Software Architecture of the da Vinci Research Kit

Zihan Chen, Anton Deguet, Russell H. Taylor and Peter Kazanzides
Laboratory for Computational Sensing and Robotics
Johns Hopkins University
Baltimore, Maryland 21218
Email: {zihan.chen,anton.deguet,rht,pkaz}@jhu.edu

***Abstract*—The da Vinci Research Kit (dVRK) has been installed at over 25 research institutions across the world, forming a research community sharing a common open-source research platform. This paper presents the dVRK software architecture, which consists of a distributed hardware interface layer, a real-time component-based software framework, and integration with the Robot Operating System (ROS). The architecture is scalable to support multiple active manipulators, reconfigurable to enable researchers to partition a full system into multiple independent subsystems, and extensible at all levels of control.**

I. Introduction

Telerobotic systems have a proven track record in several application domains, including minimally-invasive surgery, space exploration, and handling of hazardous materials. However, most real-world systems still use direct teleoperation, where a human controls each action of the remote robot, even though research in semi-autonomous teleoperation, including supervisory control, shared control, and other co-robotic methods, has been active for decades. One obstacle had been the lack of a robust common research platform, but this has recently been addressed by the availability of systems such as the da Vinci Research Kit (dVRK)[1] and Raven II robot [2].

This paper focuses on the software architecture of the dVRK, which is currently in use at more than 25 research centers around the world. The choice of architecture was influenced by the following key requirements:

1) Scalability to multiple master and slave robot arms. A full da Vinci System typically contains six active robot arms and four passive robot arms.
2) Easy reconfiguration, such as adding or removing arms or even splitting the system into multiple independent setups.
3) Use of a familiar software development environment, such as C++ on a Linux PC, for all levels of the software architecture.
4) Real-time performance for high-frequency, low-level robot control.
5) Ability to integrate with other high-level robot components and development environments, such as Matlab and Python, via middleware.

These requirements led to the adoption of a centralized processing and distributed I/O architecture [3] that enables all processing to be performed on a personal computer (PC). The dVRK uses C++ on Linux, though most of the software is portable to other platforms. The key layers of the software architecture, shown in Fig. 1, derive from the following design decisions, which are presented in subsequent sections:

1) Use of a high-bandwidth field bus that supports daisy-chain connection, multicast communication, and an efficient (low overhead) software interface, which satisfies the requirements for scalability and reconfigurability. This is discussed in Section IV, which presents IEEE 1394a (FireWire) as the primary field bus for the dVRK.
2) A real-time, component-based framework that enables high bandwidth, low latency control. Section V describes the design of the real-time software layer for the dVRK, which is based on the open source *cisst* libraries developed at Johns Hopkins University (JHU) [4], [5].
3) Bridge or proxy components that provide interfaces between the real-time component-based framework and other systems. Initially, this was provided by a custom middleware [6] based on *cisst* and Internet Communications Engine (ICE), but has since transitioned to Robot Operating System (ROS) [7], as discussed in Section VI.

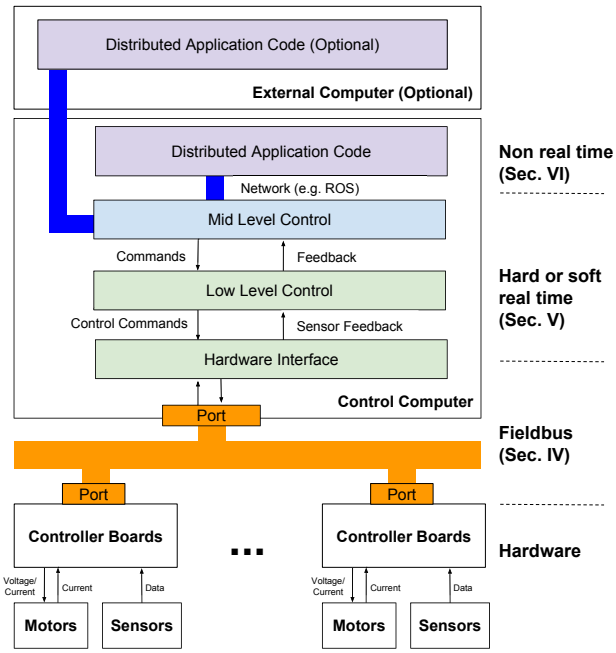

Fig. 1: da Vinci Research Kit (dVRK) control architecture

978-1-5090-6724-4/17 $31.00 © 2017 IEEE
DOI 10.1109/IRC.2017.69

II. Related Work

There has been an increasing need for open robot platforms for research. We consider a platform to be "open" if it gives researchers direct access to all sensors and actuators and allows them to freely write/modify all levels of the control software. This section reviews the control architectures of three widely available open robot platforms.

The Whole Arm Manipulator (WAM, Barrett Technology, Inc., Cambridge, MA) [8] is a 7 degree-of-freedom (DOF) cable-driven robot with an optional three-finger Barrett hand. It supports torque control of the robot and thus is an ideal platform for implementation of advanced control algorithms. The robot arm has a distributed motor controller module, called *Puck*, installed on each joint and interconnects them through a CAN bus at 1 Mbps. Robot control can either be done with the internal Linux control computer with Xenomai patched real-time kernel or with an external computer through the exposed CAN bus port. The manufacturer also released an open-source C++ library, *libbarrett*, which contains CAN bus communication and kinematics routines. Recently, [9][10] implemented control architectures that use the Robot Operating System (ROS) for high level interface and the Open Robot Control Software (OROCOS) for low-level control.

Another important open robot platform is the Personal Robot 2 (PR2, from Willow Garage, Palo Alto, California). The robot features an omni-direction wheeled base, two torque controlled 7-DOF arms with 1 DOF gripper, an actuated head and other sensors (e.g. laser sensor, stereo camera). PR2 motion control comprises Motor Controller Boards (MCB) interfacing motors and encoders, EtherCAT field bus, hard real-time control software and a non-real-time ROS-compatible software stack. The MCB closes a current PI-control loop at 100 kHz on a FPGA-based design. The main motor control PC runs a PREEMPT_RT patched Linux kernel for real-time performance[11]. A real-time process handles EtherCAT communication, servo-level control and publishes robot states via a real-time safe ROS publisher. To add flexibility and extensibility, a controller manager is implemented to dynamically load real-time compatible controller plugins. Overall, the design provides a real-time safe solution compatible with ROS, as well as extra flexibility through the use of plugins. However, the real-time code is robot specific and cannot easily be reused.

In the medical robotics field, the Raven II Surgical Robotics Research platform [2] is an open architecture, patient-side robot for laparoscopic surgery that consists of two cable-driven 7 DOF arms. It was a collaborative effort between the University of Washington (UW) Biorobotics Lab and the University of California Santa Cruz (USCS) Bionics lab, and was based on Raven I developed at UW [12]. The UW/USCS team built several Raven II systems that were installed in other research labs and subsequently spun out production to a startup company, Applied Dexterity Inc, that has continued to deliver systems. The software is publicly available under the limited GNU public license (LGPL). It utilizes a standard Linux kernel, with the CONFIG_PREEMPT_RT patch set, so that real time control software can run in user space and be coded in C or C++. The control loop currently runs at a deterministic rate of 1 kHz. Key functions include coordinate transformations, inverse kinematics, gravity compensation, and joint-level closed loop feedback control. The link between the control software and the motor controllers is a custom USB interface board with eight channels of 16-bit analog output to each joint controller, and eight 24-bit encoder inputs. The board can perform a read/write cycle for all 8 channels in 125μs[13]. The Raven II has been integrated with ROS, which allows easy integration with other robotic software.

III. dVRK System Overview

Fig. 2 summarizes the open source da Vinci Research Kit platform, consisting of the first-generation da Vinci system hardware, motor controller electronics, FPGA firmware and a component based control software stack. The rest of this section gives a brief introduction of the hardware, electronics and firmware of dVRK to provide the background information for subsequent sections. Interested readers are referred to [1] for more details.

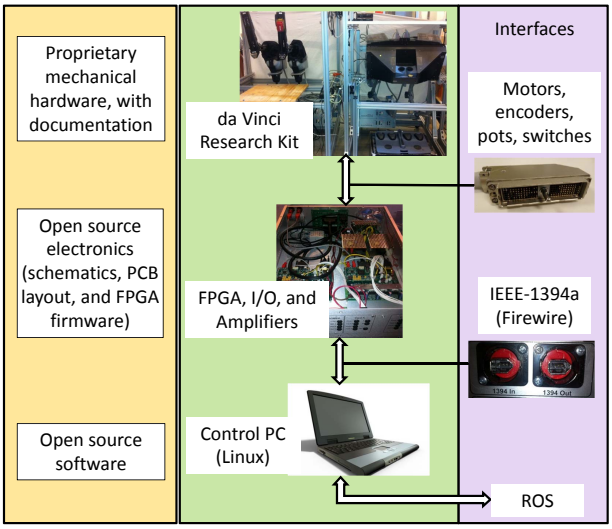

Fig. 2: Overview of the da Vinci Research Kit telerobotic research platform: Mechanical hardware provided by da Vinci Surgical System, electronics by open-source IEEE-1394 FPGA board coupled with Quad Linear Amplifier (QLA), and software by open-source *cisst* package with ROS interfaces [1].

The mechanical hardware can either be obtained from retired first-generation da Vinci Surgical Robot Systems or as a Research Kit provided by Intuitive Surgical, Inc. The Research Kit consists of the following components: two Master Tool Manipulators (MTMs), two Patient Side Manipulators (PSMs), a High Resolution Stereo Viewer (HRSV), a footpedal tray, and documentation (e.g., wiring diagrams, connector pinouts, kinematic parameters). The full da Vinci System may include a third PSM and also includes an Endoscopic Camera Manipulator (ECM), a stereo endoscope, and the passive Setup Joints (SUJs) that support the PSMs and ECM.

The control electronics is based on two custom boards: (1) an IEEE-1394 FPGA board, and (2) a Quad Linear Amplifier (QLA). The schematics, firmware, low-level software interface, and documentation are available on a public git repository. These boards were designed for general mechatronics use, but are well suited for controlling the dVRK. The IEEE-1394 FPGA board contains a Xilinx Spartan-6 FPGA, an IEEE-1394a (FireWire) physical layer chip with two 6-pin connectors, and (as of Rev 2.0) an Ethernet MAC/PHY controller with a single 10BASE-T/100BASE-TX port. The QLA attaches to the IEEE-1394 FPGA board and provides all hardware required for current (torque) control of four DC brush motors, using a bridge linear amplifier design.

The FPGA firmware is implemented in Verilog with three major responsibilities: (1) exchanging data with the PC via IEEE-1394 or Ethernet, (2) interfacing to I/O devices such as encoders and digital-to-analog converters (DACs) for output motor currents, and (3) hardware-level safety checking, such as a watchdog timer and motor current safety check.

The following three sections describe the layers of the software architecture presented in Fig. 1, which focus on hardware interface, real-time control, and system integration.

IV. Scalable and Reconfigurable Distributed Hardware Interface

No matter what software architecture is used, control programs must fetch data from hardware sensors and then send commands to actuators through a hardware communication channel. While the hardware interface can be provided by boards that are installed inside a computer workstation, it is more convenient to distribute the hardware interfaces via a field bus and is especially important for scalability and reconfigurability. This section presents the design goals, an analysis of potential options, and implementation details.

A. Design Goals

One of the most desirable properties of a field bus is to provide deterministic performance with low latency. In our experience, the largest factor that influences latency is the software overhead on the control PC. Certainly, one factor is the choice of communication protocol; for example, it is well-known that the User Datagram Protocol (UDP) has lower overhead (lower latency) than Transmission Control Protocol, Internet Protocol (TCP/IP). But, in our experience, the most critical factor is to minimize the total number of communication transactions performed by the PC. This motivates use of a field bus that supports broadcast, multicast and peer-to-peer transfers.

A second desirable property is for the field bus to support "daisy chain" connection; that is, where one cable connects from the PC to the first interface board, another cable connects from the first board to the second board, and so on. This enables the bus topology to scale with the number of manipulators and facilitates reconfiguration to support different setups. For example, the full da Vinci system, with 6 active manipulators and a passive setup joint structure, requires 13 FPGA/QLA board sets to control the full system (two 4-axis board sets for each active manipulator and one board set for all passive setup joints). Thus, scalability is an important requirement. At the same time, reconfigurability allows multiple users to simultaneously work on a single system by simply introducing another control PC and changing the network cabling and safety/e-stop chain.

Finally, it is necessary for the field bus to provide sufficient bandwidth to support all the hardware on the bus, especially when the goal is to perform even the high-frequency, low-level control on the PC.

B. Design Analysis

Several field buses are available, such as Controller Area Network (CAN) bus, Universal Serial Bus (USB), Ethernet/EtherCAT, and IEEE 1394 (FireWire). The CAN bus is an excellent protocol for control purposes but is limited by its bandwidth (1 Mbps). Although USB provides high bandwidth (480 Mbps for USB 2.0), its polling mechanism means that it is not ideal for real-time applications[14] and it has a poor scalability. Ethernet has sufficient bandwidth (10/100/1000 Mbps) but is typically wired in a "star" topology; supporting a daisy-chain connection would require a high-speed switch on each interface board. In EtherCAT, a master node (PC) periodically initiates a transaction; slave nodes receive, forward and append data packets with the aid of dedicated hardware and software. This design results in the ability to communicate with 100 axes in 100 μs [15]. FireWire is a high speed field bus (up to 400 Mbps for IEEE-1394a) featuring peer-to-peer communication, broadcasting, and physical repeaters at each node to support daisy-chain connection.

While EtherCAT satisfies all the desirable properties of a field bus (i.e., minimize PC transactions, daisy-chain configuration, and high bandwidth), it was a relative newcomer when the design decision was made in 2008 and even today remains a proprietary implementation. We therefore selected IEEE-1394 (FireWire), which also satisfies all desired properties, but requires more implementation effort to reduce the number of transactions on the PC, as discussed in the next section. However, while FireWire was a reasonable option in 2008, today it is less widely available than alternatives such as Ethernet and USB. Thus, we recently added an Ethernet port to the FPGA board so that it can act as a "bridge" between the PC and the FireWire network, as also described below.

C. Implementation

The most straightforward protocol over the FireWire bus is for the PC to individually read from, and write to, each controller board; however, this solution does not scale well because the overhead on the PC increases linearly with the number of boards. We solved this issue by taking advantage of the FireWire broadcast and peer-to-peer communication capabilities [16]. Each control cycle begins when the PC broadcasts a *query* command to all boards and then waits for $5N$ μs, where N is the total number of boards. Upon receiving the *query* command, each board broadcasts its status

(feedback) packet after waiting for $5n$ μs, where n is its node-id ($n = 0 \ldots N-1$). The PC is configured to ignore these responses, but the FPGA firmware on each board maintains a data structure that contains the status received from each board (because the FireWire link layer is implemented in the FPGA, transaction overhead is negligible). After waiting $5N$ μs, the PC reads the complete status information from one board, then computes the control output and broadcasts it as a single packet. Each board extracts its own commands from this packet based on its board ID. This protocol has been shown to enable control rates up to 6 kHz on a dVRK with 8 control boards [16] and is routinely used to achieve 3 kHz control on a full da Vinci at JHU. We have, however, discovered that some PC FireWire adapters do not properly handle the stream of broadcast packets; thus, we also provide an intermediate protocol where the PC individually reads the status from each board and then broadcasts the control output to all boards in a single packet. With this protocol, the maximum control rate is about 1.8 kHz.

As noted above, the FPGA board now includes an Ethernet port to enable it to act as a "bridge". Specifically, the PC can send and receive packets via Ethernet to the first board and then the FPGA firmware on that board communicates with the rest of the boards via the FireWire network. A prototype implementation of the Ethernet/FireWire bridge design is presented in [17]; the implementation is currently being improved to enable any board to serve as the bridge, rather than requiring a dedicated board.

Figure 3 presents a UML class diagram of the interface software that supports the above design. Two bases classes are defined: (1) *BasePort* represents the physical fieldbus port resource, which, depending on the implementation, can be a Firewire or Ethernet port, and (2) the abstract *BoardIO* class that represents the controller board. Currently, there is one derived class, *AmpIO*, that encapsulates the functionality of the FPGA/QLA board set.

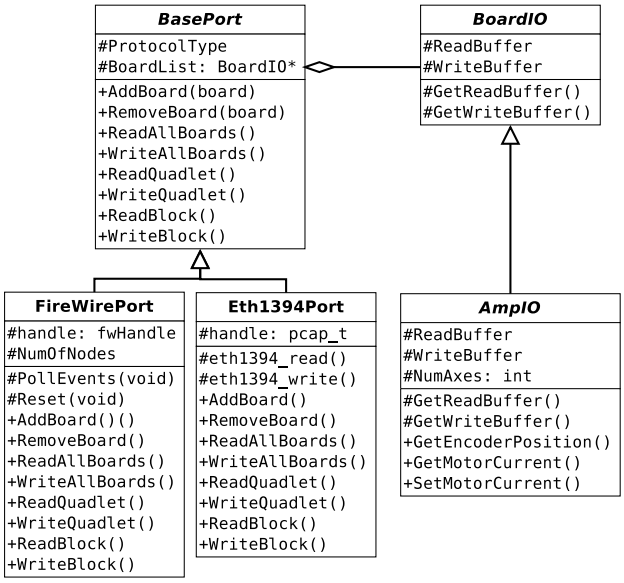

Fig. 3: UML class diagram of interface software (subset of class members shown): the design can scale and support different field bus implementations.

V. Real-time framework for robot control

This section describes the middle layer in the software architecture, which is the real-time framework for robot control. This includes the *Low Level Control* and *Mid Level Control* shown in Fig. 1. The Low Level Control implements the joint controllers for the da Vinci manipulators and is typically configured to run at 3 kHz. The Mid Level Control incorporates the robot kinematics and contains a state machine that manages the robot states (e.g., homing, idle, moving in joint or Cartesian space); it typically runs at 1 kHz.

A. Design Goals

There are two primary design requirements:

1) A component-based framework, with well-defined interfaces between components, to enable different control methods to be easily deployed to the system.
2) Efficient communication between components to support control rates of 1 kHz or more.

These requirements influence the choice of both the execution model and communication paradigm. Specifically, the components can execute as separate processes (e.g., as ROS nodes) or can execute within a single process, using multi-threading. Communication can be implemented as client/server (e.g., remote procedure call) or as publish/subscribe, as exemplified by ROS services and topics, respectively. The following section analyzes the performance tradeoffs of these choices.

B. Design Analysis

We consider two key performance characteristics, which are: (1) the manner in which low-frequency components handle feedback from high-frequency components, and (2) the latency of component communications.

First, we consider the ability to handle data exchange between components with different execution rates in a timely and reliable manner. The key requirement is to deliver the latest data to the consumer component with minimum latency and overhead. In particular, we consider the case where the consumer component (e.g., Mid Level Control) is running at a lower rate than the producer component (e.g., Low Level Control). For a publisher and subscriber system using a simple UDP implementation, the consumer's queue can become full and start to drop new arrival data (head-of-line blocking problem). Besides, UDP does not guarantee data delivery. The ROS subscriber handles this case better by dropping the oldest data in the queue and by using the TCP protocol by default for more reliable data transmission. However, when multiple messages are queued on the consumer component, the registered subscriber function is called multiple times (depending on queue size), creating extra overhead. Setting the receiver queue size to 1 removes this overhead but can result

in intermittent dropped packets; we have observed 4 dropped packets out of 27,282 packets, for a 99.985% delivery rate.

Second, we consider communication latency. As shown in Fig. 4, a ROS publisher and subscriber pair running on the same computer has a mean latency of 244 μs and a maximum latency of 2129 μs. The data is collected by time stamping a ROS message before it is published, having the subscriber run `ros::spin()` (equivalent to busy wait), and computing the time difference between the wall clock time and the stamped time in the subscriber callback function. While this latency is negligible for systems running at slower rates, such as 100 Hz, it is substantial for control loops at 1 kHz or higher. Moreover, this measurement uses a busy wait on the subscriber side and consequently does not consider the additional latency introduced by periodic calls to `ros::spin()`. A repeated measurement with subscriber updates at 1 kHz is shown in Fig. 5. As expected, the mean latency jumps from 244 μs to 792 μs with a 548 μs increase, which is equivalent to half of the node update period. To make things worse, the data does not just flow one-way in robotic control and the subscriber (e.g., low-level control node) typically needs to do some computation on the incoming sensor data and publish the results back to the publisher (e.g., hardware interface node) for execution. This doubles the overall latency time to around 1600 μs, which is well over 1 ms.

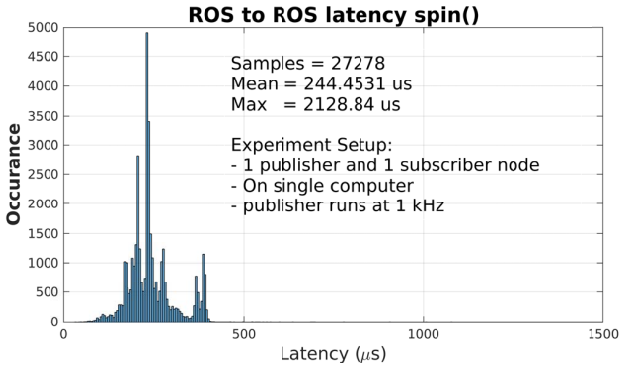

Fig. 4: ROS system publisher/subscriber latency test. Hardware: Intel i7-3630QM Quad-Core 2.4 GHz, 16 GB Memory. Software: Ubuntu 12.04 LTS (Kernel 3.8.0-44-generic), ROS Hydro.

A multi-threaded component-based robotic middleware, such as OROCOS [18] from Katholieke Universiteit Leuven and *cisst* [4] from Johns Hopkins University, can use a lock-free shared memory implementation to minimize the overhead of data delivery and to ensure that the latest data is available to the consumer component. It is true that this approach can face the same data synchronization challenge if the communicating components are in separate threads, but there is the option to chain execution of multiple components into a single thread to avoid this issue, while still maintaining the advantage of a component based architecture. In *cisst*, this is provided by special component interfaces called *ExecIn* and *ExecOut*. The parent component (e.g., I/O component) executes the child

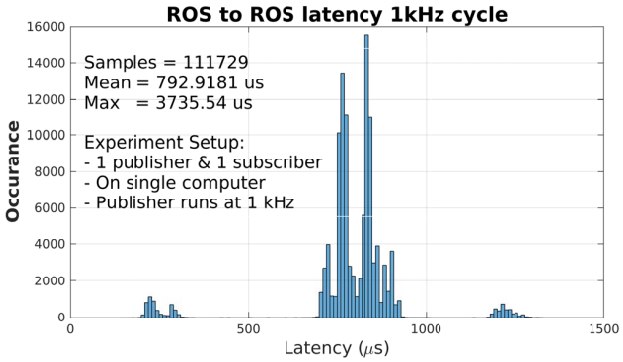

Fig. 5: ROS system publisher/subscriber latency test, subscriber updates at 1kHz, same hardware/software setup as Fig. 4.

component (e.g., low level control) by issuing a *run event*. This feature does not require modification to the component implementation (other than placement of the *RunEvent*) and is activated by connecting the *ExecIn* interface of the child component to the *ExecOut* interface of the parent component. If the *ExecIn/ExecOut* interfaces are not connected during system configuration, separate threads are created for each component and they communicate asynchronously using the same shared memory communication mechanism. Figure 6 shows the data transfer latency between two *cisst* components using the *ExecIn/ExecOut* feature. On average, the latency is 21.3 μs with a maximum value of 115.2 μs. OROCOS RTT provides a similar capability via its *PeriodicActivity* class, which serially executes components with equal periodicity and priority, based on the order in which they are started.

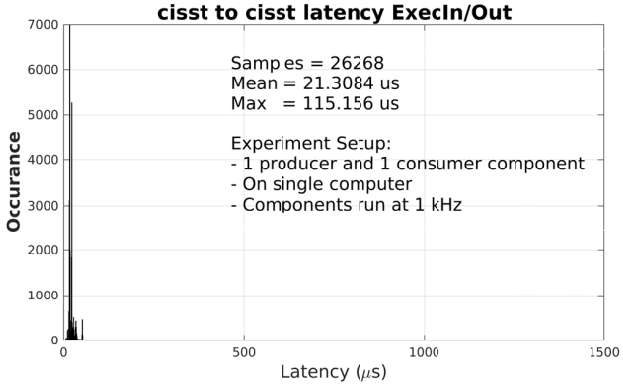

Fig. 6: Communication latency in *cisst*, using *ExecIn/ExecOut* for synchronous communication; components execute at 1kHz, same hardware/software setup as Fig. 4.

C. Implementation

Based on the above analysis, we determined that a shared-memory, multi-threaded design is better suited for the high-frequency, low-latency control requirements for the dVRK, which extend from the hardware interface (Section IV) to the low-level and mid-level control. We selected the *cisst* library due to our familiarity with its design; however, other

frameworks such as OROCOS would also be suitable. As shown in Figure 7, the architecture consists of: (1) one hardware Input/Output (I/O) component, *mtsRobotIO1394* (3 kHz), handling I/O communication, (2) multiple servo loop control components, *mtsPID* (3 kHz, one for each manipulator) providing joint level PID control, (3) mid-level control components (1 kHz, different components for each type of manipulator, such as da Vinci MTM and PSM) managing forward and inverse kinematics computation, trajectory generation and manipulator level state transition, (4) teleoperation components *mtsTeleoperation* (1 kHz) connecting MTMs and PSMs and (5) a console component (event-triggered) emulating the master console environment of a da Vinci system. All of these are connected using *cisst* provided/required interfaces. Note that although they are independent components, the I/O component and the PID components for the manipulators are interconnected via the aforementioned *ExecIn/ExecOut* interfaces to use a single thread, thereby guaranteeing synchronous communication and minimal latency for maximum control performance. In this case, the *RunEvent* is generated by the *mtsRobotIO1394* component after it receives feedback from the controller boards and before it writes the control output. Thus, the *mtsPID* components receive the freshest feedback data and compute the control output, which is immediately sent to the hardware when the *mtsPID* components return the execution thread to the *mtsRobotIO1394* component.

D. Community Extensions

Researchers from the University of British Columbia (UBC) and Stanford University developed a MATLAB Simulink®to C++ interface for controller development on the dVRK [19]. The motivation for this work is that a typical C++ based development cycle involves coding, compiling and debugging, which is time consuming, and any design changes require restart of the robot. On the other hand, MATLAB Simulink provides a block diagram environment to design, evaluate and even update controllers "on the fly", thus enabling researchers to rapid prototype controller designs. The developers created a new *mtsSimulinkController* component to connect the existing software framework to Simulink. This component establishes TCP/IP connections between Simulink blocks and *cisst* components. Conceptually, this is similar to the *cisst*-to-ROS bridge described in Section VI-A. As a proof of concept, the *mtsSimulinkController* component was used to replace the standard *mtsPID* component. This extension has been shared with other researchers in the dVRK community.

VI. System integration via ROS interfaces

Robot Operating System (ROS) is used to provide a high level application interface due to its wide acceptance in the research community, large set of utilities and tools for controlling, launching and visualizing robots, and the benefits of a standardized middleware that enables integration with a wide variety of systems and well-documented packages, such as RViz and MoveIt!. ROS also provides a convenient build system. As noted in the previous section, ROS is fundamentally a multi-process software architecture (though multiple *nodelets* can be used within a single node). While this may have disadvantages for real-time control, in a larger system it has the advantages that it limits the scope of an error to a single process and facilitates software development by minimizing the need to restart and re-initialize the robot (i.e., as long as the robot process is not restarted). This section presents the bridge-based design that enables integration of the *cisst* real-time control framework within a ROS environment, followed by a discussion of the Catkin build system, and some integration examples.

A. CISST to ROS Bridge

To add support for ROS, a bridge based design was implemented. This implementation includes a set of conversion funtions, a *cisst* publisher and subscriber, and a bridge component. The bridge component is both a periodic component (inherits from *mtsTaskPeriodic*) and a ROS node. As an *mtsTaskPeriod* component, it is executed periodically at a user specified frequency and connected, via *cisst* interfaces, to the other *cisst* components. The bridge component also functions as a ROS node with a node handle that can publish and subscribe to ROS messages.

To illustrate this design, consider the example in Fig. 8, which has one *cisst* component connected to a ROS node via a *cisst*-to-ROS bridge. The *cisst* component contains a provided interface with two commands: (1) the `ReadVal1` command to read the value of `mVal1`, and (2) the `WriteVal2` command to write a value to `mVal2`. The component assigns `mVal2` to `mVal1` in its periodic `Run` method. A *cisst* publisher is created in the bridge component that connects to the `ReadVal1` command and publishes to the ROS topic `/Val1`. Similarly, a *cisst* subscriber subscribes to the ROS topic `/Val2` and connects to the `WriteVal2` command. On the ROS side, the node simply subscribes to `/Val1`, increments the received value, and publishes to `/Val2`. At runtime, the bridge node fetches data through the *cisst* interface, converts it to a ROS message, and then publishes the message to ROS. In the reverse direction, the `ros::spinOnce` function is called at the end of the `Run` method, which calls the subscriber callback function, converts data, and triggers the corresponding *cisst* write command. The bridge always publishes at its specified update rate. If the *cisst* component is faster than the bridge component, the bridge only fetches the latest data at runtime, thus throttling the data flow. If the bridge component updates faster, it publishes the latest data at the bridge's rate. For certain applications that require publishing and subscribing at the exact controller update rate, programmers can either create a separate bridge for each *cisst* controller component or directly initialize a publisher node within the *cisst* component and call `publish` and `ros::spinOnce` manually.

B. ROS Ecosystem

In this subsection, the build system, ROS packages and simulation solutions that make the dVRK system ROS compatible are detailed.

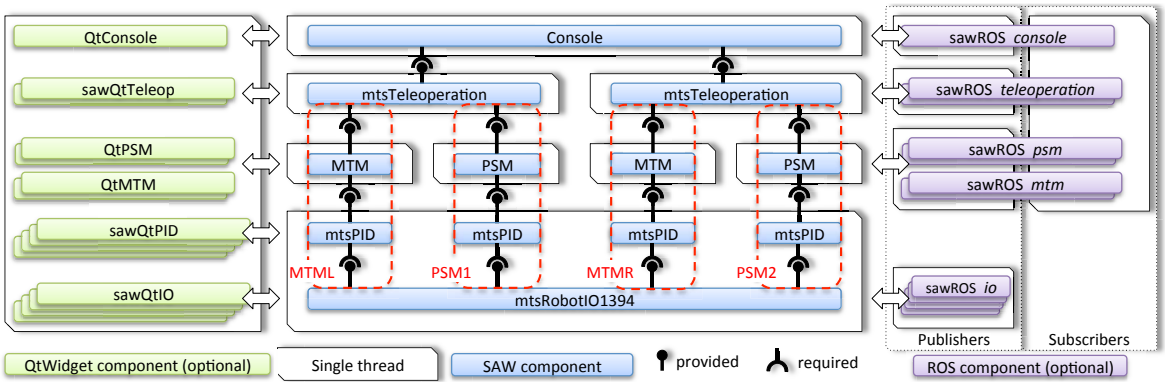

Fig. 7: Robot tele-operation control architecture with two MTMs and two PSMs, arranged by functional layers and showing thread boundaries[1].

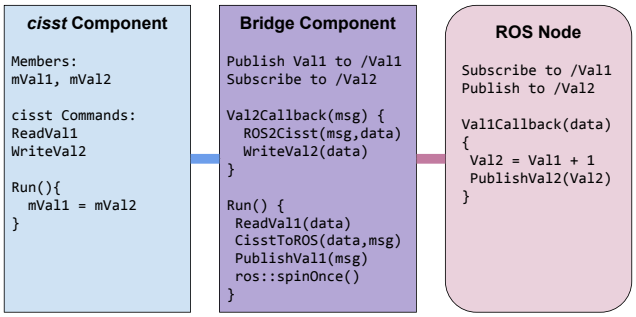

Fig. 8: *cisst*/ROS bridge example: a *cisst* component interfaces with a ROS node using a bridge component. The ROS node subscribes to `Val1`, increments it and publishes to `Val2`.

The *cisst* build system is based on CMake (www.cmake.org), but for the convenience of ROS users, we created a catkin-based solution[20] by making *cisst*, and other packages based on *cisst*, into catkin packages. This allows ROS users to download the dVRK code and use the ROS catkin tools to compile without having to learn the details about how to configure and compile the *cisst* library.

In addition, the MTM, PSM and Setup Joint models have been generated in the ROS Unified Robot Description Format (URDF) and can be used for visualization and kinematic simulation in RViz.

Some use cases that take advantage of the ROS interface and simulation are to use a real MTM and foot pedal as input devices to tele-operate a simulated PSM [21] or alternate slave robot, such as the Raven-II [2]. In fact, over half of the researchers who have dVRK systems have used this ROS interface for their research, mostly by implementing high-level controllers that communicate with the dVRK mid-level controller via ROS.

C. Community Extensions

Several researchers have taken advantage of the ROS interface to implement higher-level controllers or to integrate with other systems. In this section, we highlight a project performed by a high school student during an internship at JHU. This project was implemented with the Python programming language, using ROS to interface between Python and the dVRK C++ software. The student started with a straightforward potentiometer calibration project, where the goal was to more accurately determine the scale and offset with respect to the incremental encoder and physical joint limits. The student wrote a Python script to drive the joint of interest through a list of joint positions equally spaced between the upper and lower joint limits. At each position, the joint is commanded to pause for 5 seconds, during which time the encoder and potentiometer data are collected. The *scale* is computed by finding the slope of the best fit line of the raw potentiometer data to the incremental encoder data. In some cases, a scale correction as high as 2% was observed. The *offset* was obtained by using a custom-designed mechanical fixture to lock the final four PSM axes in a known zero configuration. The offset correction (compared to the offset values initially provided by the manufacturer) typically ranged from 0.1 degrees to 1.5 degrees. The student's Python program also saves the calibrated *scale* and *offset* values back to the XML configuration file. This software and documentation has been contributed to the GitHub repository and has been used by other researchers with dVRK systems. It is also interesting to note that the student was able to perform this work using a single PSM, often while other students were simultaneously using other parts of the system. In fact, we had several occasions where three projects were performed in parallel (e.g., the high school student using one PSM, two other students each using an MTM/PSM pair). This demonstrates the value of an architecture that supports quick and easy reconfiguration.

VII. Discussion and Conclusions

We presented a scalable, reconfigurable, real-time and ROS-compatible software architecture for the da Vinci Research Kit (dVRK), currently installed at more than 25 research institutions worldwide. The software stack is maintained by JHU, with some contributions from the community. Over the past two years, new software releases have occurred approximately every six months.

The architecture was presented as three layers: (1) distributed hardware interface via a high-bandwidth, low-latency fieldbus, (2) real-time component-based framework with multi-threading and thread-safe shared memory communication, and (3) high-level integration with the ROS ecosystem. The *BasePort* and *BoardIO* classes (and derived classes) defined in Section IV represent the transition between the distributed hardware layer and the real-time framework, whereas the *cisst*-to-ROS bridge defined in Section VI provides the interface between the real-time framework and the ROS environment.

The paper also briefly described community extensions within these layers. One observation is that it is more likely for researchers to extend the system at the higher layers (e.g., by integrating the dVRK with other systems and software). Fortunately, due to the wide adoption of ROS, many researchers have sufficient knowledge to accomplish this task. Some research extensions require real-time performance, however, which generally cannot be obtained via the ROS interfaces. In these cases, researchers can extend the real-time layer, but this introduces an additional learning curve for the *cisst* real-time framework. We are currently working to simplify this process by increasing the use of dynamically-loaded plug-in components and Javascript Object Notation (JSON) files for run-time configuration (as opposed to recompiling the entire software stack). We are also aware of an ongoing effort for Real-Time ROS (RTROS) [22], using the Ach library, which provides an identical ROS Application Programming Interface (API) while still meeting hard real-time constraints. In the future, this could be a viable alternative for implementing the real-time layer, with the benefit that the API is already familiar to many researchers. The distributed hardware interface layer is the most difficult to modify because much of it is implemented in FPGA firmware (Verilog programming language); fortunately, because it primarily manages I/O functions, it is unlikely to require modification by researchers.

In summary, the dVRK software architecture has been designed to provide scalable, real-time performance with an optional (but increasingly used) bridge to the ROS environment. Researchers can implement new algorithms within this architecture, taking advantage of the real-time framework when required. While the dVRK already provides a common research platform that enables better sharing of software and replication of results, we are currently partnering with others in the community to broaden the architecture to include other research platforms, such as the Raven II robot and other devices.

ACKNOWLEDGMENT

This work was supported by the National Science Foundation (NSF) via the National Robotics Initiative (NRI) grants 1637789, 120854, and 1327657; Engineering Research Center grant 9731748 (CISST ERC) and supplement 0646678; Major Research Instrumentation (MRI) 0722943; and by NASA NNX10AD17A, NNG14CR58C, and NNG15CR66C. We also thank all contributors to the *cisst* library and dVRK software, including Simon Leonard, Paul Thienphrapa, Long Qian, Kwang Young (Eddie) Lee, Jonathan Bohren, Ankur Kapoor, Tian Xia, and Nick Eusman.

REFERENCES

[1] P. Kazanzides, Z. Chen, A. Deguet, G. S. Fischer, R. H. Taylor, and S. P. DiMaio, "An open-source research kit for the da Vinci® Surgical System," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2014, pp. 6434–6439.

[2] B. Hannaford, J. Rosen, D. W. Friedman, H. King, P. Roan, L. Cheng, D. Glozman, J. Ma, S. N. Kosari, and L. White, "Raven-II: an open platform for surgical robotics research," *IEEE Transactions on Biomedical Engineering*, vol. 60, no. 4, pp. 954–959, 2013.

[3] P. Kazanzides and P. Thienphrapa, "Centralized processing and distributed I/O for robot control," in *IEEE Intl. Conf. on Technologies for Practical Robot Applications (TePRA)*, Nov 2008, pp. 84–88.

[4] A. Deguet, R. Kumar, R. Taylor, and P. Kazanzides, "The *cisst* libraries for computer assisted intervention systems," in *MICCAI Workshop on Systems and Arch. for Computer Assisted Interventions*, Midas Journal: http://hdl.handle.net/10380/1465, Sep 2008.

[5] M. Y. Jung, M. Balicki, A. Deguet, R. H. Taylor, and P. Kazanzides, "Lessons learned from the development of component-based medical robot systems," *J. of Software Engineering for Robotics (JOSER)*, vol. 5, no. 2, pp. 25–41, Sep 2014.

[6] M. Y. Jung, A. Deguet, and P. Kazanzides, "A component-based architecture for flexible integration of robotic systems," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, Taipei, Taiwan, Oct 2010, pp. 6107–6112.

[7] 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*, Kobe, Japan, 2009.

[8] K. Salisbury, W. Townsend, B. Ebrman, and D. DiPietro, "Preliminary design of a whole-arm manipulation system (WAMS)," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 1988, pp. 254–260.

[9] J. Bohren, C. Paxton, R. Howarth, G. D. Hager, and L. L. Whitcomb, "Semi-autonomous telerobotic assembly over high-latency networks," in *11th ACM/IEEE Intl. Conf. on Human-Robot Interaction (HRI)*, March 2016, pp. 149–156.

[10] W. F. Lages, D. Ioris, and D. C. Santini, "An architecture for controlling the Barrett WAM robot using ROS and OROCOS," in *41st Intl. Symposium on Robotics (ISR) Robotik*. VDE, 2014, pp. 1–8.

[11] Willow Garage, "PR2 Manual," 2010.

[12] M. J. Lum, D. C. Friedman, G. Sankaranarayanan, H. King, K. Fodero, R. Leuschke, B. Hannaford, J. Rosen, and M. N. Sinanan, "The RAVEN: Design and validation of a telesurgery system," *The International Journal of Robotics Research*, vol. 28, no. 9, pp. 1183–1197, 2009.

[13] K. Fodero, H. King, M. J. Lum, C. Bland, J. Rosen, M. Sinanan, and B. Hannaford, "Control system architecture for a minimally invasive surgical robot," in *Proceedings of Medicine Meets Virtual Reality*, Long Beach, CA, Jan 2006, pp. 156–158.

[14] N. Korver, "Adequacy of the Universal Serial Bus for real-time systems," University of Twente, Tech. Rep. 009CE2003, 2003.

[15] EtherCAT Technology Group. EtherCAT - the Ethernet Fieldbus. [Online]. Available: https://www.ethercat.org/en/technology.html

[16] Z. Chen and P. Kazanzides, "Multi-kilohertz control of multiple robots via IEEE-1394 (Firewire)," in *IEEE Intl. Conf. on Technologies for Practical Robot Applications (TePRA)*, 2014, pp. 1–6.

[17] L. Qian, Z. Chen, and P. Kazanzides, "An Ethernet to FireWire bridge for real-time control of the da Vinci Research Kit (dVRK)," in *IEEE Conf. on Emerging Technologies and Factory Automation (ETFA)*, 2015.

[18] H. Bruyninckx, "Open robot control software: the OROCOS project," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2001, pp. 2523–2528.

[19] A. Ruszkowski, Z. F. Quek, A. Okamura, and S. Salcudean, "Simulink® to C++ interface for controller development on the da Vinci® Research Kit (dVRK)," in *ICRA Workshop on Shared Frameworks for Medical Robotics Research*, May 2015.

[20] "Catkin command line tools," https://catkin-tools.readthedocs.io, accessed: 2016-10-31.

[21] A. Munawar and G. S. Fischer, "Towards a haptic feedback framework for multi-DOF robotic laparoscopic surgery platforms," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2016, pp. 1113–1118.

[22] J. Carstensen and A. Rauschenberger, "Real-Time Extension to the Robot Operating System," in *ROS Conference*, 2016.