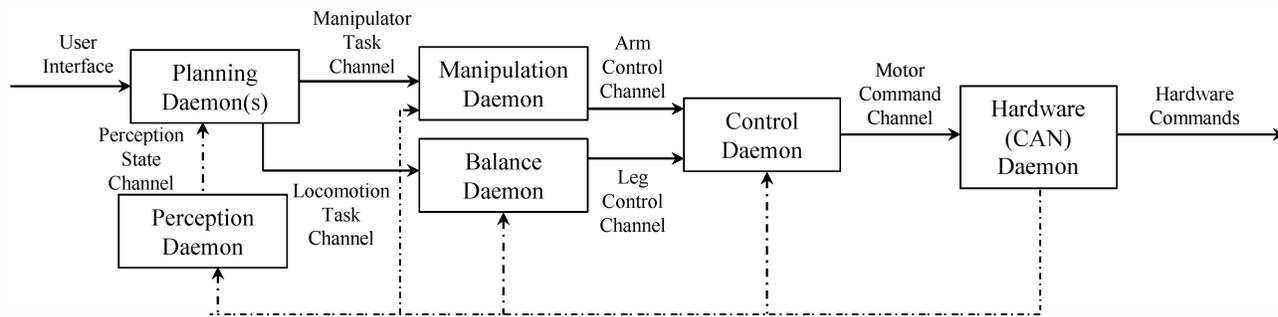


Fig. 2. Daemon Communication Flow Chart

information from the Perception Daemon to aid its decision-making. Once a plan is computed, the Planning Daemon begins feeding commands into the Manipulation Daemon, which monitors and controls HUBO’s arms and end effectors, and the Balance Daemon, which performs balance and locomotion. These two daemons determine what position, velocity, and acceleration configurations are appropriate for the arms and legs respectively. The motion parameters are then filtered into the Control Daemon which calculates the motor commands necessary to generate the desired positions, velocities, and accelerations. These motor commands are sent into the CAN Daemon (called CAN because the hardware communicates using Controller Area Network). Finally, the CAN Daemon reads state information from the motor controllers and on-board sensors and publishes them to the State Ach Channel which is read by all the other daemons.

All of these processes run concurrently, based on their priority. They do not necessarily need to wait for data or signals from other processes in order to perform their calculations or send their signals. This is hugely beneficial because it means that the performance of the daemons does not need to be stalled by waiting for other components to respond. While the CAN Daemon is sending/receiving data from the hardware, the other daemons can be performing decision-making or dynamics calculations. With a system like this, it would be feasible for the Planning Daemon to begin planning for the next task while the Manipulation and Balance Daemons are carrying out the current plan. Finally, this multi-process division takes advantage of HUBO’s multi-core CPU by performing multiple computations simultaneously.

Along with the parallelization, the individual daemons do not require commands on any regular basis in order to perform their functions. The exact behavior of each daemon varies, but in general the daemons are designed to carry out to completion the last task given to them by their parent daemon. Specific behaviors are discussed in the next subsection. Decoupling the daemons from their parents and endowing each of them with a degree of autonomy ensures that overall performance is never held up by any single issue.

B. Specific Roles of Various Daemons

Presently, the bottom two tiers (Hardware Daemon and Control Daemon) are fully developed and stable while the third tier (Manipulation and Balance Daemons) are functional but undergoing continued development. The left-most tier (Planning

and Perception) will be developed within upcoming projects. It is also important to note that this whole design is extensible; it is not limited solely to the daemons as described in Fig. 2. More daemons or processes can be easily integrated into the structure. This potential integration will be described in greater detail in Section II-C.

1) *CAN (Hardware) Daemon*: The sensor and motor controller boards in the HUBO platform all communicate using a Controller Area Network (CAN) bus. The CAN Daemon is responsible for taking the command data structures assembled by other daemons and converting them into CAN frames to be sent to the motor and sensor boards. The CAN Daemon is also responsible for polling all the boards for state information, such as encoder positions, motor current (ampere) values, and sensor values (including force-torque and accelerometer data). After grabbing the state information from the CAN bus, the information is converted into data structures which are convenient for other processes to utilize, and these data structures are published to the State Ach channel. Along with the latest state data, the CAN Daemon provides the data with a timestamp. This timestamp allows the daemons to synchronize their behavior with the CAN daemon (and each other) if desired. It also provides a clear indication of the time lapse between instances of state data so that velocity (and any other time derivatives) can be calculated reliably. The daemon operates in real-time to allow Phase-Locked Loop communication over CAN. It sends commands at a fixed frequency to ensure that the CAN bus bandwidth is never saturated. This fixed frequency also allows other daemons or applications to operate at any frequency without affecting the rate of communication over CAN.

2) *Control Daemon*: Presently, the motor control boards on HUBO only offer position control. The position control gains used on the boards are extremely high due to Dr. Jun-ho Oh’s design philosophy for HUBO. Specifically, the philosophy is that if the hardware does precisely as it is instructed, then no further feedback control system is necessary in the software [5]. However, these extremely high gains can result in violent behavior if position commands are not sent to the boards very carefully. The Control Daemon ensures that the values sent to all of the motor controller boards are *always* sane. Motor control commands are not allowed to go to the boards without passing through the Control Daemon first. The Control Daemon has three particular modes:

Position Control – For each joint, the Control Daemon is

send a desired position (in jointspace), a nominal velocity, and a nominal acceleration. It is then the responsibility of the Control Daemon to smoothly move the joint from its current position and velocity to whatever desired position was requested. In particular, “smoothly” means that it must accomplish the task without ever exceeding the nominal velocity or the nominal acceleration. This generates a position trajectory like what is seen in Fig. 3 where each end of the trajectory is smoothed out parabolically. Note that there is an intentional deceleration prior to reaching the target position in order to prevent a sudden stop. The command containing the desired position only needs to be sent to the Control Daemon once and it will be carried all the way through to completion. But at the same time, it does not hurt in any way to repeatedly send the same control command to the Control Daemon. Rapidly sending alternating commands which strongly oppose each other (for example, fluctuating rapidly between $-2rad$ and $+2rad$) will simply cause the joint to move back and forth without ever violating the velocity and acceleration limits. In other words, it will not shake or do anything violent, no matter how extreme the position commands are.

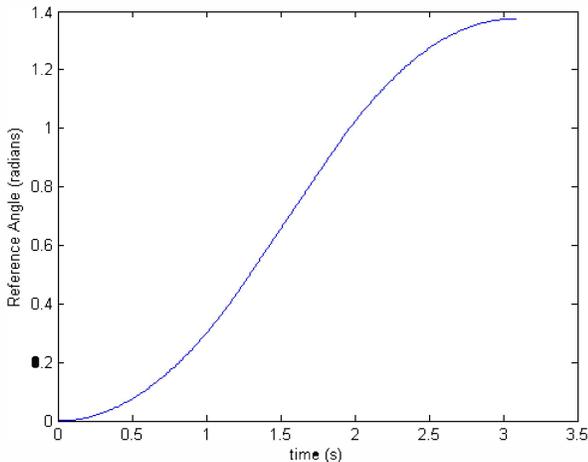


Fig. 3. Parabolic Smoothing of a Position Command

Velocity Control – For each joint, the Control Daemon is sent a desired velocity (in jointspace) and a nominal acceleration. The Control Daemon will then drive the joint at the desired velocity without ever exceeding the nominal acceleration. In general, this will generate a trapezoidal velocity trajectory. A key difference between velocity control mode and position control mode is that, unlike position control, the velocity control mode requires periodic updates on what the velocity should be or else the Control Daemon will decelerate the joints back down to 0. The time waited by the Control Daemon before decelerating is a parameter which can be set by the user or by whichever process is sending the velocity control command. This behavior is so that if a parent program using velocity control is interrupted or crashes, the Control Daemon will not blindly continue to push the joint forward.

Passthrough Control – Use of this control mode is generally discouraged. In this mode, the Control Daemon will simply pass any control commands straight through to the CAN Daemon without filtering or monitoring them. The use of this

mode is in order to accommodate outside processes which need to perform joint control without being subjected to any filter. If a particular control algorithm needs to send motor reference position commands without those commands being tampered with, they would use the passthrough mode.

3) *Manipulator Daemon*: This daemon takes end effector position and orientation commands and uses analytical inverse kinematics to generate the required joint angle and velocity configurations. These desired joint angle positions are then sent into the Control Daemon. In our current setup, the end effector commands are being streamed into the manipulator daemon from a Polhemus:FASTRAK sensor suite which follows the position and orientation of a human hand. This allows us to teleoperate the robot’s arms by simply holding a sensor in each hand and having the robot mimic our movements.

4) *Balance Daemon*: This daemon uses IMU and force-torque readings to maintain the robot’s balance at all times. As development continues, this will progress into a dynamic model-based controller. Moreover, it will ultimately be responsible for locomotion and controlling the gait of HUBO. It will receive commands like a state machine, which means there will be a finite enumerated list of commands (such as “step forward”, “step backward”, “turn by x degrees”) and the Balance Daemon will follow these commands in an intelligent way, ignoring commands which cannot be performed or delaying them until they *can* be performed.

5) *Other Daemons*: In general, any other daemons will be used for decision-making in some capacity, whether they are performing perception, planning, or optimization. These are high-level daemons which are not considered critical to the safety of the robot (although they may be critical to the successful execution of the robot’s task). They will communicate their plans to the Manipulation and Balance Daemon, and then those daemons will be responsible for converting the high-level demands into lower-level control commands. These daemons have no requirement to run in real-time. For example, a walking trajectory planner currently exists which uses ZMP Preview Control to generate a full body trajectory. It runs at a much lower frequency than the balance daemon and then periodically sends a chunk of trajectory to the real-time process which is responsible for running and maintaining the plan in real-time.

C. Application Programming Interface

A design such as this is not as useful if it is too burdensome to interface with. If a programmer needs to spend an inordinate amount of time dealing with “housekeeping” (such as organizing a data structure, consciously sending off packages, or parsing incoming messages) it distracts from the programmer’s real task of implementing a good algorithm.

The Ach suite comes with a library which makes these tasks straightforward. A simple function *ach_put* will send off a message to a channel, while *ach_get* will retrieve data from a channel. The messages are sent as raw byte arrays. Our software for HUBO uses C structures for messages, so no additional parsing or serialization/deserialization is necessary.

The software architecture which has been developed for HUBO takes this a step further. All of the “housekeeping”

is taken care of within a shared library which wraps all functionality up inside of a single C++ class. The constructor for this class automatically opens up all necessary communication channels, and member functions of the class handle all message and error handling. There are functions which have intuitive names (such as “setLeftArmAngles”) which take care of all data structure formatting and message sending. This approach enables the user to focus solely on algorithm design without needing to worry about the gritty details of implementation.

III. RESULTS

Here we will outline the results observed from using this design over repetitive trials. The goal of this section is to demonstrate – beyond mere postulation – the effectiveness of this design. First we will benchmark latencies in the system. Then we will list the observed behavior when individual components are forced to fail.

A. Benchmarking

In order to gauge the response frequency, commands from the balance daemon were tracked to see how long the following process took: 1) State data sent from Hardware Daemon, 2) Data processed by Balance Daemon, 3) Control Command sent by Balance Daemon, 4) Control command processed by Control Daemon, 5) Motor motor command sent to Hardware Daemon, 6) Motor command delivered to hardware, 7) New state data acquired by Hardware Daemon.

This time represents an upper bound on how much latency exists between the acquisition of data and the response of the system. A more exact measure of the latency could have been measured, but this would have required double the amount of data logging, which is a computationally expensive procedure. In fact, the four spikes seen in Fig. 4 are likely attributable to the logging.

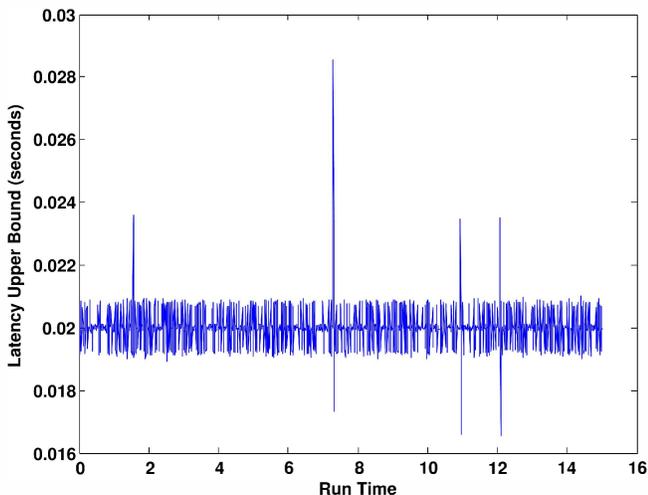


Fig. 4. Latency of commands travelling from Balance through Hardware Daemon

The upper bound on the latency hovers close to $0.02sec$. The reason for this is the Hardware Daemon operates at 100

Hz³, meaning it completes a single loop roughly every $0.01sec$. At the end of its loop, it sends off the latest state data. At the start of its loop, it grabs the latest commands which were sent out by the Control Daemon. These latest commands will be based on the state information (and timestamp) sent out by *the loop before the last one*, because the latest control commands were being computed *while* the Hardware Daemon was handling the CAN communication in the previous loop. What this amounts to is that *control computations are being performed while CAN communication is happening*, and those control computations will typically be available for the Hardware Daemon to send out by the time it begins its next loop.

B. Robustness to Software Errors

In the interest of having prior knowledge of what would happen if any component in the design were to fail, experiments were run in which each daemon was forced to quit via the operating system terminal during operation. This was performed multiple times while standing still and while in motion with consistent results, and those results are listed here. These describe the behavior which would be exhibited if each component were to fail unexpectedly.

1) *CAN (Hardware) Daemon*: The Hardware Daemon is certainly the most fundamental for maintaining control over the system. However, since the motor controller boards utilize position feedback, cutting off communication with them will simply result in the boards locking in place. If none of HUBO’s joints are moving when the Hardware Daemon is cut off, the boards will simply no longer respond to any further commands until the Hardware Daemon is restarted. If a joint *is* in motion when the Hardware Daemon is cut off, any moving joints will very suddenly stop.

2) *Control Daemon*: The physical result of the Control Daemon being cut off is identical to the result of the Hardware Daemon being cut off. The reason for this is that, either way, the boards will not receive any new reference position commands. Internally, however, all the other processes will continue to be updated about the states of HUBO, and therefore they can continue to monitor the state and condition of the hardware. The Control Daemon can simply be restarted to resume activity.

3) *Balance Daemon*: Balance is performed using velocity control on the leg joints. Currently they have a time-out delay of 0.5 seconds. This means that if a new velocity control command is not sent within a half-second, the Control Daemon will smoothly wind the velocity of all the joints down to zero, ultimately freezing them in place until a new velocity command is received. Therefore, if the Balance Daemon is killed prematurely, the Control Daemon will continue driving the joints at the last velocity command it received until 0.5 seconds have elapsed; then it will wind down all the leg velocities to zero. If the robot is in a stable position as the leg velocities get wound down, then the robot will remain upright.

4) *Manipulator Daemon*: Manipulation is performed using position trajectories. In particular, the Manipulator Daemon sends “desired location” commands to the Control Daemon

³100 Hz is dictated by the need to not saturate the CAN bus. However, 100 Hz is not an upper-limit, and this might be increased in future operations on HUBO.

Failed Component	Stability of Motion		
	Static	Quasi-Static	Dynamic
Hardware	No Effect	Sudden Stop in All Joints	Falls over
Control	No Effect	Sudden Stop in All Joints	Falls over
Balance	No Effect	Smooth Stop in Legs	Falls over
Manipulator	No Effect	Smooth Stop in Arms	Smooth Stop in Arms
Planner	No Effect	Finishes last plan provided	Finishes last plan provided
Perception	No Effect	Finishes last plan provided	Finishes last plan provided

TABLE I. FAILURE MODES OF DIFFERENT COMPONENTS

which generates a smooth trajectory from the current joint locations to the desired location. Therefore, when the Manipulator Daemon is forced to crash, the Control Daemon will ensure that the arms simply arrive at the last position requested by the Manipulator Daemon. Moreover, the Control Daemon is aware of all joint limits, so it will never try to send a joint past its limit.

5) *Planning & Higher-level Daemons*: Presently only a walking trajectory planner exists, but its failure mode would be reflective of all other high level daemons (such as a perception daemon or a manipulator planner daemon). The trajectory planner daemon is designed to always send segments of a trajectory plan which ends in a stable configuration. Therefore, if it is cut off, the Balance Daemon will continue its operation up to this stable configuration and then wait until a new command is provided. In general, the balance and manipulator daemons have default behaviors to fall back on in the event that their pipeline to the higher level daemons is interrupted.

A chart of these results is presented in Table 1. In all of these circumstances, each daemon can be restarted and resume its task seamlessly. There are only three daemons for which crashing would pose a potential hazard to the robot: the Balance, Control, and Hardware Daemon. In theory, if one of these crashed while the robot was in a dynamically unstable configuration, the robot would fall over. Because of this, a great deal of emphasis is placed on the consistency, stability, and reliability of these three daemons. An advantage of having the distributed architecture is that development efforts can be fine-tuned to the specific components which are most critical.

IV. CONCLUSION

We have presented the outline of a general software architecture for real-time robot planning and control which was designed with emphasis on safety and modularity. This architecture is meant to be easily extendible to many platforms, though it is particularly well suited for humanoid robots or other platforms where errors or unexpected behaviors are dangerous. A particular implementation of this architecture for a HUBO2 Plus robot was discussed and analyzed to demonstrate the practical value of the architecture.

This multi-process architecture offers several key advantages over a single-process approach: 1) Future development does not need to be crammed into prior development. Instead, a new process can be developed which simply communicates with previously developed processes. So a planning algorithm does not need to be fit inside of a control system or vice-versa. They can exist and be developed independently. 2) Different functionalities do not have critical dependencies on each other. For example, if a planner experiences a segmentation fault or crashes for any other reason, the balancing and control

processes can remain in-tact and prevent the robot from falling over or being damaged. 3) Latency in any single component does not affect the performance of any other component of the software. In general, the individual processes do not have to wait for any higher level process in order to perform their tasks, except when waiting for instructions to begin performing a new task. 4) The operating system is able to take full advantage of modern multi-core processor hardware. Multi-threaded processes have the benefit of utilizing multi-core processing, however they still suffer from the first two disadvantages mentioned above: potentially bloated codebases and vulnerability to instant irreversible critical failure.

ACKNOWLEDGMENT

The authors thank Peter Vieira and Rowland O'Flaherty for their extensive efforts in operating HUBO and running trials. This work was supported in part by DARPA #N65236-12-1-1005: DARPA Robotics Challenge and NSF CNS-0960061 MRI-R2: Unifying Humanoids Research.

REFERENCES

- [1] N. Dantam and M. Stilman. *Robust and efficient Communication for real-time multi-process robot software*. International Conference on Humanoid Robotics (Humanoids). 2012.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] Baek-Kyu Cho and Sang-Sin Park and Jun-ho Oh. *Controllers for running in the humanoid robot, HUBO*. Humanoid Robots, 2009. Humanoids 2009. 9th IEEE-RAS International Conference on, Dec. 2009.
- [6] Lofaro, Daniel M. and Ellenberg, Robert and Oh, Paul and Oh, Jun-ho *Humanoid throwing: Design of collision-free trajectories with sparse reachable maps* Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on Oct. 2012.