2016 IEEE International Symposium on Robotics and Intelligent Sensors (IRIS2016)
17-20 December 2016, Tokyo Japan

Reliable Software Architecture Design with EtherCAT for a Rescue Robot

Youngwoo Lee, Wonsuk Lee, Byunghun Choi, Gyuhyun Park, and Yongwoon Park
Agency for Defense Development
Daejeon, South Korea
Email: {ywlee, wsblues, bhunchoi, khpark, yongpark}@add.re.kr

***Abstract*—Modern robot researches focus on disaster response to carry out several missions by a robot itself. Robot software should handle a large number of calculation under real-time constraints. And also, the software design should be complex to process a lot of sensors, which increases a number of factors to consider for the system performance.**

In this paper, we present the software architecture design for the rescue robot which rescues a wounded person and moves dangerous objects in disaster situations. The software architecture is designed with real-time APIs from a Xenomai real-time kernel patch. The architecture uses EtherCAT which is an Ethernet-based real-time network to communicate with all joint controllers. The processors in the architecture communicate each other by the shared memory. We study two considerations to improve the performance and reliability. First, we present a method to minimize the memory synchronization procedure which is a drawback of the shared memory communication. We also consider processor affinity for all real-time tasks in the architecture to stabilize execution time of high priority tasks.

I. Introduction

Modern robot researches focus on disaster response that contains several missions to be managed by robot itself such as rescuing a person or closing a valve. In the Defense Advanced Research Projects Agency (DARPA) Robotics Challenge (DRC), many intelligent robots had shown their possibilities as the replacements of human at disaster response. According to the demand of the disaster response robot [1], [2], we have been developing a rescue robot to carry a person and to move several harmful objects from the dangerous place. The rescue robot is a half-humanoid as shown in the Figure 1. The upper body consists of a waist and a dual-manipulator with two types of hands (rescuing and grasping hands) for different purposes. The lower body is a variable configuration platform made of two legs covered with caterpillar tracks for the mobility on various terrains.

The disaster response robot requires well-designed software architecture. To solve particular missions alone, the robot would contain a number of sensors to take the place of human sense. The robot handles the sensor data to make motion behaviors which are a sequence of joints angle set. Every sequence requires a number of complex matrix calculations that should be calculated in certain time to operate robot in the real-time. It means that the software requires many computational power and real-time operation.

Many researchers have made efforts to design robot software architecture. There are a number of robot software because the robots can have any combination of various hardware. Robot Operating System (ROS) [3] is widely used as a communications layer among control application layers as well as a hardware interface layer[4], [5], [6]. Some robots have their own software frameworks with typical real-time operating systems such as Linux real-time kernel patches (Xenomai [7], RTAI [8]) and VxWorks [9]. For instance, PODO[10], software of Hubo, uses shared memory to communicate between software control tasks and a hardware interface. In this manner, typical robot operating software separates control application layers and a hardware interface layer with its own communication layer. This modularization may have a number of benefits in minimization of single error effect, easy developing procedure, and so on.

The one of biggest issues of designing software of such disaster response robots is reliability. Operating software of the robot should answer from the user input and external sensor data sensitively. Although multi-process software design is necessary to increase the reliability, it brings complexity in the calculations and may disturb real-time operation. Several internal factors of the multi-process desgin should be considered for the system performance. In other hands, the robot may be damaged when contacting a human or dangerous objects, especially in fields. In case of damage in several joints by external shock, the robot may calculate incorrect motion behaviors while solving kinematics if it did not recognize it.

In this paper, we present a reliable software architecture for the rescue robot considering several factors. We first introduce the rescue robot briefly and show the software architecture. To provide the reliability, we contemplate improvements of our design with shared memory and processor affinity. We present a method to minimize memory synchronization. We define memory space that is allocated for several processes. The processes can use own memory space freely and require memory synchronization when only it is needed. We suggest a simple setting method about processor affinity which increases reliability of processes with high priority. We also suggest a reconfiguration method from external shock in the software architecture which uses the EtherCAT network. The EtherCAT network is an Ethernet based real-time protocol and supports more than 1 KHz of guaranteed network update rates. However, the EtherCAT has a drawback for field robots because of ring network topology; If a joint is broken, the entire network will be stopped. We suggest a method to tackle the

978-1-5090-6084-9/16/$31.00 ©2016 IEEE

Fig. 1. The Rescue Robot

problem, which checks malfunction of joints and notifies it to algorithms. The method also contains to backup information of the joints with a file system for system reset.

II. BACKGROUND

A. Software Architecture in Humanoid Robots

The robot is a real-time system. Unexpected delay may cause unbalanced motion behaviors, which are unwelcome problems to algorithm developers. Moreover, the robot control trends are moving to the dynamic control from the behavior based control recently. Dynamic robot control needs a large number of calculations to make motion behaviors in every period with balancing, avoiding collisions, reacting from external inputs, and etc. The robot software architecture design has become more complex to handle lots of calculation and real-time processing.

To provide modularity and reliability in the complex robot software architecture, a multi-process design is necessary [11]. Each process has own identity and cooperates with each other to operate complex algorithms. Single fault of process has less effect to the software compared to the single-process design. Despite these benefits, the multi-process design brings several considerations in inter-process communication (IPC) and process affinity. Processes need IPCs to communicate each other. IPC is provided by the operating system and also distributed by other researches[12], [13]. Process affinity binds or unbinds the processes to CPU thread set to execute only on the designated to CPU thread set. Typically, a real-time task is allowed to execute on only a CPU thread which is pre-defined before the execution. To select the IPC and to set process affinity, entire robot environment should be considered because these two factors are closely related with system performance and implementation methods. We discuss two considerations in detail at Section IV-A and Section IV-B.

B. A Real-time network system : EtherCAT

The network system is one of important factors in the software architecture. The control messages must be delivered to robot joint controllers in a certain time. Since typical network systems are slower than other processing devices in general, The system control loop frequency is dependent on the network systems. Most popular real-time networks are mainly designed in order to meet time constraints, and they are not well-suited for delivering large data. For instance, CAN is the most Popular real-time communication network and widely used in robot systems. CAN, however, supports only 1 Mbps of bandwidth and it is not suitable for the systems which needs large bandwidth.

EtherCAT is an Ethernet-based protocol and a master-slave communication protocol. An EtherCAT master and one or more EtherCAT slaves in a network are connected into any network topologies with the EtherCAT switch such as line, ring, tree, and so on. The EtherCAT protocol needs full duplex Ethernet physical layers to propagate data from and to the master, therefore, actual protocol logic works like a ring topology. The master creates more than one EtherCAT frame, which contains particular memory space of all slave process images, and propagates the frame to all the slaves from the first slave to the end. The last slave, which is not connected with any slaves except the previous slave, receives the frames and returns them. At the first propagation, the slaves read and write the frames on-the-fly. After the last slave returns the frames, every slave only propagates them to the master. The particular memory space is pre-defined in network scanning time. The network scanning collects the process images of all slaves which are defined with slave configurations.

The biggest merit of using EtherCAT is a clock synchronization mechanism known as Distributed Clock (DC). DC enables the network to be synchronized within several tens of nanoseconds. It is possible because of the ring topology. The master writes reference time on the EtherCAT frame and sends it to all slaves. And every slave writes own local time on the EtherCAT frame twice, once at receiving and once at returning in the ring topology. The master calculates the propagation delays between all slaves with all slave local time. Every slave can find difference between each local time and the reference time and the master sends all propagation delays calculated in the previous step to all slaves.

Unfortunately, however, the ring topology and fixed address of EtherCAT frame cause a drawback for the robot. If a slave is damaged by unexpected external sources, the EtherCAT frame cannot reach to other slaves after the slave. Moreover, the fixed address of EtherCAT frame does not match with the network topology anymore. EtherCAT has a functionality called cable redundancy to compensate for the problem. However, cable redundancy does not work with DC synchronization.

III. THE RESCUE ROBOT AND THE SOFTWARE ARCHITECTURE

In this section, we introduce the rescue robot briefly and present the software architecture of the robot. We focus on

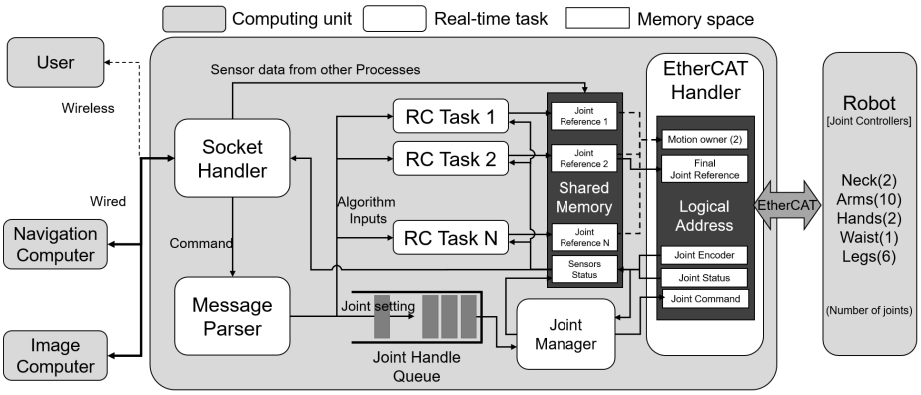

Fig. 2. The overall software architecture of the rescue robot

only designing reliable software in this paper. Main concepts of the robot and control methods are introduced in other researches [14], [15].

A. The Rescue Robot

Figure 1 shows the rescue robot. The main purposes of the robot are to rescue a person and to move dangerous objects from disaster area or battlefield. The robot is controlled by remotely. The robot has 21 joints and many vision sensors. Because all joints consist of worm gears, even though the power of robot is turned off, the robot maintains own posture. For an elaborate control, we can control all joints or several parts of the robot such as a upper body or driving joints only or end-effectors through kinematics calculation by a joystick or a haptic device.

Our robot contains 3 identical computers containing an Intel Core i7-3517UE 1.70GHz CPU, 2 CPU cores and 4 CPU threads, and 8 GB of RAM. The first one handles GPS data and IMU data for navigation related applications. The second one processes image data with all vision sensors. The last computer, which is focused on by this paper, controls robot motions with various motion control methods and stabilization applications with Ubuntu Linux 12.04 (Linux kernel 3.4.6) and the Xenomai 2.6.3 real-time kernel patch. In this paper, we present the software architecture of the last computer to provide reliable computing environments to the applications.

The robot uses the EtherCAT to communicate with joint controllers. The EtherCAT network is an Ethernet based real-time protocol and widely used in industrial robots. EtherCAT supports more than 1 KHz of guaranteed network update rates, and provides a distributed clocks (DC) synchronization functionality which enables whole joints move simultaneously with under nano-seconds of error. Because the robot should handle and contact with the wounded person, DC synchronization is important to gaurantee operating the robot simultaneously. However, the EtherCAT has the drawback for field robots because of ring network topology. If a joint is demaged, the system would be stopped. The cable redundancy can not be used with DC synchonization. We suggest a method to solve the problem in Section IV-C.

B. Software Architecture

Figure 2 shows the overall software architecture of the motion control computer. All tasks are implemented by Xenomai real-time task APIs. The software contains several real-time tasks with own different purposes. Real-time tasks communicate each other through the shared memory. Reasons of that we choose the shared memory as the inter-process communication in our robot are implementation convenience and communication speed. The shared memory has an advantage in data exchange speed[12]. The Real-time Control (RC) task developers can easily handle several tens of sensor data and joint data with the shared memory.

Socket handler processes all socket messages from a user or other computers. Socket handler separates the data into sensor data from the other computers and control data from the user. When Socket handler receives the sensor data, e.g. IMU data and GPS data, it stores the data in the shared memory. The control data from the user is passed to Message parser.

Message parser has two different types of control data. The first data is inputs of the RC Tasks. The rescue robot contains many different RC Tasks such as a whole-body control task or a stabilization control task. Message parser delivers the control data to certain RC task. The second data is a joint control message. It is a kind of joint setting message such as selecting an operation mode in a position control mode or a velocity control mode or a torque control mode. Message parser provides a particular functionality to give convenience to the user, that divides the joint control message. For example when the user starts to operate the robot, the user should initialize the robot. It has several steps including control mode selection, setting position or torque reference, and enable. Message parser divides several pre-defined user messages, like above mentioned, into several joint setting messages. The divided control messages are queued in Joint handle queue.

RC tasks receives the inputs from Message parser. The inputs contain target positions to reach with end-effectors of the robot. In every period, the RC task reads sensor and joint data then it calculates next joint references with own algorithms. At last, the RC task writes the references in the shared memory. The algorithms of RC tasks contain inverse

TABLE I
CHARACTERISTICS OF TYPICAL REAL-TIME TASKS AND ASSIGNED PRIORITY AND PROCESS AFFINITY

	Priority	Shared memory	Characteristic	CPU thread affinity
Basic software	lowest	×	Running by the operating system	0
Socket Handler	low	△	Handling bulk data from external sensor data	1
Message parser	low	×	Sending commands and inputs to Joint manager and RC tasks	1
Joint manager	high	◯	Handling joint status	2
RC Tasks	high	◯	Running a number of matrix calculation but not always run	2
EtherCAT handler	highest	◯	Communicating with all joint controllers, the highest priority	3

kinematics and forward kinematics which have a large number of matrix operations. Besides, because the robot has enough redundancy for making motion behaviors, RC task can execute several sub-tasks, such as center of mass (COM) balancing or detecting and avoiding self-collision concurrently. So it could make the computation burden increased. To reduce the calculation time, we use gcc compiler -O3 optimization option.

Joint manager monitors and manages the robot statuses as a deterministic finite state automaton (DFA) with the shared memory. Joint manager separates the robot statuses in 5 states like shown in Figure 3. Each state has proper inputs to move to another mode. The DFA prevents irregular state transition which may occur dangerous robot operations. Joint manager reads the joint control message to change the state when Joint handle queue is not empty. Joint manager also monitors the robot information from the shared memory. If it detects the improper motions such as out of limit position references or exceeded joint velocity, it would lead to Error state.

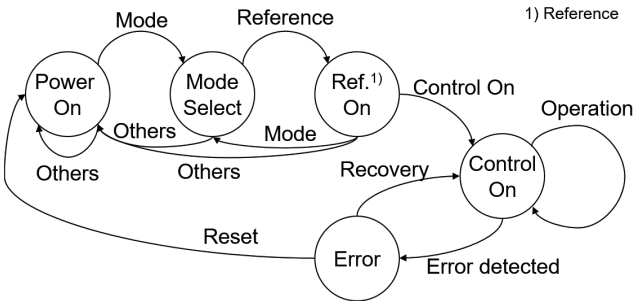

Fig. 3. Joint status management with DFA

EtherCAT handler is an end point of the software architecture. It exchanges the EtherCAT frame with joint controllers and sensors in every 1 ms. It updates joint encoders and statuses in the shared memory and sends joint references to the joint controllers periodically. Aperiodic messages such as joint control messages are also delivered to joint controllers. EtherCAT handler also checks limits of a set of joint position, velocity, and torque value. If any one of these overstep the limit, EtherCAT handler would stop to operate the robot.

IV. PROVIDING RELIABILITY IN THE SOFTWARE ARCHITECTURE

A. Minimizing memory synchronization

The rescue robot uses the shared memory for an inter-process communication because it has an advantage in data exchange speed. Contrary to the benefit of the shared memory, memory synchronization is a critical issue. Without the synchronization, several tasks may access the shared memory concurrently and it may cause system faults. The synchronization decreases system performance, since a task which wants to access the shared memory may be stopped by another task which already occupies the shared memory.

To minimize the memory synchronization procedure, we adopt the concept of the EtherCAT logical memory space. EtherCAT frame defines logical memory space which contains data variables of all slave devices before the operation. Then, EtherCAT protocol communicates with slave devices by exchanging pre-defined memory space and the memory of slave devices. We define memory space of each RC tasks in the shared memory. Each RC tasks freely access own memory space. Only the Motion owner space of the shared memory requires the synchronization to access.

In the figure 2, we describe an example. RC task 2 receives the algorithm inputs from the user. RC tasks 2 tries to lock the Motion owner to make own joint control references. After locking the Motion owner, RC task 2 writes the references in own memory space in the shared memory, Joint Reference 2. EtherCAT handler checks which task has the ownership with Motion owner, and waits when RC task 2 finishes the calculation. EtherCAT handler receives the event from the RC task 2 and copies the Joint Reference 2 to final joint references. Even though other RC tasks receive the algorithm inputs from the user faults, the tasks cannot lock the Motion owner because it is already locked by the task 2. The tasks calculate own motion control references in own memory space, but it does not have any effect to the robot. We implement this concept that minimizes the memory synchronization process and disturbing the system performance.

B. Setting process affinity

We first check whether the architecture without any process affinity considerations can guarantee the real-time operation in the robot. We measure the execution time of two major real-time tasks using the RDTSC instruction [16]. The first task is a RC task which makes the rescue motion behaviors containing a main task of two end-effector poses planning (lowering, forwarding, backwarding), two sub tasks of lower-arm level motion control and center of mass balancing. The second task is EtherCAT handler which communicates with robot joints and it should meet the real-time constraints. In every sequence of the system period, the RC task makes next

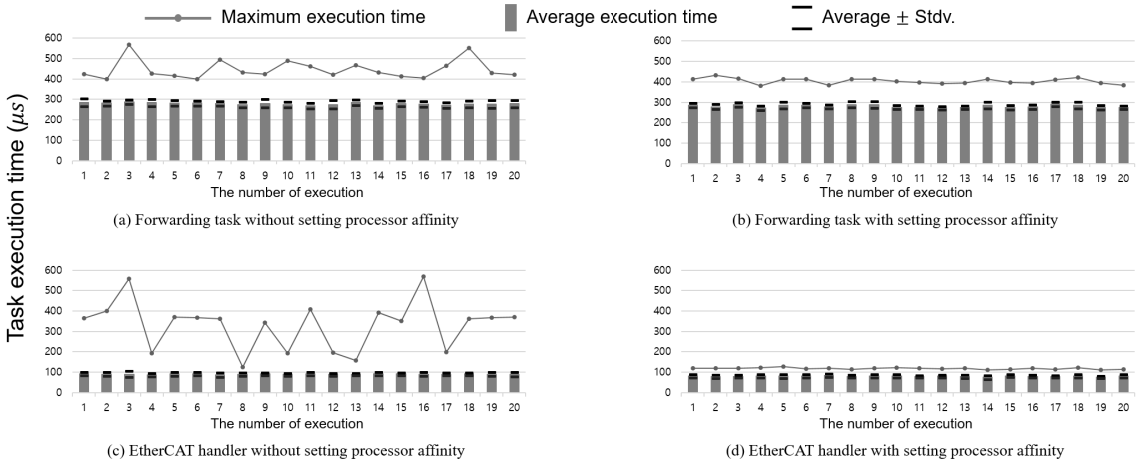

Fig. 4. Execution time of real-time tasks in every period (1 ms)

motion behaviors and writes them in the shared memory, then EtherCAT handler reads the joints references of the next step and sends them to the joint controllers with the EtherCAT protocol. The two tasks consume almost time of operating the robot in every sequence. In this experiment, we want to gauge the performance of entire system in realistic conditions.

Figure 4 (a) and (c) show that the execution time of the two real-time tasks without process affinity considerations. Results show that it seems that the two tasks are stable. The average values and standard deviations are similar in every execution. However, the maximum values can not plot any trend line in both case. In the architecture, socket handler receives many sockets from the other computers and the user by wired or wireless communication devices. Moreover, every operating system has own basic software which are necessary to manage the computer. Without any process affinity considerations, all tasks can be related and affect each other even if some tasks have higher priority than others. In the worst case, the summation of the maximum execution time values of two tasks does not meet the system period, 1ms.

To provide robustness in the software architecture we consider processor affinity. First of all, we isolate CPU threads real-time tasks from basic software. Although all real-time tasks have higher priority than basic software, they may have disadvantage in the context switching when they use same CPU thread. We allocate all basic software in CPU thread 1 which is not used by any real-time tasks to minimize effect of basic software. The real-time tasks are allocated in 2, 3, 4 threads. In addition, we characterize all tasks in the software architecture as shown in Table I. EtherCAT handler communicates with the joint controllers and sensors which reflect behaviors of the robots and information whether contacting with a human or not. So, we provide the highest priority to EtherCAT handler. We provide high priority to Joint manager and RC Tasks. Joint manager handles robot statuses and the reconfiguration process from the external damage. RC Tasks calculate main algorithms for the robot missions. The other tasks have low priority. We also confirm whether the real-time tasks use the shared memory or not. It is an important characteristic because it is highly related with a cache hit ratio. When the tasks located in a core use same memory space, the CPU core stores the space in the cache and it leads high access performance. Because the cache is located in every CPU cores and not in every CPU threads, we tried to locate all tasks which use the shared memory in the same CPU core. The result of set process affinity is shown in Table I

Figure 4 (b) and (d) show the impact of setting process affinity. The result shows that the maximum values of the execution time are also stable. It is reasonable because all unexpected time consuming factors are separated in the first CPU core. Surely, socket handler and message parser may be more influenced in comparison with the first case because they are located in the same core with basic software. However, RC tasks and EtherCAT handler are more related with system reliability in the robot and their results show that they are separated from unexpected time consuming factors dramatically. These results validate that setting process affinity method is effective in satisfying the real-time requirement.

C. Discussion : A reconfiguration process

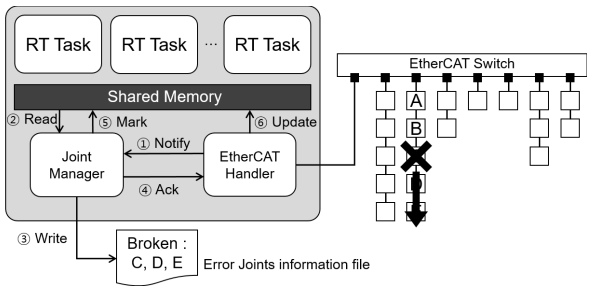

Fig. 5. Joint Manager reconfiguration procedure

Field robots such as the rescue robot are particularly susceptible in various external environment. Varying terrains and

contacts with a person or objects make the robot difficult to address the missions. These difficulties could cause severe situation that the rescue robot may break down in several joints rather than indoor robots. In this case, when RC tasks make motion behaviors, damaged joints are not reflected and RC tasks may produce incorrect motion behaviors. To minimize the problem, each part of the rescue robot is connected to different port of the EtherCAT switch. Even if an arm of the robot is damaged, the other parts, such as necks, another arm, hands, and legs, can operate. In this section, we suggest a reconfiguration process which contains detecting and managing the damaged joints. In this case, the reconfiguration process is handled by Joint manager. Figure 5 shows a procedure of the reconfiguration process. The detail procedure is explained with an example as follows:

1) EtherCAT notifies to Joint manager when it has problem to communicate C, D, E joints.
2) Joint manager reads current information of C, D, E joints from the shared memory.
3) Joint manager writes an error joints information file about C, D, E joints with a file system.
4) Joint manager answers to EtherCAT handler.
5) Joint manager marks that C, D, E joints have problem and changes the joint statuses in the shared memory.
6) EtherCAT handler resumes to update information of the other joints.

EtherCAT handler can detect some joints have problem because it communicates to all joints with an EtherCAT frame. Because EtherCAT uses the ring topology, even if only C joint has a problem, EtherCAT handler cannot communicate to all of C, D, E joints. To manage the malfunctioning joints, Joint manager writes information of the joints with a file system. Error joints information file should be saved in non-volatile memory. The reason of the backup this information is that the robot should know last states of the joints because the joints and worm gears keep own positions. If the robot is reset or power is turned off, the shared memory does not have any information of C, D, E joints. Then, Joint manager can read the information of the joints form Error joints information file. After Joint manager marks that the joints statuses in the shared memory, RC tasks calculate motion behaviors without the malfunctioning joints.

This procedure is based on an EtherCAT network, however, it can be applied to other networks easily. The other networks can detect the issued joints with communication errors.

V. CONCLUSIONS

In this paper, we have presented the reliable software architecture for the rescue robot. We operate the rescue robot with several real-time tasks with the shared memory as the interprocess communication. We have proposed two considerations which help to provide reliability into the software architecture. To use the shared memory intelligently, we have presented the pre-defined memory space method to minimize the memory synchronization for real-time tasks. The second idea is to separate real-time tasks into different CPU threads to stabilize execution time of high priority tasks. Moreover, we have shown the reconfiguration procedure when some joints have problem from external collisions. We have a plan to develop the reconfiguration procedure in variety situations. We believe the proposed software architecture could support to operate field robots with reliability.

ACKNOWLEDGMENT

This research was supported by a grant for the Project managed by the Agency for Defense Development. "Technology development for a rescue robot capable of lifting over 120 kgf", funded by Civil-Military Technology Cooperation Program.

REFERENCES

[1] "Bear Robot." [Online]. Available: http://www.vecna.com/research
[2] J. Ding, Y.-J. Lim, M. Solano, K. Shadle, G. Park, C. Lin, and J. Hu, "Giving patients a lift-the robotic nursing assistant (rona)," in *2014 IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*. IEEE, 2014, pp. 1–5.
[3] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.
[4] M. DeDonato, V. Dimitrov, R. Du, R. Giovacchini, K. Knoedler, X. Long, F. Polido, M. A. Gennert, T. Padır, S. Feng, *et al.*, "Human-in-the-loop control of a humanoid robot for disaster response: A report from the darpa robotics challenge trials," *Journal of Field Robotics*, vol. 32, no. 2, pp. 275–292, 2015.
[5] N. A. Radford, P. Strawser, K. Hambuchen, J. S. Mehling, W. K. Verdeyen, A. S. Donnan, J. Holley, J. Sanchez, V. Nguyen, L. Bridgwater, *et al.*, "Valkyrie: Nasa's first bipedal humanoid robot," *Journal of Field Robotics*, vol. 32, no. 3, pp. 397–419, 2015.
[6] M. Zucker, S. Joo, M. X. Grey, C. Rasmussen, E. Huang, M. Stilman, and A. Bobick, "A general-purpose system for teleoperation of the drc-hubo humanoid robot," *Journal of Field Robotics*, vol. 32, no. 3, pp. 336–351, 2015.
[7] "Xenomai Home page." [Online]. Available: http://www.Xenomai.org
[8] "RTAI Home page." [Online]. Available: http://www.rtai.org
[9] B. G. Woolley, G. L. Peterson, and J. T. Kresge, "Real-time behavior-based robot control," *Autonomous Robots*, vol. 30, no. 3, pp. 233–242, 2011.
[10] J. Lim, I. Shim, O. Sim, H. Joe, I. Kim, J. Lee, and J.-H. Oh, "Robotic software system for the disaster circumstances: System of team kaist in the darpa robotics challenge finals," in *Humanoid Robots (Humanoids), 2015 IEEE-RAS 15th International Conference on*. IEEE, 2015, pp. 1161–1166.
[11] M. Grey, N. Dantam, D. M. Lofaro, A. Bobick, M. Egerstedt, P. Oh, and M. Stilman, "Multi-process control software for hubo2 plus robot," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1–6.
[12] N. T. Dantam, D. M. Lofaro, A. Hereid, P. Y. Oh, A. D. Ames, and M. Stilman, "The ach library: A new framework for real-time communication," *Robotics & Automation Magazine, IEEE*, vol. 22, no. 1, pp. 76–85, 2015.
[13] A. S. Huang, E. Olson, and D. C. Moore, "Lcm: Lightweight communications and marshalling," in *Intelligent robots and systems (IROS), 2010 IEEE/RSJ international conference on*. IEEE, 2010, pp. 4057–4062.
[14] W. Lee, Y. Lee, G. Park, S. Hong, and Y. Kang, "A whole-body rescue motion control with task-priority strategy for a rescue robot," *Autonomous Robots*, pp. 1–16, 2016, doi:10.1007/s10514-016-9562-4.
[15] S. Hong, Y. Lee, K. H. Park, W. S. Lee, B. Choi, O. Sim, I. Kim, J.-H. Oh, and Y. S. Kang, "Dynamics based motion optimization and operational space control with an experimental rescue robot, hubo t-100," in *Advanced Intelligent Mechatronics (AIM), 2015 IEEE International Conference on*. IEEE, 2015, pp. 773–778.
[16] I. Coorporation, "Using the rdtsc instruction for performance monitoring," *Techn. Ber., tech. rep., Intel Coorporation*, p. 22, 1997.