# A Scalable, High-Performance, Real-Time Control Architecture with Application to Semi-Autonomous Teleoperation

by

Zihan Chen

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

October, 2017

# Abstract

A scalable and real-time capable infrastructure is required to enable high-performance control and haptic rendering of systems with many degrees-of-freedom. The specific platform that motivates this thesis work is the open research platform da Vinci Research Kit (dVRK).

For the system architecture, we propose a specialized IEEE-1394 (FireWire) broadcast protocol that takes advantage of broadcast and peer-to-peer transfers to minimize the number of transactions, and thus the software overhead, on the control PC, thereby enabling fast real-time control. It has also been extended to Ethernet via a novel Ethernet-to-FireWire bridge protocol. The software architecture 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.

This architecture has been applied to two semi-autonomous teleoperation applications. The first application is a suturing task in Robotic Minimally Invasive Surgery (RMIS), that includes the development of virtual fixtures for the needle passing and knot tying sub-tasks, with a multi-user study to verify their effectiveness. The second application concerns time-delayed teleoperation of a robotic arm for satellite servicing. The research contribution includes the development of a line virtual fixture with augmented reality, a test for different time delay configurations and a multi-user study that evaluates the effectiveness of the system.

# Acknowledgments

First I would like to thank my advisor Professor Peter Kazanzides. His guidance, motivation, firm and continuous support and encouragement was a constant presence during my Ph.D. and Master work over the past years. He was always willing to share his deep knowledge in electronics, software and robotics, always available whenever I wanted to discuss ideas, issues and progress. I could not have had a better advisor and mentor for my Ph.D. study.

I would like to thank Professor Louis L. Whitcomb for his guidance and for participating on my thesis and GBO committee. I would also like to thank Professor Russell H. Taylor for the wonderful Computer Integrated Surgery course, which introduced me to the medical robotics field and gave me the opportunity to work with Dr. Kazanzides, for advising the virtual fixture for suturing project, and for participating on my thesis committee.

I would additionally like to thank Professor Iulian Iordachita for his help in mechanical design and manufacture, as well as for participating on my GBO committee. I would like to thank Professor Scott Smith for participating on my GBO committee, for sharing his insight into software engineering and for writing my annual review letter. Special thanks to Professor Allison Okamura for advising me during my master program. Thanks to Professor Simon Leonard for sharing his knowledge in software, robotics and computer vision.

Many thanks to Anton Deguet for helping me write and debug numerous lines of code, teaching me almost every aspect of programming, sharing his broad wisdom in software design as well as his appreciation of fine coffee. I am also thankful for the many fun conversations we had. I would also like to thank Balazs P. Vagvolgyi for sharing his expertise in computer vision and software.

I would like to thank members of the Sensing, Manipulation, and Real-Time System (SMARTS) group for all the guidance, help, support and discussion over the years. Special thanks to Dr. Paul Thienphrapa for his patient guidance for software design, FPGA programming and FireWire. Thanks to Dr. Tian Xia for help in compiling and debugging code and advice on Ph.D. life. Thanks to Dr. Min Yang Jung for his help on various software/hardware topics. Thanks to Dr. H. Tutkun Sen and Ehsan Azimi for the endless wonderful conversations. Thanks to Long Qian for his help in developing FPGA firmware. Thanks to Jie Ying Wu for organizing all the lab lunches. Thanks also to the other members of SMARTS: Nathan Cho, Sungmin Kim, Andrew Hundt, Paul Wilkening and Isha Kandaswamy. I was also lucky to be part of the larger Laboratory for Computational Sensing and Robotics. I owe special thanks to Dr. Jonathan Bohren for sharing his insights in software and robotics and introducing the Robot Operating System. Thanks to Dr. S. Swaroop Vedula for his help in designing the user study, as well as analyzing data. Thanks to Dr. Anand Malpani for his help in designing virtual fixtures, and for paper writing. Thanks to Dr. Kelleher R. Guerin for sharing his expertise in mechanical design and robotics. Thanks also to Preetham Chalasani for all the great conversations and the good food.

Finally, I would like to thank my parents, Yuewei Chen and Ping Chen, for their constant love, guidance, encouragement, and support.

# Dedication

I dedicate my dissertation work to my parents for all the help, support, encouragement and love during the long journey. This Thesis would not have been possible without you!

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

A scalable and real-time capable infrastructure is required to enable high degrees-of-freedom systems that need high performance control and force rendering. The specific platform that motivates this thesis work is the open research platform da Vinci Research Kit (dVRK) [48]. The next section presents the rationale for developing a research platform based on the da Vinci Surgical System® (Intuitive Surgical, Inc., Sunnyvale, CA), which is called the da Vinci Research Kit (dVRK). This is followed by a discussion of the challenges involved in designing such a system, including an overview of a system first developed in 2004. This leads to a discussion of different robot controller architectures, which culminated in the selection of a *centralized computation and distributed Input/Output (I/O)* architecture for the dVRK. These sections are provided as background information, as this architectural decision preceeded the work described in this thesis. The contributions of this thesis, with respect to the system architecture, are presented starting with Section 1.4.

## 1.1 The da Vinci Research Kit (dVRK)

Minimally Invasive Surgery (MIS) has the benefits of smaller incisions and faster recovery times. However, traditionally, surgeons face the challenge of a limited and constrained workspace and loss of direct visualization. Some of the limitations have been resolved by the use of robotic devices such as the telesurgical da Vinci Surgical System. Due to the benefits of such systems, telesurgical robotics has been an active research field since the mid-1990s.

While open-source robot software, such as Robot Operating System (ROS) [71], has seen widespread adoption, there are relatively few open hardware/software platforms in widespread use within the robotics research community, especially for telesurgical robots. A platform is considered to be "open" if it allows researchers to modify all levels of the control software. The lack of such open platforms means that it requires significant system integration effort to create a research system and a lot of the research experiments and results are not easily replicated by other researchers.

Telesurgical systems require master input devices, preferably with haptic feedback, and slave (or patient-side) robots with the ability to actuate surgical instruments. Currently, researchers typically choose haptic input devices with open interfaces such as Phantom Omni (now Geomagic Touch) [61] or an Omega haptic interface (Force

Dimension Inc., Switzerland) [53]. On the slave side, researchers have tried to use non-medical robots with open interfaces, such as the Whole Arm Manipulator (WAM) from Barrett Technology, Inc. (Cambridge, MA) [79], the KUKA-DLR Lightweight Robot arm [6], the UR5 or UR10 robots from Universal Robots A/S (Odense, Denmark), or use customized robots as slave robots [30]. Recently, the Raven II research robot [32] has been disseminated to several institutions and is available for purchase from Applied Dexterity, Inc. (Seattle, WA). The Raven II enables researchers to modify the real-time servo control code, which runs on a Linux Personal Computer (PC) and communicates with the hardware (e.g., motors and encoders) via a Universal Serial Bus (USB) interface. An open telesurgical platform can be created from these components, but would likely involve a lot of effort and would not present a unified control framework.

One alternative is to create a research platform from an existing telesurgical system, such as the da Vinci Surgical System. The da Vinci Surgical System can be configured (by the manufacturer) to provide a read-only research interface to both the master and slave manipulators [24]. While useful for some research projects (e.g., skill assessment [1, 91]), this interface does not enable modification of the control algorithms and therefore cannot support research in new control methods, including autonomous or semi-autonomous control. This robot mechanical hardware is becoming increasingly available to researchers via the reuse of retired first generation systems. Because the electronics and software are either proprietary (closed) or not

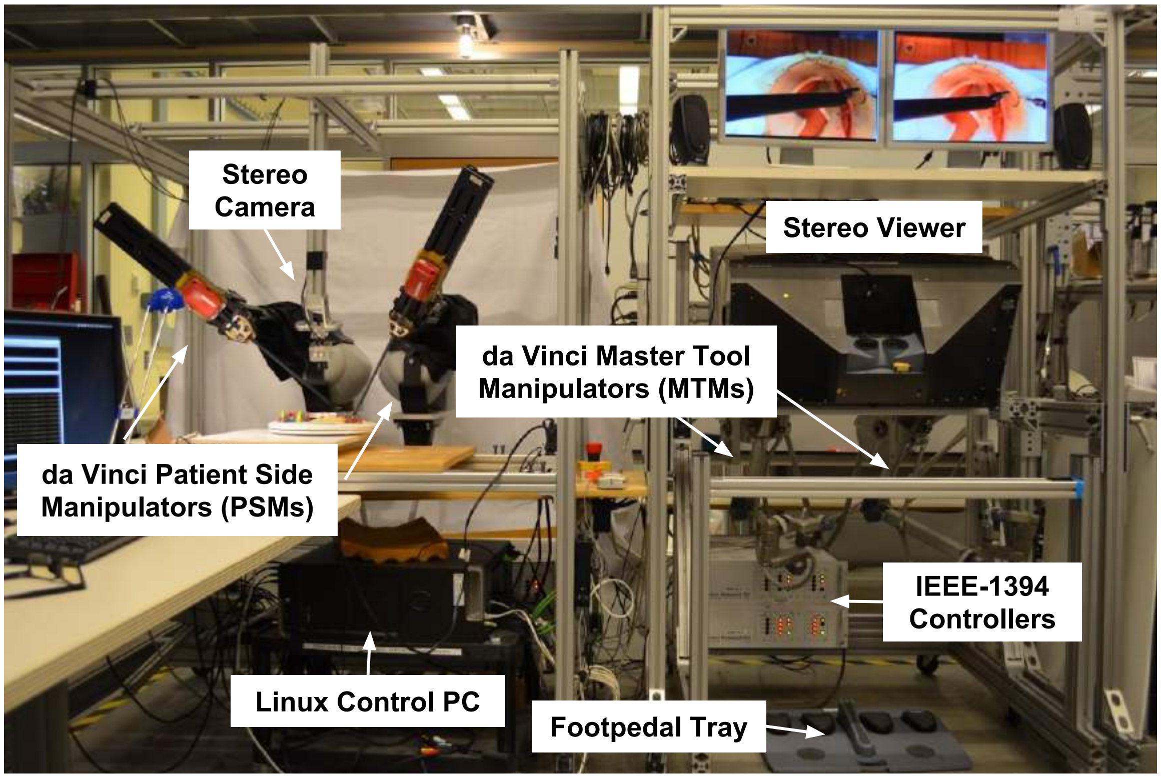

Stereo Camera
Stereo Viewer
da Vinci Master Tool Manipulators (MTMs)
da Vinci Patient Side Manipulators (PSMs)
IEEE-1394 Controllers
Linux Control PC
Footpedal Tray

**Figure 1.1:** Research da Vinci System at JHU: two 7-dof Master Tool Manipulators (MTMs) and two 7-dof Patient Side Manipulators (PSMs), for a total of 28 axes, controlled by eight custom board sets (packaged in 4 enclosures), each consisting of an IEEE-1394 FPGA board mated with a Quad Linear Amplifier (QLA).

included, it motivates the development of a common, open-source, high-performance electronics/software platform for the research community. This thesis concerns the development of the architecture and some use cases towards its application to semi-autonomous teleoperation.

## 1.2 Challenges of a scalable and high performance architecture for dVRK

Despite the fact that the mechanical hardware is readily available, the system mechatronics design faces several challenges due to the high degrees of freedom inherent in the system as well as the design goal to provide a system that enables researchers to easily implement new algorithms at any level of control.

As shown in Figure 1.1, a da Vinci system comprises multiple robotic manipulators, where each of them can have up to 7 DOFs. In a dVRK setup with two Master Tool Manipulators (MTMs) and two Patient Side Manipulators (PSMs), the system has a total of 28 DOFs, and the number of total DOFs can be as high as 39 DOFs including a third PSM and the Endoscopic Camera Manipulator (ECM) in a full da Vinci System setup. This high degrees of freedom requires a solution with high scalability. In fact, back in 2004, a previous version of Industry Standard Architecture (ISA)-based controllers, called the Low Power Motor Controller (LoPoMoCo), were used to drive two MTMs and two PSMs. In this design, all I/O signals, including command and feedback signals, were transmitted in raw analog form between the robot and the control computer over a long cable. A custom I/O device circuit board converted these analog sensor signals into digital data and transformed robot control commands to analog signals; it was directly attached to the computer via the parallel ISA bus. However, parallel buses limit the number of I/O channels that can be connected to one control computer. This ISA-based, centralized I/O design could only control four channels (DOFs) per board due to physical size limitations, and it was discovered that most industrial-grade computers could only reliably drive up to four ISA cards (two manipulators). As a result, with these controllers a teleoperated application involving more than two manipulators needs multiple computers to run, which not only makes the control software design more complicated, but can also introduce delay and negatively affect control performance. This challenge led to a new design with distributed I/O running on a high speed serial field bus.

The high degrees of freedom raises another challenge in terms of control performance. At the servo level, a control loop is typically closed at 1 kHz or higher to achieve a good performance. With the exponential growth of computation power, the computation load of a servo controller with even tens of channels can be finished well within 1 ms. However, the I/O time together with software overhead of such a system can potentially break the timing limit. While in a centralized I/O design, I/O time is less of a concern because the computer has direct bus access, it is a challenge in a system with distributed I/O.

The third challenge comes from the goal to provide researchers easy access to all levels of control in a familiar environment. At the time the distributed I/O system was designed, existing off-the-shelf motor controllers did not allow modification of the low-level servo control algorithm. With the assumption that researchers will be familiar with a Linux development environment, preferably with either the RT-

Preempt patch or a real-time extension such as Xenomai [29] or RTAI [60], a system architecture that enables all software to be implemented in such environment was preferred.

## 1.3 Overview of Architectures

Several design architectures were considered during the design of the controllers for the da Vinci Research Kit. This section provides an overview of these architectures as background material for the work presented in this thesis and is based on the content presented in [49] by Kazanzides and Thienphrapa. Figure 1.2 presents these architectures, which can be categorized based on whether the computation and I/O are centralized or distributed.

Historically, processing and network limitations favored a centralized computation and I/O approach (Figure 1.2a), where all robot cables connect directly to I/O boards inside a central computer via its high-speed internal bus (originally ISA, now Peripheral Component Interconnect (PCI) or Peripheral Component Interconnect Express (PCIe)) as in the initial design. One advantage of this architecture is that the entire control system can be implemented on a familiar development platform (PC), rather than requiring expertise in embedded systems programming. The disadvantage, however, is that a significant amount of cabling is needed to connect the robot sensors and actuators to the electrical interfaces inside the PC, which reduces reliability, increases

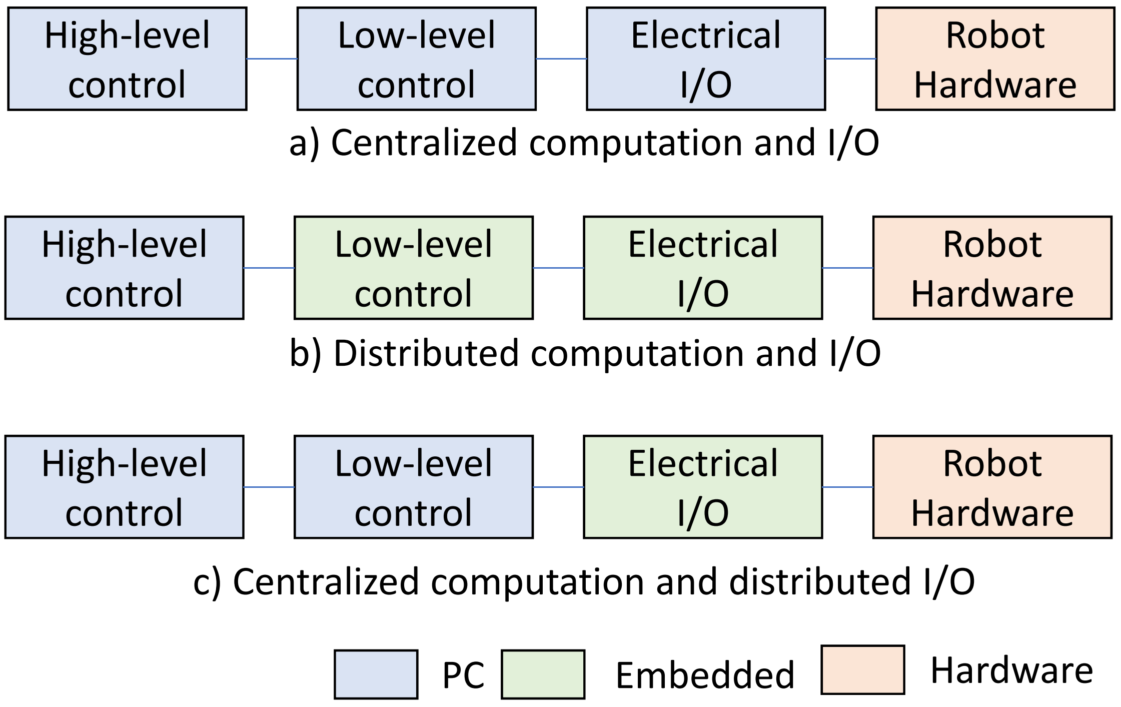


**Figure 1.2:** Three common architectures for robot control.

signal noise, and makes reconfiguration difficult, especially if it is required to open the PC chassis. Another disadvantage, as pointed out earlier, is that the physical form factor of a control computer and its electronic drive capability may limit the number of channels a single computer can control. The architecture, therefore, does not scale well and does not meet the requirement to control a full da Vinci research system.

An alternative approach that scales well is to distribute both computation and I/O (Figure 1.2b), where high-level control is performed on a single computer, with low-level control executed on embedded microprocessors connected via a serial network such as Controller Area Network (CAN), Ethernet, or RS-485. This approach does not require a high-performance (i.e., low latency and high bandwidth) network because the high-level control typically executes at hundreds of Hertz and provides setpoints to the low-level control at this rate instead of directly commanding actuator torque or velocity. This design can easily scale to a large system by adding more controller boards to new manipulators. Meanwhile, distributed I/O helps to clearly confine most wires to local joint sites and a co-located microprocessor guarantees high-performance, low-level control. Nevertheless, a researcher needs to have embedded programming knowledge if he/she intends to change low-level control algorithms and may have to upgrade the embedded microprocessors' firmware every time a change is made.

The availability of high-speed serial networks with real-time performance, such as Ethernet for Control Automation Technology (EtherCAT), SErial Real-time COmmunication System (SERCOS) III, and IEEE-1394 (FireWire), has enabled an approach that can be called centralized computation and distributed I/O [49]. In this approach (Figure 1.2c), the real-time communication network allows all control computations to be implemented on a high-performance computer that contains a familiar software development environment (e.g., Linux, with or without real-time extensions), while preserving the advantages of reduced cabling by distributing the I/O. This allows a researcher to develop both high-level supervisory control and low-level joint control in the same development environment, thus enabling high flexibility in control algorithms while maintaining precise real-time hardware control. This is particularly useful for developing haptic interactions and virtual fixtures. Several systems have adopted this architecture. The WAM [78] distributed motor amplifier I/O boards to each joint and interconnected them using CAN bus. Pratt reported a system that uses IEEE-1394a to communicate between a control PC and distributed Field-Programmable Gate Array (FPGA) boards in a 12-axis biped robot system [68]. The MIRO surgical robot developed by the German Aerospace Center (DLR) uses SpaceWire, a 1 GB/s full duplex serial link with latency less than 20 us, to connect distributed FPGA-based I/O boards to a centralized control PC, running the QNX real-time operating system [31]. Among the Ethernet-based real-time protocols, EtherCAT appears to be gaining the widest deployment. As an example, Willow Garage used EtherCAT to close a 1 kHz loop between a control PC (with real-time operating system) and the encoders and motors in its two-armed mobile robot system (PR2) [75].

## 1.4 Proposed Approach

The centralized processing and distributed I/O architecture was implemented by designing custom electronics that uses an FPGA to provide direct, low-latency, interfaces between the high-speed serial network (IEEE-1394a, in our case) and the I/O hardware. The FPGA board is referred to as IEEE-1394 FPGA board or slave node, given that it is a node on the FireWire bus. IEEE-1394a was chosen because it is widely available, has high performance (up to 400 Mbps), supports daisy-chaining at the physical layer, and there is ample documentation [2] to enable implementation of the link layer protocol on an FPGA. The potential disadvantages of IEEE-1394a are the lack of high-flex cables and the length limits that make it difficult to route cables inside a robot arm. These disadvantages are not relevant for controlling the da Vinci robot, since it is not feasible to place the controller boards inside the robotic arms.

As stated before, the challenge of a centralized processing and distributed I/O design is that the existence of software overhead could potentially break the timing limit on a large system with many controller boards. In the case of a FireWire transaction, even though the data transmission is fast (less than 1 $\mu s$ for a packet smaller than 50 bytes), the operating system introduces a large software overhead in each FireWire transaction. That said, the key strategy to get an efficient I/O is to minimize number of transactions from the control computer. This thesis proposes a broadcast based protocol that reduces the number of transactions to three regardless of the number of boards under control. The details of this protocol are presented in Section 2.5. Another contribution of this thesis is the implementation of an Ethernet/FireWire bridge that enables high-bandwidth control of multiple axes over standard Ethernet, as described in Section 2.6.

In addition, this thesis contributes to a component-based multi-threaded software architecture for servo and mid-level control of the robot system using the *cisst* library, as described in Chapter 3. For high-level interface, an additional component is developed to bridge the low level control to a publisher/subscriber based ROS environment.

This hybrid approach is proposed for the scalability of a multi-threaded, component-based architecture that has low latency and high performance for real-time control while still benefitting from the flexibility and interactivity (e.g., scripting languages and MATLAB interfaces) available via a publisher/subscriber ROS interface.

## 1.5 Applications

Powered by the proposed control/software architecture, the da Vinci Research Kit can achieve a servo update rate of 3 kHz or higher even with a full system setup with 39 active axes from 2 master manipulators (7 DOFs), 3 slave manipulators (7 DOFs), and 1 endoscope control arm (4 DOFs). The da Vinci master manipulators are cable driven, back-drivable impedance type robot arms. This design, coupled with the proposed high performance control architecture, makes it an ideal research platform for haptic rendering and virtual fixture assistance. In this thesis, two potential applications using this platform are examined: one (Chapter 4) is in the medical space using a da Vinci to evaluate the benefits of virtual fixtures in a suturing task for novice users and the other (Chapter 5) is a space application involving time-delayed teleoperation of a robot for refueling aging satellites.

## 1.6 Broader Impact

The application of the system is not limited to the two presented in this thesis. At the time of writing, with more than 25 dVRK systems installed in research institutions across the world, the proposed architecture has been used by researchers to investigate a wide range of applications and research areas. For example, Murali and Sen et al. [65, 83] from the University of California Berkeley used the system to investigate automating cutting and suturing tasks using learning by observation; Mohareri et al. [63] from the University of British Columbia explored the applicability of an asymmetric force feedback control framework for bimanual robot-assisted surgery and Wang et al. [95] from Vanderbilt University and Johns Hopkins University updated virtual fixtures from exploration in force-controlled model-based telemanipulation.

## 1.7 Overview of Contributions

The thesis is composed of the following contributions.

### 1.7.1 A broadcast-based fieldbus communication protocol

Within the proposed architecture, we developed a broadcast based communication protocol that reduces I/O communication time and scales well with the number control nodes by minimizing the number of transactions initiated by the control computer. Using the protocol, a full da Vinci system with 12 nodes can close its servo loop at up to 3 kHz.

### 1.7.2 A bridge design to enable real-time control over a conventional network

We have designed a real-time control network based on Ethernet and FireWire, where Ethernet provides a convenient, cross-platform interface between a central control computer and a FireWire subnetwork that contains multiple distributed I/O control boards. This design benefits from the availability of real-time Ethernet drivers and utilizes the broadcast protocol for real-time performance. This bridge design approach can be extended to other fieldbuses.

### 1.7.3 An analytical model of the timing performance for the proposed protocols and EtherCAT

We developed an analytical model for the timing of the proposed protocols (i.e., broadcast-based protocol without and with Ethernet) and for EtherCAT. Further, we estimated the model parameters from experimental data and then used the models to compare our proposed protocols to the state-of-the-art EtherCAT fieldbus. The comparison shows that our proposed protocols have comparable performance.

### 1.7.4 A scalable and extensible software architecture

The dVRK software architecture is composed of a component-based low and mid level control for performance and a high-level API using ROS interfaces connected via bridge components. This thesis contributed elements of this architecture. Specifically, it develops a design pattern for managing multiple controller boards over a single shared resource (i.e., the fieldbus). The design keeps the boards as separate objects, while still allowing efficiency of broadcast-based transactions that, by necessity, combine data to/from all boards. It also contributes ROS interfaces via a cisst-to-ROS bridge component. The bridge component was developed with help from Anton Deguet.

### 1.7.5 Virtual fixtures for suturing and knot tying tasks

We have implemented virtual fixtures (VF) to assist users during the needle passing and knot tying sub-tasks on a teleoperated robotic system. A user study has been conducted and the result indicates that VF can improve users' performance in terms of better needle exit accuracy in a needle passing sub-task and shorter task completion time and fewer slips in a knot tying subtask.

### 1.7.6 Virtual fixtures for time-delayed teleoperation for satellite servicing

The dVRK master console has been used to teleoperate a WAM robot in a satellite servicing task under 4 seconds video and telemetry time delay. Within this ongoing project, we implemented a model-based telemanipulation framework with haptic feedback and augmented reality and performed a multi-user study to demonstrate that the assistance can significantly reduce task completion time and overall operator workload. Specific contributions include a user interface mechanism for intuitive specification and adjustment of a line virtual fixture on the master console, as well as the multi-user study itself.

# Chapter 2

# System Architecture

This chapter presents research contributions related to system architecture. The first contribution is a broadcast-based FireWire communication protocol for scalable real-time performance. Then we describe a bridge design that enables real-time control over a conventional Ethernet interface by leveraging the previously proposed protocol. Timing performance of both designs are evaluated analytically and experimentally and compared to EtherCAT.

## 2.1 Introduction and Overview of Thesis Contributions

Our goal is to have a scalable, high-performance and programmer friendly system that can support robotic systems with many degrees of freedom (DOFs). A motivating example is the full da Vinci system that consists of six robot arms, for a total of 39 DOF. As discussed in Chapter 1, a centralized computation and distributed I/O approach was selected because it allows all control algorithms to be implemented on a PC in a familiar software development environment while preserving the advantage of reduced cabling by distributing the I/O. It is not surprising that such a system requires not only careful fieldbus selection, but also an optimized communication protocol and firmware, as well as a software architecture that can both scale as systems grow and interface with high level control software for easy programming.

This chapter begins with the historical context that served as the starting point for the work described in this thesis, which led to the selection of a distributed architecture based on the IEEE-1394a (FireWire) fieldbus. This is followed by a survey of fieldbus options, including FireWire, and a description of the FireWire-based controller that had been developed prior to the start of this research. Analysis of the initial implementation revealed limitations, especially when attempting to scale to a system with 39 DOF. This led to the development of two contributions in this thesis:

1. A broadcast-based communication protocol for scalable real-time performance. This protocol utilizes the broadcast and peer-to-peer transfer capabilities of FireWire bus and scales well with increase of nodes under control. This work was published in [19]. Credits: Zihan Chen.

2. A bridge design to enable real-time control over a conventional Ethernet inter-

face. The control PC connects to the bridge board via Ethernet, which forwards command packets to control nodes on the FireWire bus, collects robot status and sends them back to the PC. The work was published in [70] Credits: Developed in a collaboration with Long Qian. The general concept and design was developed by Zihan Chen. Long Qian implemented the initial bridge firmware and assisted with experiments.

Although these two contributions solved the identified problems, it is desirable to re-evaluate the use of FireWire when compared to other alternatives available today, which were reviewed in the fieldbus survey. In particular, we consider the popular EtherCAT fieldbus, which at the present time (2017) is considered to be a leading fieldbus for real-time control. We conclude with a comparative analysis, based on analytical models and experimental data, of EtherCAT to our FireWire and Ethernet/Firewire distributed systems, which is the third contribution of this chapter. Credit: Zihan Chen conducted the experiment and analysis.

## 2.2 Historical Context

The work described in this thesis began in 2011, when several institutions expressed interest in replicating the research da Vinci system developed at the Johns Hopkins University (JHU). At that time, however, the system was nearing obsolescence, as it relied on custom controller boards, called the Low Power Motor Controller

(LoPoMoCo) [46], that were developed in 2004 and interfaced to a PC via the ISA bus. The LoPoMoCo had initially been developed to control a small snake robot and, subsequently, an updated controller based on IEEE-1394a (FireWire) had been developed for a newer version of the snake robot. FireWire had been selected due to its fast transmission speed, deterministic media access control protocol and its support of daisy-chain topology. Some early results from using this controller with a 7 dof snake robot were presented in 2008-2011 [49, 85, 86, 87]. This experience led to the development of a general-purpose FireWire-based motor controller, consisting of an FPGA board and a Quad Linear Amplifier (QLA) board. This FPGA/QLA controller was well-suited to control the da Vinci robot and thus was selected as the controller to use when replicating the research da Vinci system, which has since been called dVRK.

## 2.3 Fieldbus Survey

Although FireWire was selected for the dVRK controller, it is prudent to keep abreast of the state-of-the-art among real-time control fieldbusses, which is the goal of this section.

The beginning of fieldbus dates back to the first industrial networks in the 1970's [88]. It creates a network where field devices such as sensors, actuators, controllers, human-machine interfaces, etc., are connected. With over 30 years of history, a great number of fieldbus solutions have been proposed to meet the requirements of a variety of applications. These fieldbus solutions differ in their transmission medium, maximum transmission speed, medium access control protocol, physical connector, and latencies. For the distributed I/O and centralized processing architecture that we adopted, both input data and control command data are transmitted from and to the control nodes over the fieldbus within the high-frequency, low-level control loop running on a control computer. To be able to close the loop at more than 1kHz, the fieldbus must have a high-enough bandwidth and low latency by design. In the rest of this section, a few fieldbus candidates including CAN, USB, Ethernet, EtherCAT, and FireWire are surveyed.

CAN bus was designed at Robert Bosch GmbH in mid 1980s [25] to fulfill the requirements of automotive applications. Its message-based protocol design allows micro-controllers and devices to communicate with each other without a host computer and is ideal for control purposes and has been used in robot manipulators like the 7 DOF WAM robot [78]. Despite this, CAN bus can only transmit data up to 1 Mbps (i.e., 1000 bits or 125 bytes per ms), which largely limits its scalability in real-time control applications with multiple manipulators. In fact, even for the 7 DOF WAM robot, the CAN bus transmission time alone takes 850 $\mu$s [4] without PC software latency, which makes it challenging to meet a 1 kHz control update rate.

USB is an industrial standard, designed for computer peripherals (e.g., keyboards), that defines cables, connectors and communication protocols. The earlier USB 1.x version could only transfer data at low-speed (1.5 Mbps) and full-speed (12 Mbps), but the newer USB 2.0 standard can transfer data at high-speed (480 Mbps), which is ideal for control communication with large data size. In a USB system, there two types of devices: a single host device and up to 127 USB slave devices. The devices are connected to the host via USB interconnect in a tiered star topology as shown in Figure 2.1. The hub device expands a single USB port into several so that more slave devices can connect to a host device. The host device manages the bus and initiates all communication, and the slave devices only respond when queried. This design implies that a slave device cannot communicate with another slave device directly, therefore, it lacks peer-to-peer transfer capability. In a control application, the number of transactions from the control PC (the USB host) increases linearly with the number of slave nodes under control. Moreover, as shown in Figure 2.2, data is transferred over the USB bus in frames, whose duration is set to 1 ms for low-speed or full-speed transactions and 125 $\mu s$ for high-speed micro-frames in one direction. Hence, the minimum round-trip transaction latency is 250 $\mu s$ for the high-speed bus, without considering software latency. It may be fast enough to control a small number of nodes, however it does not scale well. For these two reasons, USB is not suitable for real-time control of large systems.

Similar to USB, IEEE 802.3 (Ethernet) has the advantage of market dominance and sufficient bandwidth (10/100/1000 Mbps) for high performance communication. However, the standard Ethernet protocol introduces some challenges. First, the orig-

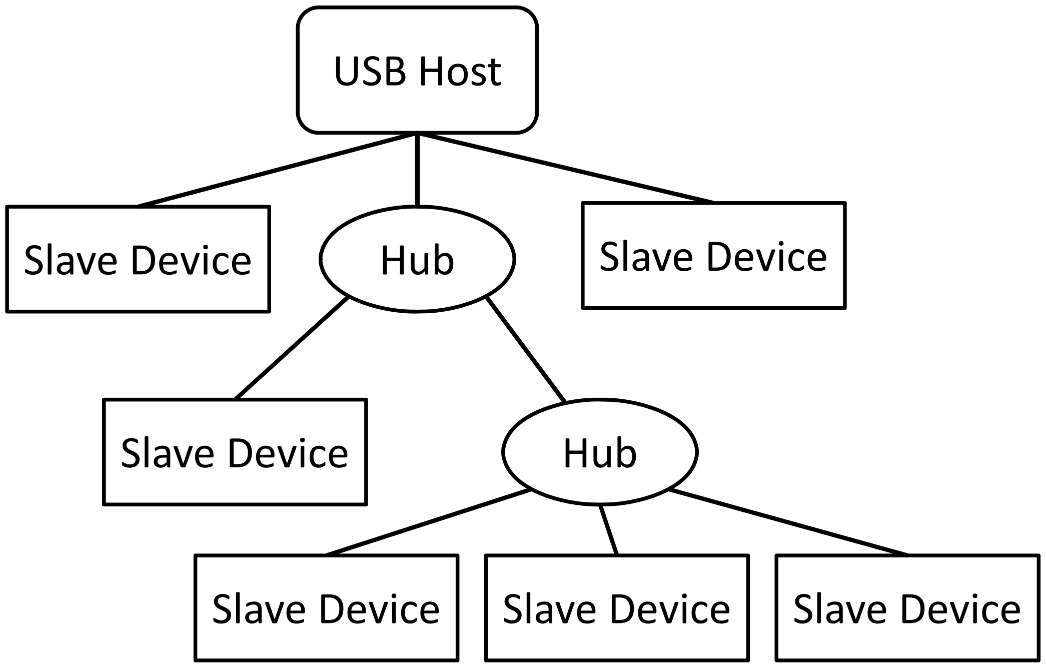


**Figure 2.1:** USB tiered star topology: a host device, multiple slave devices and USB hubs

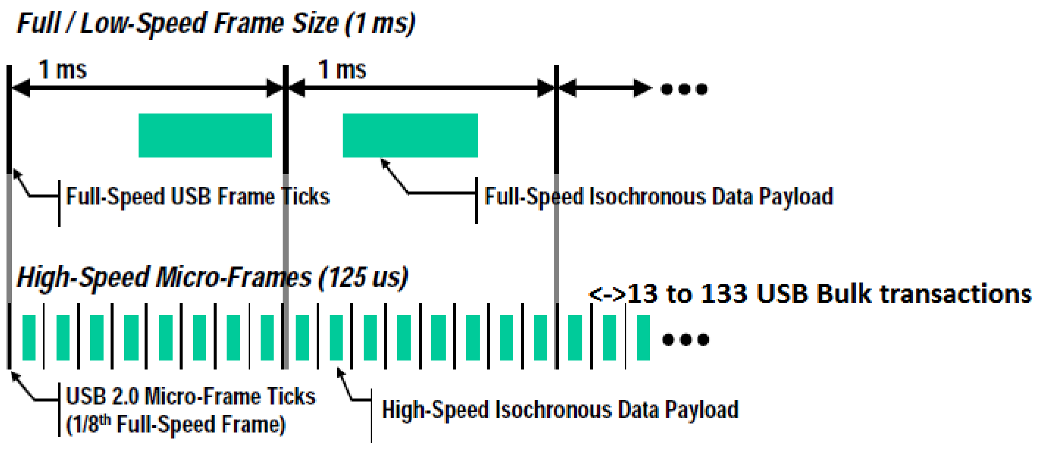

Full / Low-Speed Frame Size (1 ms)
1 ms
1 ms
Full-Speed USB Frame Ticks
Full-Speed Isochronous Data Payload
High-Speed Micro-Frames (125 us)
<->13 to 133 USB Bulk transactions
USB 2.0 Micro-Frame Ticks
(1/8th Full-Speed Frame)
High-Speed Isochronous Data Payload

**Figure 2.2:** USB transaction frame and micro-frame latencies [54]

inal Ethernet was a half-duplex design and used a Carrier Sense Multiple Access/ Collision Detection (CSMA/CD) media access control (MAC) protocol. When a node on the Ethernet detects a collision, it drops the sending/receiving packet, waits for a random period of time based on a binary exponential backoff algorithm and then tries to retransmit. The backoff algorithm controls the medium load, but introduces the possibility of random transmission time; this non-deterministic behavior is not suited for real-time control. One solution with half-duplex Ethernet is to use an isolated network with a self defined media access control algorithm that avoids all potential collisions [50, 81]. The non-deterministic nature of CSMA/CD is less of an issue today, as most Ethernet installations run in full-duplex mode and use point-to-point connections between nodes, leading to a "star" topology. This topology does not scale well, and typically requires high-speed switches to support a daisy-chain connection.

Several Ethernet variations have been developed and employed in some industrial control applications. Ethernet POWERLINK (ethernet-powerlink.org) ensures its real-time determinism by extending the standard Ethernet Data Link Layer with an additional bus scheduling mechanism. Each POWERLINK network contains exactly one Managing Node (MN) responsible for managing the scheduling of the basic bus cycle and several application specific Controlled Nodes (CN). As shown in Figure 2.3, the MN starts a bus cycle by sending out a Start of Cycle (SoC) frame to all nodes, then polls data from each CN in the isochronous phase and finally sends out a Start of Asynchronous (SoA) frame to allow CNs to transmit non-time critical data in the asynchronous phase. Performance-wise, Baumgartner and Schoenegger [5] reported a 250 $\mu s$ cycle time on a system with 1 MN and 3 CNs on Fast Ethernet (100 Mbps). Another variation is the third generation SERCOS (IEC1491) [58]. SERCOS was created in the 1980s by a consortium of machine-tool and numerical-control manufacturers and designed for high-speed serial communication and motion control. Traditionally, its transmission medium is optical fiber with transmission rates of 2/4/8/16 Mbps, but its recent combination with 100 Mbps Ethernet has endowed it with the ability to cycle at 250 $\mu s$ with 70 drives (12 bytes). A relatively new real-time Ethernet protocol is EtherCAT, which was first demonstrated in 2003. Compared with a time-division access protocol like POWERLINK, the EtherCAT master node pre-configures data input and output of each slave node on the bus and only transmits one assembled Ethernet packet. Slave nodes extract input data from, fill in output data to and forward data packets on the fly with the aid of dedicated hardware and software, much like a train going through stations (slave nodes). The EtherCAT Technology Group reported that it can communicate with 100 Servo-Axis nodes (each with 8 byte input and output) in 100 $\mu s$. In 2010, EtherCAT was deployed on the multi-DOF Personal Robot 2 (Willow Garage, Palo Alto, California) [96].

IEEE-1394 (FireWire) is a modern high-speed serial network that supports up to 400 or 800 Mbps for 1394a or 1394b, respectively. Compared with Ethernet, IEEE-1394 supports fair bus access and is deterministic. The protocol supports real-

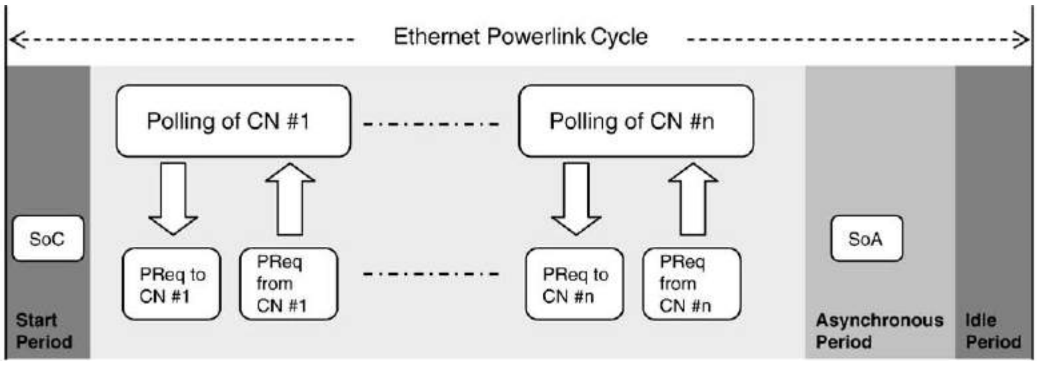


**Figure 2.3:** Ethernet Powerlink bus cycle with three phases: 1) Start Phase, 2) Isochronous Phase, and 3) Asynchronous Phase [15]

communication with guaranteed 125 $\mu s$ bus cycles in isochronous mode and fast concatenated asynchronous transactions. At the physical layer, a FireWire chip repeats data signals; this design allows multiple FireWire nodes to be connected in a "star" topology or daisy-chained together. The latter topology is desired as it reduces the number of cables from the control PC to the robot to one, which allows a significant cabling reduction compared to connecting a separate network cable to each I/O board. FireWire has been shown to be an effective solution for real-time control [80, 100] and by its use in fly-by-wire systems [3]. A more detailed description of the FireWire protocol is presented in Section 2.4.1 to give sufficient background for the work presented in this thesis.

## 2.4 Distributed I/O System

Our distributed I/O system, shown in Figure 2.4, is a complete open source design, with the schematics, layout, and FPGA firmware (Verilog) available on GitHub. The controller consists of two boards, an IEEE-1394 FPGA board and a Quad Linear Amplifier (QLA), that are mated via two 44-pin connectors. Most of the 88 signals are connected directly to I/O pins on the FPGA; the rest are used for power (+3.3V, +5V) and ground. This design allows researchers to create alternate I/O boards (to replace the QLA) to satisfy different hardware requirements, or to design a new FPGA board to introduce a different communication network.

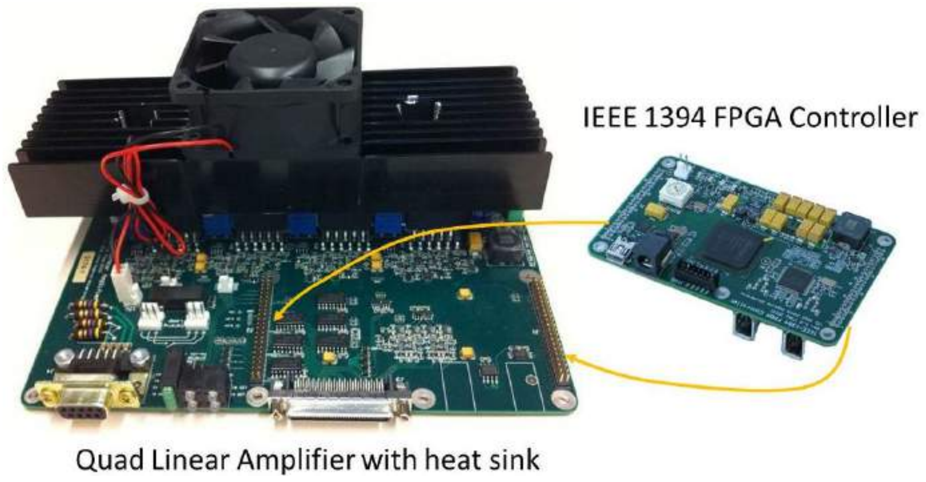

IEEE 1394 FPGA Controller
Quad Linear Amplifier with heat sink

**Figure 2.4:** IEEE-1394 FPGA board and Quad Linear Amplifier (QLA)

The IEEE-1394 FPGA board contains a Xilinx Spartan-6 XC6SLX45 FPGA, configuration PROM, IEEE-1394a physical layer (PHY), two IEEE-1394a 6-pin connectors, a low-speed USB interface (virtual COM port), and required power supplies. It contains two 44-pin connectors that provide power and FPGA I/O to a companion board, such as the QLA. It also contains a 16-position rotary switch for board identification.

The QLA board provides all hardware required for current (torque) control of four DC brush motors, using a bridge linear amplifier design (Figure 2.5). Each of the four channels contains the following components: one 16-bit digital-to-analog converter (DAC) to enable the FPGA to set the desired motor current, two 16-bit analog-to-digital converters (ADCs) to digitize the measured motor current and an external analog sensor (e.g., potentiometer), differential receivers for one quadrature encoder with A, B, and Z (index) channels, two OPA-549 power operational amplifiers (op amps) to provide bi-directional control of a motor from a single power supply (up to 6.25 Amps at up to 48 Volts), digital inputs for one home and two limit switches (these can also be used as general-purpose inputs) and one digital output. The board also contains a software-controlled safety relay, which allows the software to disable the motor power supply, and two heat sink temperature sensors.

This is a general-purpose mechatronics system, but currently its primary application is to control research systems based on the mechanical components of the first-generation da Vinci® Surgical System [17, 48], as shown in Figure 1.1. The low-level control software is implemented on a Linux PC, which is connected via a daisy-chain to several FPGA-QLA board sets, as illustrated in Figure 2.6. This system has been replicated by the dVRK consortium and provided to more than 25 institutions, producing a research community around a common hardware and software platform.

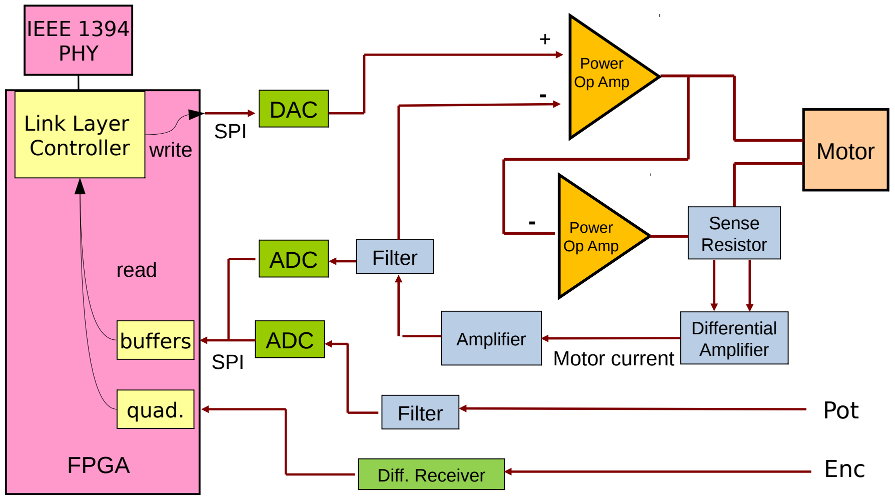


**Figure 2.5:** Block diagram of I/O devices (digital I/O, safety relay, and temperature sensors not shown)

## 2.4.1 Introduction to IEEE 1394

The IEEE-1394 interface [34] is a high-speed, peer-to-peer, full-duplex fieldbus with low overhead that is well suited for real-time control applications. It is a Control and Status Register (CSR) architecture with a tree-like topology that supports up to 64 nodes on a single bus. The IEEE-1394a physical medium transmits data at a speed of up to 400 Mbps. In later specifications (IEEE-1394b), the bus can support data transfers of 800 Mbps and even up to 3.2 Gbps.

As shown in Figure 2.7, FireWire supports two types of transactions: asynchronous and isochronous. It operates based on a 125 $\mu s$ bus cycle (8 kHz), which is triggered by a cycle start packet followed by an isochronous period and then an

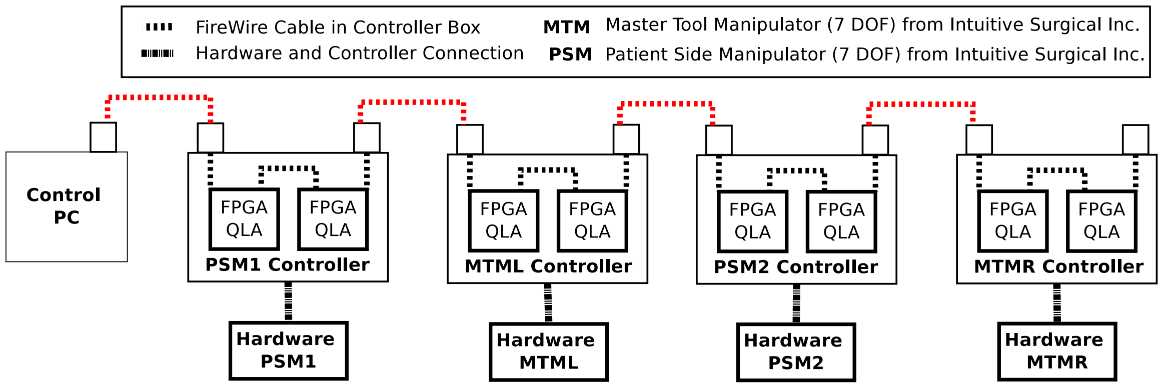

FireWire Cable in Controller Box
Hardware and Controller Connection
MTM Master Tool Manipulator (7 DOF) from Intuitive Surgical Inc.
PSM Patient Side Manipulator (7 DOF) from Intuitive Surgical Inc.
Control PC
FPGA QLA
PSM1 Controller
MTML Controller
PSM2 Controller
MTMR Controller
Hardware PSM1
Hardware MTML
Hardware PSM2
Hardware MTMR

**Figure 2.6:** Hardware architecture: one control PC and 8 IEEE-1394 FPGA/QLA board sets controlling the 4 da Vinci manipulators (7 DOF each).

asynchronous period. An isochronous transaction running at 8 kHz has a reserved bandwidth, and can only happen within the isochronous period. It uses a channel number to address its target nodes and requires no acknowledgment or response packet. Despite its high frequency, an isochronous transfer has no guarantee of data delivery and can suffer from cycle start packet time drifting. This makes it a natural choice for video and audio streaming applications, rather than for real-time control. Asynchronous transactions can only occur in the asynchronous phase after isochronous transactions have completed. Unlike the isochronous transactions, asynchronous transactions use a 64-bit address for data transfer. The whole FireWire bus network can be mapped into the 64-bit address space, with 10 bits for the bus number, 6 bits for the node number and 48 bits for the node address. An asynchronous transaction is designed to be error free by requiring an acknowledgment packet for each data transmission and a response packet for every asynchronous request. Often it is split into two subactions: a request and a response, allowing other asynchronous subactions in between. Asynchronous subactions are separated by short idle periods called subaction gaps (see Figure 2.7) for bus arbitration. Asynchronous transactions are typically used for control commands and reliable message transmission.

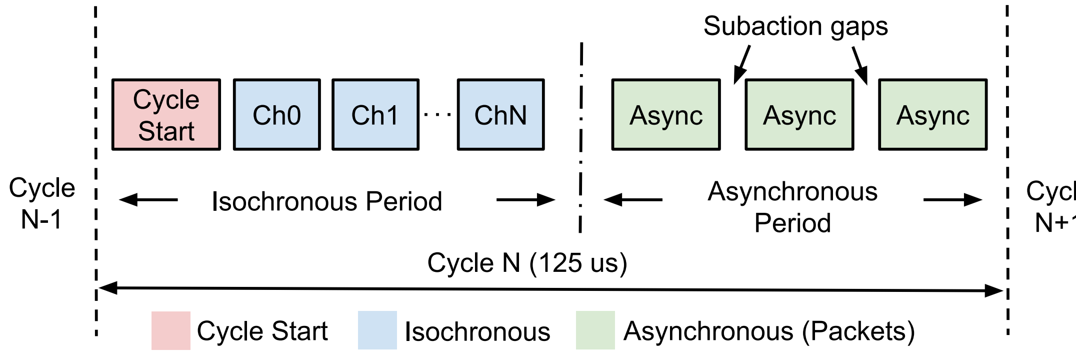


**Figure 2.7:** IEEE-1394 cycle with isochronous and asynchronous transactions

Similar to other network interfaces, the FireWire specification has defined four protocol layers to simplify both the hardware and software implementations (Figure 2.8). Each layer defines a set of associated services. The bus manager manages the FireWire bus configuration and other resources such as the channel number and isochronous bandwidth. The transaction layer only provides service to software drivers for asynchronous transactions including read, write and lock operations. The link layer is in between the transaction layer and the physical layer. It translates asynchronous requests and responses to FireWire packets and sends to the physical layer. For isochronous transactions, the software driver will directly interact with the link layer controller. Finally, the physical layer is the electrical and mechanical interface for data transmission and reception. It also provides bus arbitration services and ensures that only one FireWire node transfers data on the bus at a time.

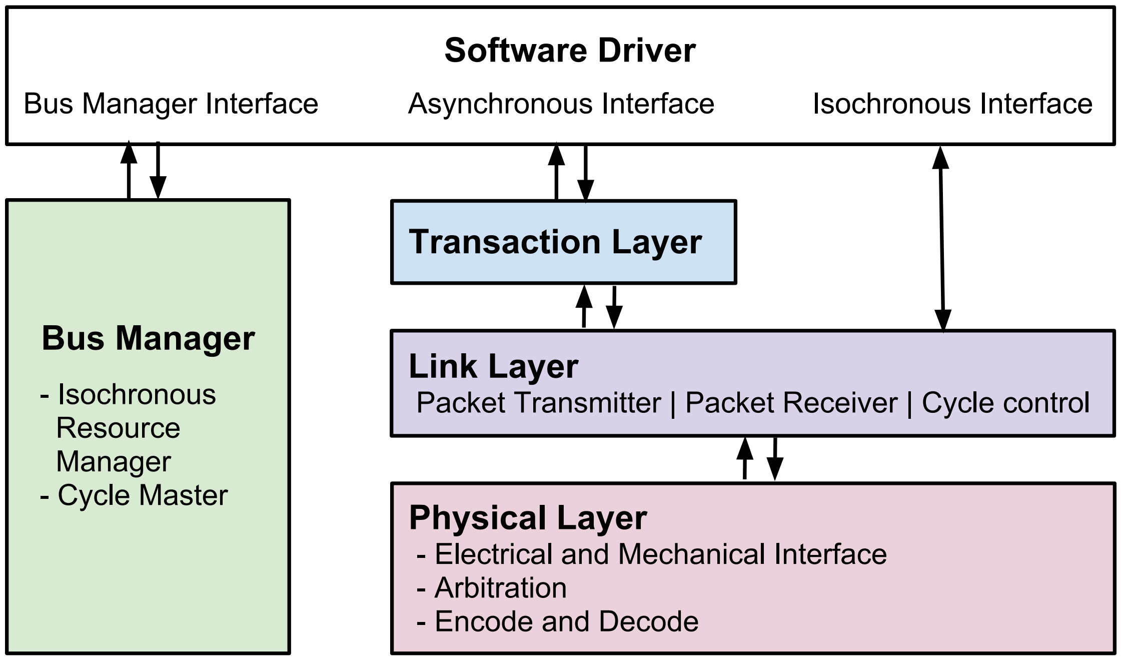


**Figure 2.8:** IEEE-1394 4 layer protocol architecture

In prior work, Thienphrapa [85, 86, 87] developed firmware to control a snake robot using asynchronous transmissions, initiated by the PC, to fetch and send data from and to the FPGA boards. Furthermore, for efficiency considerations, asynchronous transactions are implemented as concatenated transactions, where the acknowledgment packet and response packet (if a read transaction) are sent back to the requesting node without releasing the bus. Compared to split transactions, this eliminates the need for the responding node to wait for the subaction gap (at least 10 $\mu s$) and negotiate for bus access. This design has also been used for the da Vinci Research Kit [17, 48]. The following section presents experimental data to assess the performance of the FireWire transaction types, which guides the design of a protocol to achieve higher control performance.

### 2.4.2 FireWire Transactions Timing Performance

### 2.4.3 System Performance

This subsection presents the performance measurement of concatenated asynchronous read/write transactions, analyzes I/O versus computation ratio in a servo control loop, and reveals the bottleneck of achieving better control timing performance. All the data is collected on a Linux PC (FireWire chip *Ricoh R5C832*) with a *3.2.0-49-generic* kernel, *Juju* FireWire driver stack, and *libraw1394* API library. Timing data is queried using the *gettimeofday* function.

Figure 2.9 and Figure 2.10 show the time required for asynchronous block read and write transactions initiated by the PC software, each based on 5,000 iterations. The read and write payload sizes are 68 and 16 bytes, respectively, which match the payloads used for the FPGA-QLA board set. Mean read and write times are 31.99 and 33.74 $\mu s$, with standard deviations of 12.02 and 8.56 $\mu s$, respectively. Thanks to the concatenated implementation, the measured data is only half the value (around 60 $\mu s$, depending on the kernel) reported in [80].

The most straightforward protocol is to perform one asynchronous read (to obtain feedback data) and one asynchronous write (to send control output) to each FPGA board in each servo control loop. In this case, the total mean I/O time for a robot system with $N_{boards}$ FPGA boards would be:

$$T_{I/O} = (T_r + T_w) \times N_{boards}, \tag{2.1}$$

where $T_r$ and $T_w$ are the mean asynchronous read and write times, respectively. For a da Vinci Research Kit with eight FPGA boards, the computed I/O time cost is $(32.22 + 34.13) \times 8 = 531.12\ \mu s$. This number is consistent with data collected experimentally, which has a mean time of 495.98 $\mu s$ and standard deviation of 75.45 $\mu s$. Because servo loop computation time $T_C$ is very low (less than 40 $\mu s$ for an 8-board system), I/O time often takes over 90% of the minimum control period ($T_c + T_{I/O}$) and more than 50% of a 1 kHz control loop. This means that I/O performance is the bottleneck and would make it difficult to: (1) control a more advanced system (e.g., a system with 16 FPGA boards) with a 1 kHz servo loop, or (2) control an 8 board da Vinci system at frequencies greater than about 1.8 kHz.

### 2.4.4 Analysis

Figure 2.11 shows each step in an asynchronous read transaction, starting from the call to the *libraw1394* API function *raw1394_read* to the return of this function call. The average 32 $\mu s$ transaction time is comprised of two operating system calls, two data transmission times, and data processing time on the FPGA. We are able to measure the data transmission and data processing time in the FPGA, which is less

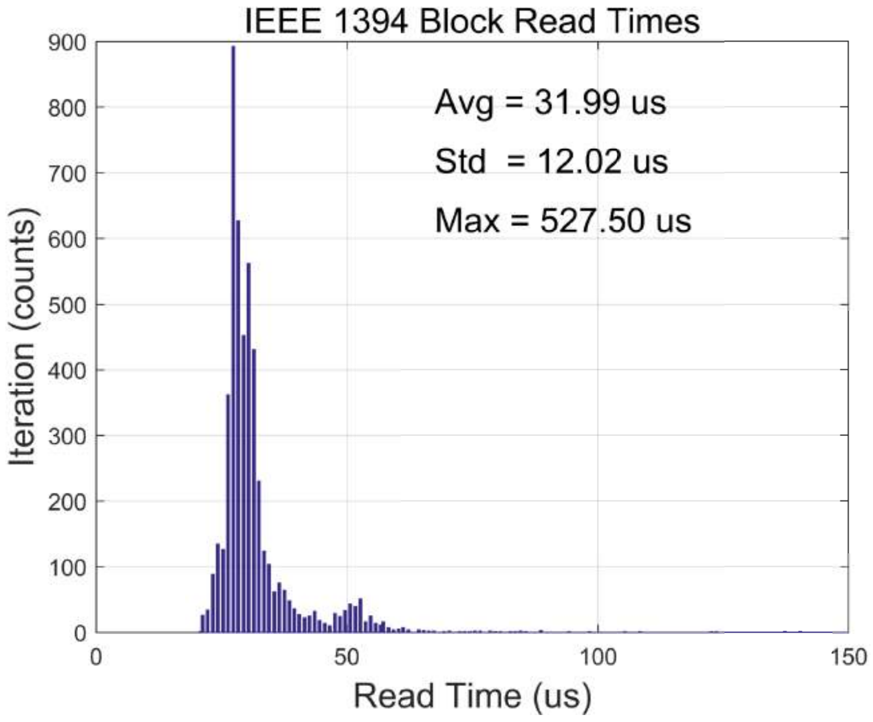


**Figure 2.9:** Asynchronous Block Read (400 Mbps)

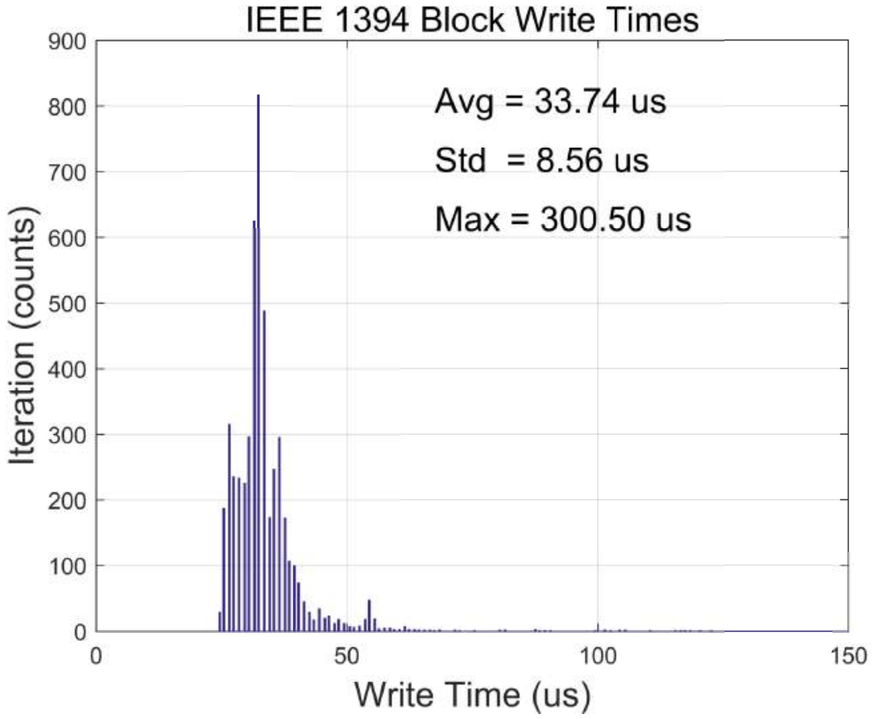


**Figure 2.10:** Asynchronous Block Write (400 Mbps)

than 5 $\mu s$. This implies that the latency is mainly due to software overhead in the operating system. An obvious inference is that in order to improve the I/O performance, the best approach is to reduce the total number of transactions initiated by the control PC. This insight guides us to use an asynchronous broadcast transaction-based solution, where we compensate for the lack of acknowledgment packets by embedding an acknowledgment (actually, a sequence number) in the packets sent from the FPGAs to the PC.

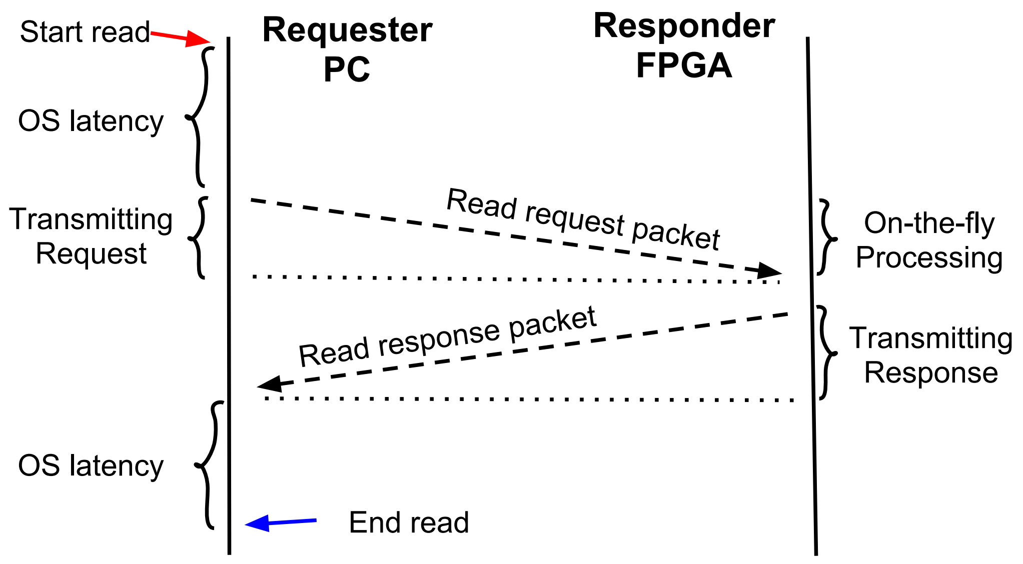


**Figure 2.11:** IEEE-1394 asynchronous block read includes two operating system (OS) calls, data transmission time, and data processing time.

## 2.5 Broadcast Communication Protocol

This section presents one contribution of this thesis, which is the newly designed high-performance communication model, including several optimizations to further improve performance, and discusses system characteristics, including protocol determinism, system integrity and backward compatibility.

### 2.5.1 Transmission model

As shown in Figure 2.12, a servo control cycle starts with an asynchronous broadcast write packet from the PC, serving as query (or sync) command to all FPGA boards. The payload of this packet contains two pieces of information: a sequence number that increments every control cycle and a board exist mask that indicates which boards are under control. The sequence number is used to ensure data integrity and is discussed further in Section 2.5.3.2. The board exist mask is constructed in the initialization phase. The Nth bit of the mask is set if the board with board ID N exists and the user wants to control this board. After sending this packet, the PC software sleeps for 5 $\times N_{boards}$ $\mu s$, where $N_{boards}$ is the total number of boards under control. Upon receipt of this packet, each FPGA board uses the board exist mask to count the number of boards ($N_{wait}$) that have a smaller board ID and are under control, waits for $5 \times N_{wait}$ $\mu s$, then transmits its status data using an asynchronous broadcast block write packet. This is a Time Division Multiple Access (TDMA) method, similar to the isochronous transfers already present in the IEEE-1394 specification, but scheduled with respect to the query command, which can have an arbitrarily specified frequency. All broadcast packets are received and cached by every FPGA node, so that all nodes maintain a copy of the entire robot status feedback. Upon awakening, the PC sends one asynchronous block read request to any node to fetch this information. This node can be called a *hub node*, though it is important to note that any FPGA node can serve this function. The PC software then performs the control computations and broadcasts new command data for all FPGA boards. This completes a servo control cycle. The key ideas behind this design are to reduce operating system overhead by cutting the total number of transactions initiated from the control PC, and to use broadcast packets to minimize the number of data packets on the bus. In fact, the number of PC-initiated transactions (3 transactions) is now independent of the number of FPGA boards (nodes) on the bus. The I/O time for the broadcast protocol is:

$$T_{I/O\_bc} = T_Q + 5\mu s \times N_{boards} + T_R + T_W, \tag{2.2}$$

where $T_Q$, $T_R$ and $T_W$ are the time for query, block read and command write transactions, respectively.

In theory, the PC could serve as the hub node and receive all status broadcast packets directly, but we have found that this is not a reliable solution. In our experiments, we detected a 2% packet loss when attempting to use the PC as a hub node. We hypothesize that this is due to the use of a software driver to handle asynchronous requests, which is inherently slower than a hardware-based (FPGA) solution. We therefore disabled receipt of the broadcast packets by the PC and introduced the hub node concept to solve this problem. By design, all FPGA boards are hub capable and the PC can read complete status feedback from any FPGA board on the bus. We note, however, that a real-time kernel, such as RTAI [60], with a real-time FireWire driver [100] could be another solution to prevent dropped packets.

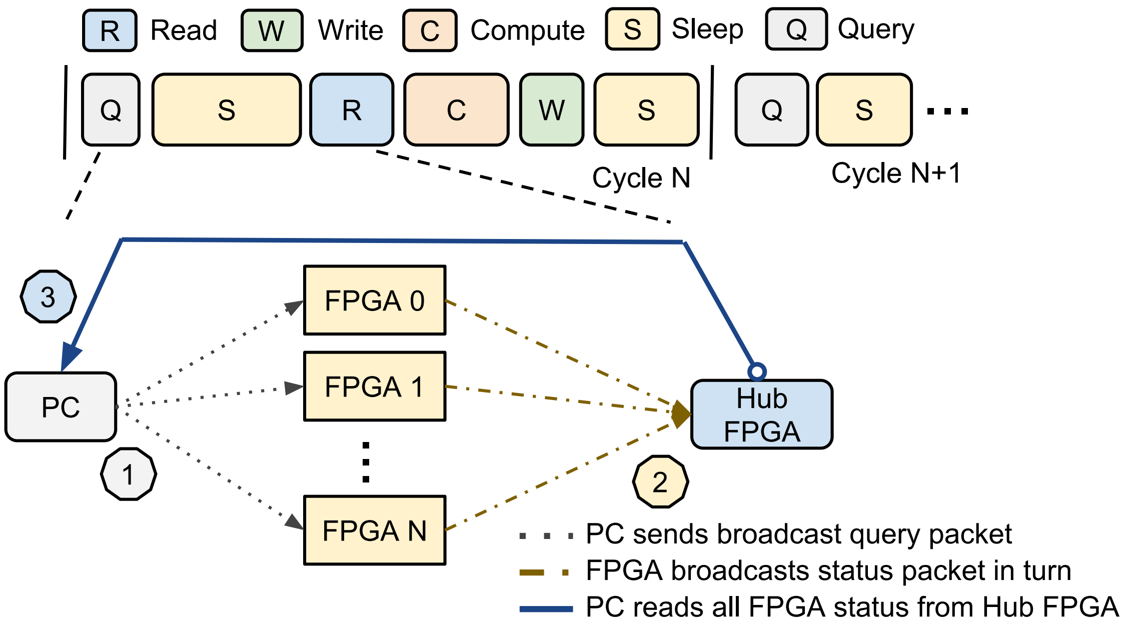

R Read
W Write
C Compute
S Sleep
Q Query
Q
S
R
C
W
S
Q
S
...
Cycle N
Cycle N+1
3
PC
FPGA 0
FPGA 1
FPGA N
Hub FPGA
1
2
PC sends broadcast query packet
FPGA broadcasts status packet in turn
PC reads all FPGA status from Hub FPGA

**Figure 2.12:** Asynchronous broadcast based communication model. Hub FPGA is not a separate FPGA, but is any one of the N FPGA boards.

### 2.5.2 Bus optimizations

While the above protocol greatly improves the performance of the system, the design incorporates several other optimizations, as detailed in this section. These particular optimizations are feasible in a closed system (e.g., where there are no other nodes on the FireWire bus) and could be omitted if necessary.

**Bus arbitration acceleration:** Whenever the link layer controller wants to transmit data to the FireWire bus, it sends an arbitration request to the physical layer chip and the physical layer will in turn arbitrate for bus ownership. In the FireWire specification, the link layer controller can only issue *priority* or *fair* requests for an asynchronous subaction. For these two types of requests, the physical layer chip starts bus arbitration after it detects a subaction gap, which nominally is 10 $\mu s$ [2]. This subaction gap time limits the overall performance. But, because the link layer is implemented in an FPGA, we are able to improve performance by issuing an *isochronous* bus request to the physical layer chip, even though we intend to send an *asynchronous* packet. In this case, the physical layer only waits for a 0.04 $\mu s$ isochronous gap before starting to arbitrate for the bus. In a standard FireWire system, this is possible because an isochronous bus request is only issued when the bus is performing isochronous transactions. In our design, it works because the time each node starts transmitting is deterministic (e.g., based on the TDMA method described above). This mechanism accelerates the bus arbitration process, improves bus bandwidth usage, and breaks the limitation of using regular asynchronous transactions. But, it assumes that we have complete control over the FireWire bus and can prevent an "outside" node (e.g., a FireWire camera or hard drive) from interfering with this protocol.

**Disable cycle start packet:** The cycle start packet is transmitted from the cycle master node (i.e., PC) on the bus at 8 kHz to synchronize isochronous data transfers. Because we do not use isochronous transactions and, more importantly, to avoid interfering with the broadcast write packets, the cycle master node capability is disabled and no cycle start packet is issued on the bus. It is also reported [89] that this optimization can increase asynchronous transaction performance by 5%.

**Full speed broadcast packets:** In the standard Linux kernel, the Juju FireWire driver sets the asynchronous broadcast speed to 100 Mbps, as it is used during the self-identification process and needs to support slower FireWire devices running at 100 Mbps. Given that all the boards are 400 Mbps capable, the broadcast speed can be changed to 400 Mbps to shorten the data transmission time from the PC and yield about a 4 $\mu s$ performance gain, at the cost of having to modify and recompile the FireWire driver source code.

### 2.5.3 System Characteristics

Besides high performance, our design has other system characteristics that favor a network-connected centralized processing and distributed I/O control architecture.

#### 2.5.3.1 Determinism

In a networked control architecture, determinism is beneficial and sometimes even required. This means that given a certain bus state, the next bus state is completely determined. This feature is extremely important when doing control at an extremely high frequency, such as 5 kHz. In our design, the determinism is guaranteed by bringing optimizations on top of the IEEE-1394a specification and by not implementing certain functionality in the FPGA FireWire module. The determinism of the system includes using a fixed root node (the PC), data transmission synchronization via a broadcast write packet from the control PC, and pre-configured bandwidth and offset.

By not implementing the bus manager layer on the FPGA nodes and not allowing other types of FireWire nodes on the bus, we can be assured that the control PC is the only node that can be bus manager, isochronous resource manager, and cycle master and is therefore forced (by the IEEE-1394 specification) to be the root node on the bus. This determinism also simplifies the procedure to disable cycle start packets. Because the FPGA nodes do not initiate asynchronous transactions (the broadcast asynchronous write packet is considered a “response” to the packet from PC), the software running on the PC has complete control over what data, at what time, is on the FireWire bus.

#### 2.5.3.2 Error tolerance

Data integrity is crucial in a robot control application. This is especially true for a medical robot that is designed to operate on patients. This is also the reason we favored regular asynchronous read and write over fast isochronous transactions in our previous design. For the same reason, we include three mechanisms to ensure data integrity, even when using broadcast packets for which there is no acknowledgment packet. The basic feature, a Cyclic Redundancy Check (CRC), is compulsory as it is specified in the IEEE-1394 standard. This provides a basic error detection mechanism. A more important feature is to include data integrity information and a sequence number (16 bits) from the PC write packet in the "response" broadcast packet from each FPGA. This feature is a remedy for the lack of an acknowledgment packet for asynchronous broadcast write packets. In a situation where the packet from the PC is corrupted or the data is incorrect, the sequence number in the FPGA packet is set to 0xFFFF; otherwise the received sequence number is returned. If the PC software receives a response with an incorrect sequence number (including 0xFFFF), it triggers a software error handling mechanism. Finally, the FPGA firmware includes a watchdog that needs to be refreshed by an asynchronous broadcast write packet from the control PC. This guarantees that in extreme cases (e.g., a software crash on the PC), the FPGA board will disable the amplifiers and ensure that there is no power to the robot system.

#### 2.5.3.3 Backward compatibility

The new design greatly improves communication performance between the control PC and FPGA and retains the support for asynchronous read/write transactions, thereby remaining backward compatible.

### 2.5.4 Experiments

This section experimentally examines the performance of the broadcast communication protocol using both the FPGA board and PC software. The measurement data is compared to the prior asynchronous protocol described in Section 2.4.3.

#### 2.5.4.1 FPGA hardware-based measurement

With a soft JTAG tool, we captured the data transmission on the FireWire bus in one complete servo cycle, as shown in Figure 2.13. The blue section of the data bus indicates that its value is changing and the LLC is either receiving (does not mean the data is targeted at the FireWire node) or transmitting data from or to the PHY chip. The counts at the top show the number of time cycles (clock is 49.125 MHz, $1\mu s = 49.125$ cycles). The cycle starts with a broadcast quadlet (four-byte) packet with less than 10 cycles. The total time for 8 FPGA boards to finish data transmission is around 2,000 cycles (40.71 $\mu s$), with each board taking 250 cycles on average (5.1 $\mu s$). After these transactions, an asynchronous read request, indicating the start of the third phase, has triggered the asynchronous block read response packet from the hub FPGA node. This asynchronous read, including the final ACK packet from the addressed node to the control PC, takes 20 $\mu s$. However, this number does not include operating system latency before the asynchronous read request is sent out and the latency after the data packet has physically arrived at the PC hardware. The time between the asynchronous block read (Async Hub packet) and the broadcast block write (PC Command packet) is PC computation time. Finally, the broadcast command packet from the PC transmitting at 400 Mbps takes 160 cycles (3.26 $\mu s$).

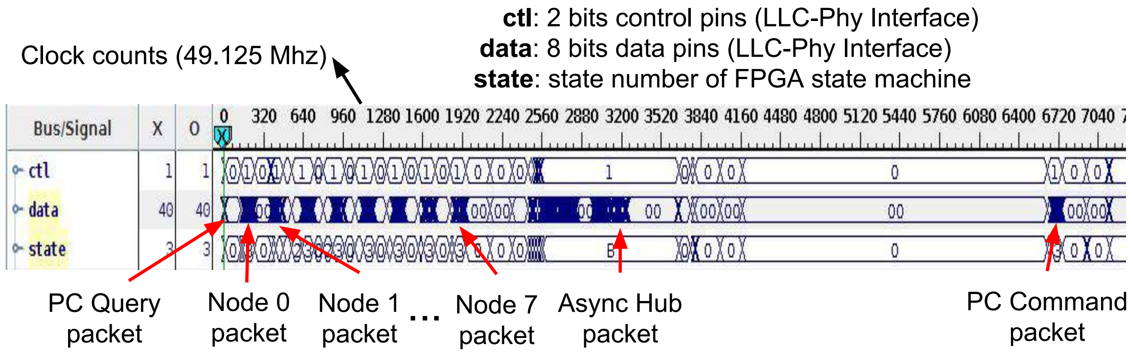

**ctl**: 2 bits control pins (LLC-Phy Interface)
**data**: 8 bits data pins (LLC-Phy Interface)
**state**: state number of FPGA state machine
Clock counts (49.125 Mhz)
Bus/Signal
ctl
data
state
PC Query packet
Node 0 packet
Node 1 packet
...
Node 7 packet
Async Hub packet
PC Command packet

**Figure 2.13:** Waveform of control, data and state bus within one I/O cycle

#### 2.5.4.2 Model parameter estimation with PC software-based measurement

While the measurement on the FPGA board is more accurate, it does not include latencies introduced by the PC software, such as the operating system scheduling delay. Thus, timing data measured from the PC is presented here. In the broadcast protocol, 1) an asynchronous broadcast write, 2) an asynchronous read and 3) an asynchronous broadcast write are involved. A detailed FireWire asynchronous timing model is presented, analyzed and simplified. Timing data of these three transactions are collected and modeled.

Timing models of these three types are then substituted into Equation (2.2) to obtain a timing model for the dVRK system. The model is then verified with timing data collected on hardware with different numbers of boards. We also compare the broadcast protocol performance with the asynchronous protocol. The measurement of asynchronous block read and write times was initially reported by Thienphrapa [86] but has been repeated here because the hardware and software are different.

The total timing of an asynchronous concatenated transaction call comprises of a list of the basic elements as shown in Figure 2.14. These elements include the software overhead ($T_{sw1}$) of sending a request from software to the FireWire card, the subaction gap ($T_{gap}$), the bus arbitration time ($T_{arb}$), the asynchronous request packet transmission time ($T_{request}$), the acknowledge gap ($T_{ack\_gap}$), the acknowledge packet transmission time ($T_{ack}$), the asynchronous read response packet time ($T_{response}$) (if the request is of type quadlet read or block read) and the software overhead ($T_{sw2}$) from the FireWire device to the calling software. Equation 2.3 summarizes the total timing cost.

$$T_{async} = T_{sw1} + T_{gap} + T_{arb} + T_{request} + T_{ack\_gap} + T_{ack} + T_{response} + T_{sw2} \tag{2.3}$$

Although an asynchronous concatenated transaction has many elements, it can be simplified into a linear model in our application. The following analysis is for an asynchronous block write transaction, but could be extended to other types of transactions. The PC program initiates a blocking asynchronous block write by calling *raw1394_write*. The operating system first constructs a FireWire packet with the specified payload and then transfers the packet to the FireWire card's transaction layer interface via its PCI/PCIe bus. This period of time is captured in $T_{sw1} = T_{sw1\_os} + 8 \times SZ_{request}/BW_{PCI}$, where $T_{sw1\_os}$ is the operating system overhead, $SZ_{request}$ is the size of the block write request packet in bytes and $BW_{PCI}$ is the bandwith of the PC's internal bus. For a block write request, it includes 6 quadlets (24 bytes) overhead (header and CRC data) and the write data payload $SZ_{data}$, i.e. $SZ_{request} = 24 + SZ_{payload}$ bytes. Then, the FireWire card starts the FireWire bus arbitration process by waiting for a subaction gap. $T_{gap}$ represents the subaction gap between different asynchronous transactions (See Figure 2.7), which is around 10 $\mu s$. $T_{arb}$ is the time for the FireWire bus arbitration process. In the arbitration process, the node closest to the root node wins the arbitration. Given that the control PC is guaranteed to be the root node and all FireWire transactions are initiated from the control PC, the control PC is guaranteed to win the arbitration process within a small fixed period of time. $T_{request}$ is the time cost of transmitting the asynchronous block write packet on the FireWire bus and is given by $T_{request} = 8 * SZ_{request}/BW_{FireWire}$, where $BW_{FireWire}$ is the FireWire transmission bandwidth (400 Mbps in our case). $T_{ack\_gap}$ is a small fixed arbitration gap between 0.04 and 0.05 $\mu s$ [2] as the recipient of the request has guaranteed access to the bus and uses an immediate arbitration service. The recipient then sends out an acknowledgment packet with size of 8 bits, which takes around 0.02 $\mu s$. $T_{sw2}$ models the time from when the FireWire card receives this acknowledgment packet to when the operating system returns from the *raw1394_write* function. As shown in the above analysis, all the items in equation 2.3 are either a constant time or a linear function of the payload size $SZ_{payload}$; thus, the sum of all the elements that contribute to $T_{async}$ can be modeled with a linear function. This analysis is verified with experimental data.

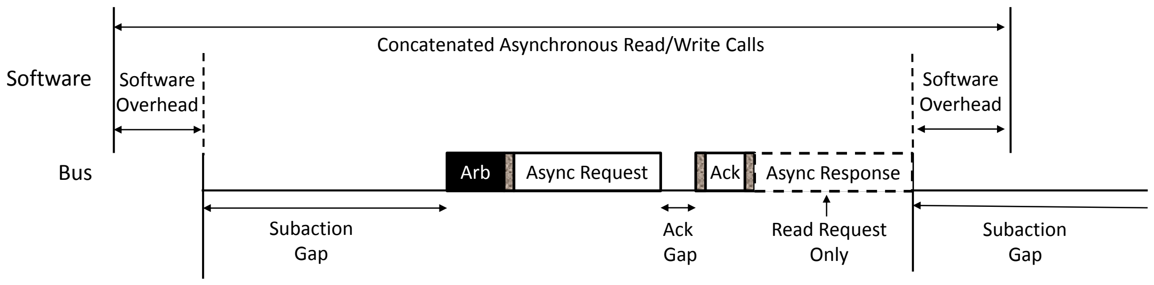


**Figure 2.14:** FireWire asynchronous timing

Figure 2.15 shows the raw transaction sampling time of asynchronous read and write over the full range of data block sizes allowed at 400 Mbps speed. The block sizes are measured in quadlets, as the standard requires the packet size to be aligned at 32 bits. For each data size, 10,000 timing data were collected and the mean data is reported. The data plots verified that asynchronous block read and write timing can be modeled using a linear model with bandwidth and base latency as parameters.

$$T = T_{latency} + \frac{8 * SZ_{data}}{BW}, \tag{2.4}$$

where $T_{latency}$ is the latency introduced by software, FireWire bus arbitration as well as packet overhead, $SZ_{data}$ is the payload size in bytes, and $BW$ is the bandwidth, in bits/sec (bps), for transmitting data in the PC internal bus and on the FireWire bus. The PC internal bus is significantly faster than the FireWire bus, so the limiting factor is the FireWire bandwidth of 400 Mbps.

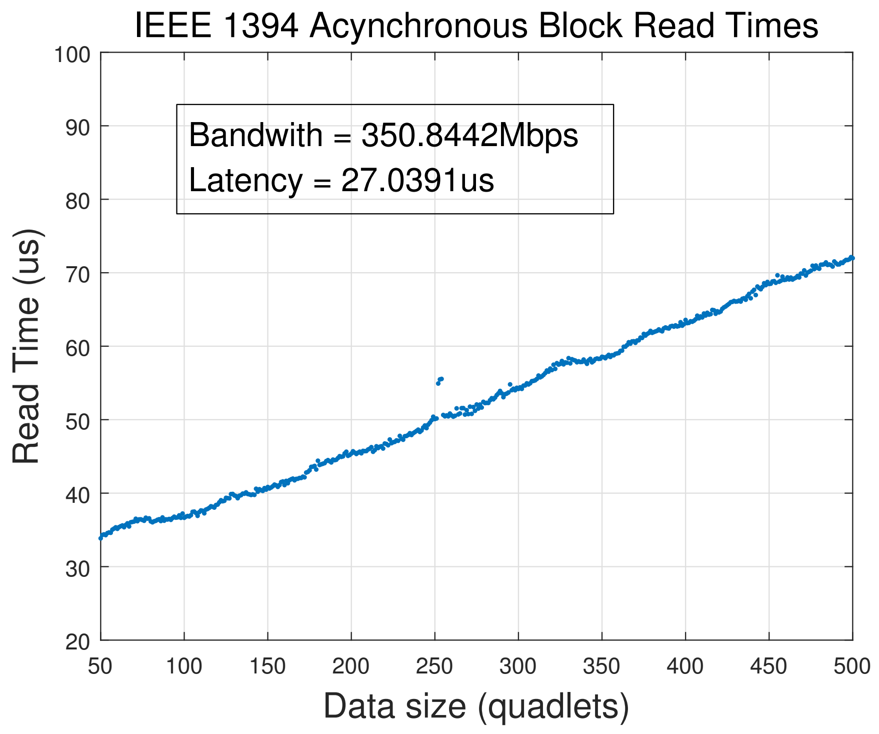
After a least square fitting process, the plots show base latencies ($T_{latency}$) of about 27.04 $us$ for reads and 28.22 $us$ for writes. Further, the average speeds of reads and writes (bandwidth) are 350.84 Mbps and 265.12 Mbps. Neither value reaches the nominal 400 Mbps rate. Using Equation (2.4), the time of asynchronous block read and write are:

$$T_{bread} = 27.04us + 0.022802 * SZ_{data} \tag{2.5}$$

$$T_{bwrite} = 28.22 + 0.030175 * SZ_{data} \tag{2.6}$$

The asynchronous broadcast write request is sent out to the FireWire bus by calling the *raw1394_start_write* function. Figure 2.16 shows that the average time cost is 2.51 $\mu s$ with a standard deviation of 2.71 $\mu s$. Note that this value is only the time cost for calling the function and does not include, or not fully include, the transmission time. Compared with a regular asynchronous block write with a 16 byte payload, the time cost is 90% less, which is not surprising since the broadcast does not require an acknowledgment packet and thus saves the time of waiting for a response.

In subsection 2.5.1, we presented the timing model of broadcast protocol in Equation 2.2: $T_{I/O\_bc} = T_Q + 5\mu s \times N_{boards} + T_R + T_W$. The items $T_Q$, $T_R$ and $T_W$ are a



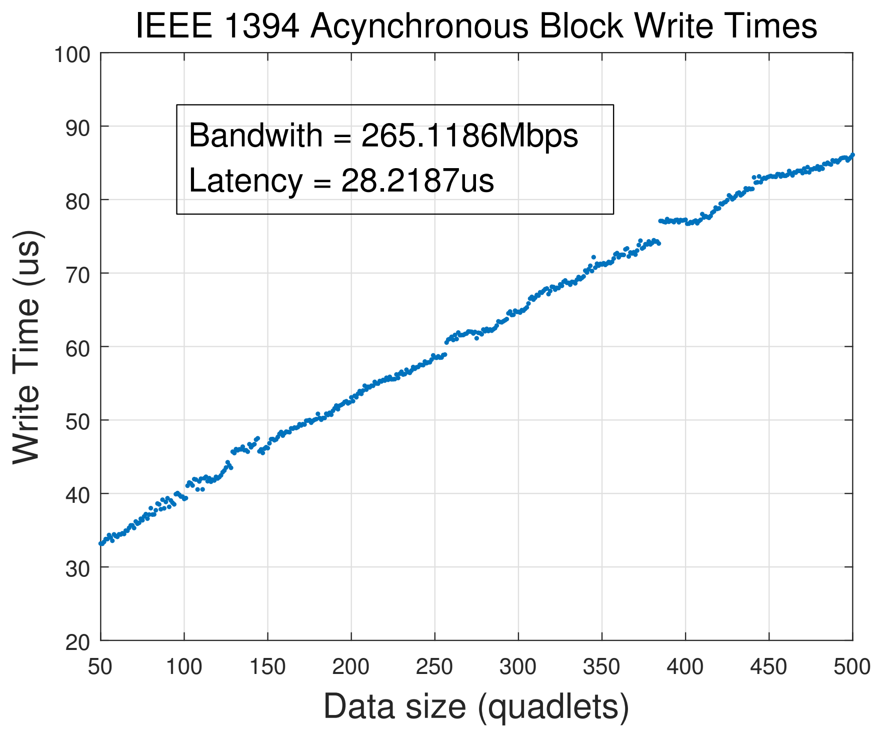


**Figure 2.15:** Asynchronous block read and write times (400 Mbps)

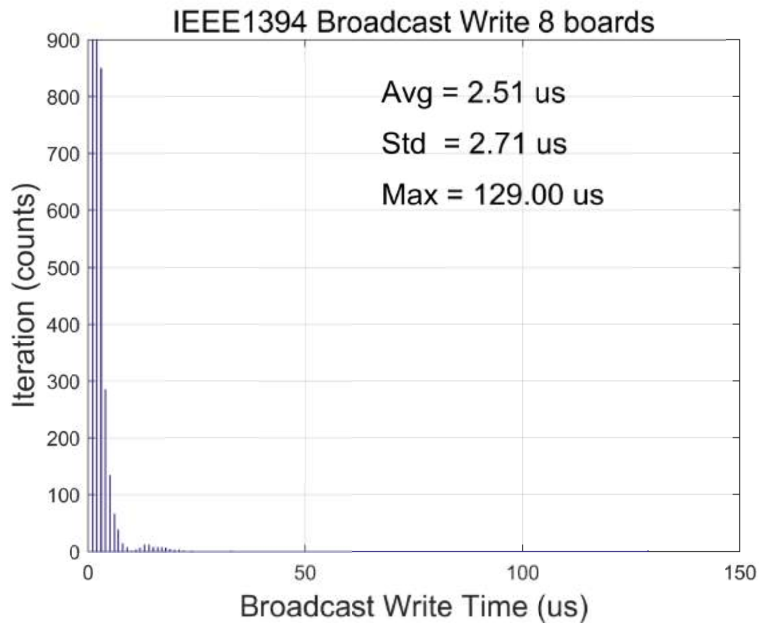


**Figure 2.16:** Asynchronous Broadcast Block Write

broadcast write (query) transaction, an asynchronous block read transaction and an asynchronous block write transaction, respectively. By substituting Equation (2.5), (2.6) and broadcast write mean time into Equation (2.2), the timing model equation can be rewritten as:

$$T_{I/O\_bc} = 2.51+5\times N+(27.04+0.022802*SZ_r*N)+(28.22+0.030175*SZ_w*N) \quad (2.7)$$

where $N$ is the number of boards, $SZ_r$ and $SZ_w$ are the sizes, in bytes, of the status packet and command packet for a single board, respectively. For the da Vinci Research Kit, $SZ_r$ and $SZ_w$ are 68 bytes and 16 bytes, respectively. The final timing model is a linear function:

$$T_{I/O\_bc} = 57.77\mu s + 7.0333\mu s \times N \quad (2.8)$$

To verify the accuracy of this model, we experimentally collected timing data on systems with 1 to 10 boards. Similar to previous experiments, a total of 10,000 timing data samples are collected for each setup and the mean timing values are plotted in Figure 2.17. The timing data collected from the hardware is higher than the model predicted data, with a mean difference of 20 $\mu s$. One of the reasons is that we added 10 $\mu s$ to the wait after the read request packet to ensure that all boards have finished transmitting. Another possible source of the difference is measurement error. The standard deviation of the timing data is relatively large, especially for smaller numbers of boards.

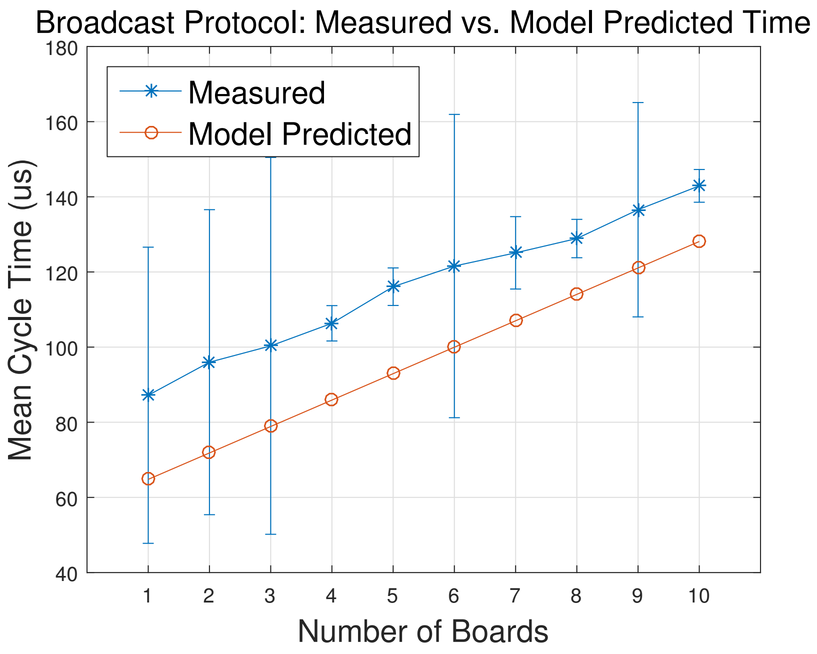


**Figure 2.17:** Broadcast protocol mean cycle time on dVRK, predicted by model (red) and measured (blue, with error bars showing standard deviation)

We also experimentally compared the proposed broadcast protocol to the prior asynchronous protocol. As shown in Figure 2.18, the broadcast protocol shows a huge performance increase, especially for a system with many nodes. Using the broadcast protocol, a da Vinci system with 8 boards (4 manipulator arms) could conceivably run at 6 kHz.

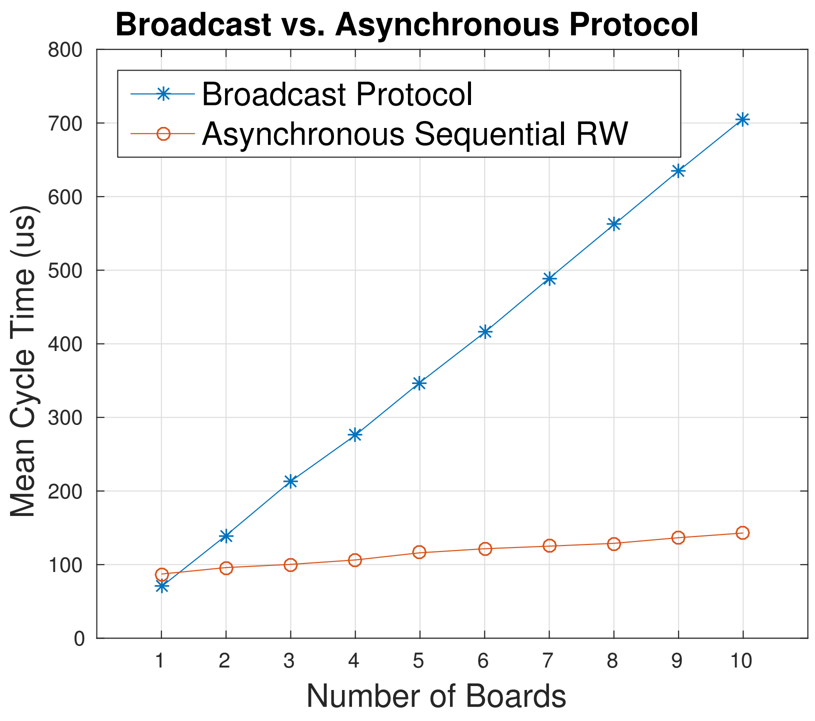


**Figure 2.18:** Mean cycle time comparison between Broadcast and Asynchronous protocols on dVRK

### 2.5.5 Discussion

Our analysis of the original control system timing performance revealed that the latency due to PC operating system overhead was the primary cause of the I/O performance bottleneck, which led us to a series of bus optimizations and a new communication protocol. This new protocol reduces the number of PC-initiated transactions to three by using broadcast packets and enabling all FPGA controller boards to broadcast and cache status packets. Performance-wise, the new design shows good scalability and cuts the average I/O cycle time of a full dVRK system to under 200 $\mu s$. One limitation of the current implementation of the protocol is that it is application-specific. For example, the FireWire status packet size and the wait time per node are fixed, but they can be parameterized so that the current protocol can be used for other robotics systems. The general design of the protocol can be applied to any fieldbus with broadcast and peer-to-peer communication capabilities. For other fieldbuses, the idea to minimize the number of PC transactions and fieldbus optimizations at the link layer can still be applied to achieve an efficient I/O timing performance.

## 2.6 Ethernet-to-FireWire Bridge for Real-time Control

This section presents one contribution of this thesis, which is a bridge design to enable real-time control over a conventional Ethernet interface, including a FPGA based bridge board design and a packet forwarding mechanism, and timing performance modeling and experimental evaluation. The general concept and design was developed by Zihan Chen with help from Dr. Peter Kazanzides. It was prototyped with help from visiting summer student Long Qian, from Tsinghua University, who wrote the initial FPGA bridge firmware and assisted with performance testing.

### 2.6.1 Introduction

Our choice of FireWire as the fieldbus for dVRK has proven to be successful. With the proposed broadcast, we achieved multi-kilohertz control on a full dVRK. However, FireWire today is less prevalent than in the past and even the real-time PC driver stack, RT-Firewire [101], is no longer maintained. Thus, our solution suffers from occasional timing outliers. Furthermore, while the new broadcast protocol can achieve up to 6 kHz control rates, we have found that it does not work reliably with some PC FireWire chipsets/drivers. Finally, FireWire interfaces are not as common as Ethernet on modern computers and laptops, and use of *libraw1394* primarily restricts the system to Linux and its real-time variants. Although a Windows version of libraw1394 has been reported [90], this would only be suitable for non-real-time applications. More importantly, some dVRK sites have invested in other real-time platforms, such as Matlab Simulink Real-Time (formerly called Matlab xPC), which supports Ethernet and EtherCAT but not FireWire. Thus, while one could invest in developing real-time FireWire drivers for the different platforms and require all control computers to have FireWire interfaces, it is more practical to leverage the existing hardware and software support for Ethernet as a real-time control fieldbus.

We considered two approaches to leverage Ethernet-based technology for our multi-node distributed control system: (1) replace the FireWire interfaces on each node with EtherCAT, or (2) introduce an Ethernet-to-FireWire bridge between the PC and FireWire subnetwork. The first approach has the potential advantage that the cables are conventional unshielded twisted pair (UTP) and can be high-flex, longer, and more easily routed inside robotic structures. Cabling is not an issue for the dVRK, however, and this approach would require a substantial retrofit of existing systems including modification of the FPGA board design to include two Ethernet ports and change of firmware to include an EtherCAT Slave Controller IP core from Beckhoff. More importantly, all existing FireWire-based FPGA boards would no longer work with this approach and would need to be replaced. So we adopted the second approach, which is illustrated in Figure 2.19. This section presents the Ethernet-to-FireWire bridge design and the results of experiments, including those with the actual dVRK hardware, to demonstrate that with the appropriate software, it provides hard-real-time performance for multi-kiloherz centralized control of a large number of distributed robot axes.

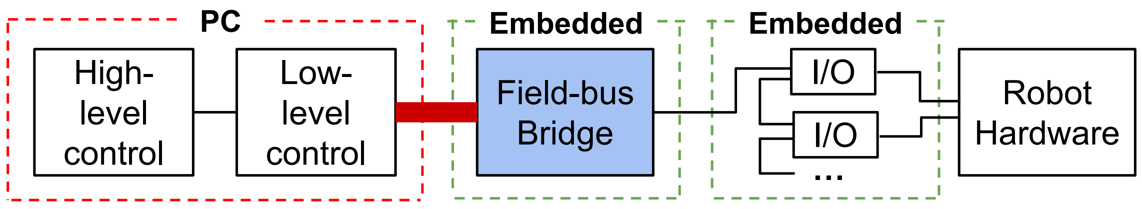


**Figure 2.19:** Control system architecture with a fieldbus-bridge

It is important to notice that although the bridge is designed for dVRK and realized as an Ethernet to FireWire bridge, the method itself can be generalized to convert other buses to a real-time capable fieldbus, for example USB to FireWire.

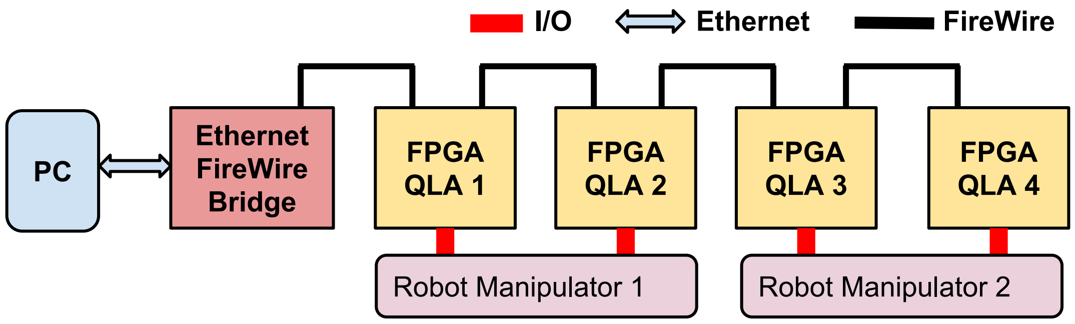


**Figure 2.20:** Hardware architecture with prototype Ethernet to FireWire bridge

### 2.6.2 Ethernet-to-FireWire Bridge Design

Compared with a fully FireWire-based system (Figure 2.6), Figure 2.20 introduces a new system design for dVRK with an embedded bridge between the PC and FireWire-based control network. All communications between the PC and slave nodes are done via the bridge node. While the bridge talks to the rest of control network using the FireWire-based broadcast protocol, the connection between the PC and bridge node is point-to-point, thereby eliminating the need for a complicated media access control protocol.

This section describes the bridge design, frame transmission protocol and status control process implemented on the firmware of the bridge node.

#### 2.6.2.1 Prototype Bridge Board Design

The prototype bridge consists of our custom FPGA board, which contains a Xilinx Spartan-6 XC6SLX45 FPGA and IEEE-1394a physical layer chip, coupled with an off-the-shelf Ethernet PHY and MAC controller board (KSZ8851-16mll-EVAL). The

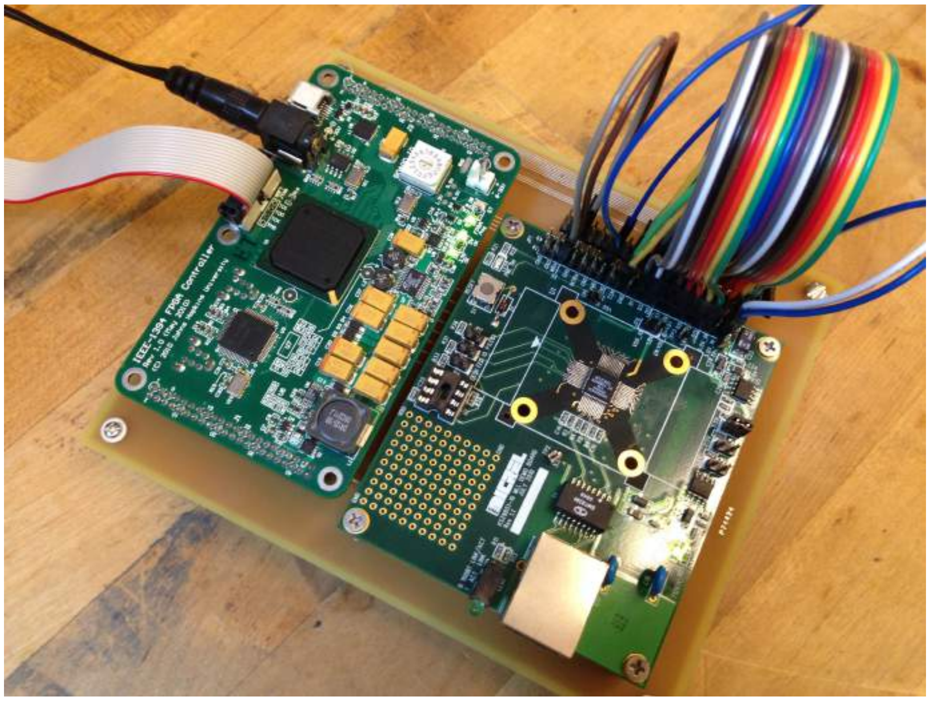


**Figure 2.21:** Prototype Ethernet-to-FireWire Bridge

Ethernet chip manufactured by Micrel is a single-port controller chip with a non-PCI Interface and is available in 8-bit and 16-bit bus designs. We utilized the 16-bit bus in our design for better efficiency in Ethernet data I/O. We designed a custom connector board to interface these two boards, as shown in Figure 2.21.

### 2.6.2.2 Frame Transmission Protocol

The main functionality of the bridge node is to convert an Ethernet packet from the PC into a FireWire packet, perform the FireWire transaction, and convert the FireWire response to an Ethernet packet for the PC. To simplify the development of the bridge node FPGA firmware and to maximize system efficiency, the FireWire packet construction and parsing is implemented in the PC software.

Frames transmitted from the PC to the bridge node include *quadlet read/write, block read/write*, the previously defined *broadcast-based write/query*, and *system synchronization*, each with an appropriate Ethernet header and checksum. The frame structure is presented in Figure 2.22. *Quadlet read/write* and *block read/write* are defined in the IEEE-1394 standard, enabling basic read and write transactions of variable length between two individual nodes within a FireWire network. As demonstrated previously, the *broadcast-based write/query* accelerates the control system by eliminating multiple requests to each separate node from the PC. The *system synchronization* frame is used to inform the bridge of the number of active nodes, $N$, in the FireWire network. The PC controller is authorized to add or remove an existing board to or from the list of current active boards. After a broadcast query is transmitted, the bridge will serve as a hub, collecting $N$ responses before sending the combined packet back to the PC.

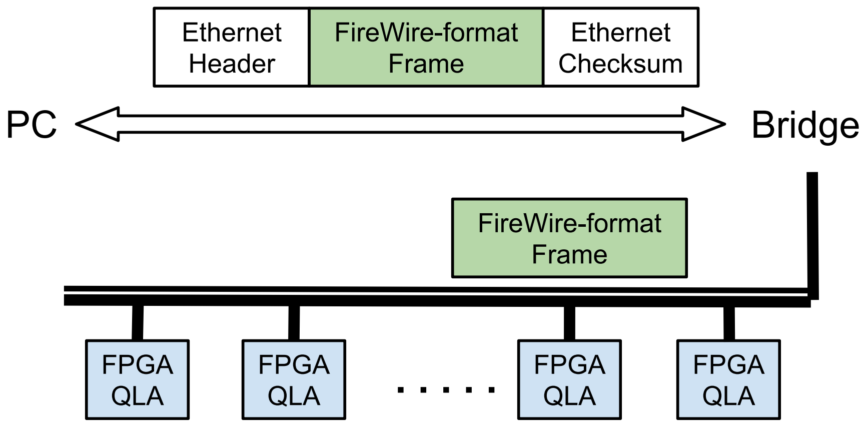


**Figure 2.22:** Ethernet Frame Structure

Upon receiving an Ethernet packet, a parallel validation checking process based on the frame format is activated. The validated frame is passed to the FireWire network with the Ethernet header and checksum removed. If a *system synchronization* packet is received, the local parameter $N$ is updated; it is not necessary to relay that frame to the FireWire subsystem.

In the reverse direction, *quadlet/block read response*, *write acknowledge frame*, and *broadcast query response* are transmitted through the FireWire field-bus. The first two frame types are defined in the IEEE-1394 standard as responses to *quadlet/block read/write* requests. When *quadlet/block read responses* are received by the bridge node, they are passed to the Ethernet network with a specific Ethernet header and a correct checksum. The *write acknowledge frame* triggers the switch of state in the bridge. The PC controller is not acknowledged because the overhead of transmitting an Ethernet frame is comparatively high. The *broadcast query response* is initiated by individual slave nodes in the distributed I/O subsystem, and provides the feedback information for the PC to perform closed-loop control. The bridge gathers $N$ feedback frames and then transmits them in one Ethernet packet with a predefined Ethernet header.

With this design, we successfully inherit the advantages of the FireWire-based approach and at the same time benefit from the ubiquity of Ethernet hardware and its well-maintained real-time driver stack for the PC. Robustness is also improved because the FireWire broadcast protocol no longer involves the FireWire chipset on the PC, which was problematic on some systems.

#### 2.6.2.3 Status Control

Featuring its parallel operation, hard real-time capability and plentiful I/O extensions, the FPGA is more suitable than the PC to implement the finite state machine of the control loop. Sequential state transitions are predefined in the firmware of the bridge, along with the signals triggering state changes. Though the PC is responsible for initiating read or write commands, its request does not act as an interrupt for the bridge node. Instead, commands are buffered until the bridge reaches the status of fetching a PC request. A timeout is set in order to avoid unnecessary waiting caused by errors. When the timeout requirement is met, the bridge switches back to the *Broadcast Read Request from PC* state, where it constantly polls for the trigger representing the arrival of a request initiated by the PC controller. In a complete control cycle, the finite state machine (FSM) is implemented as illustrated in Figure 2.23. Five triggers are utilized to initiate state transitions in the FSM architecture.

#### 2.6.2.4 Ethernet Software

The software of the bridge-based da Vinci Research Kit is arranged into several functional layers which remain unmodified compared to the FireWire-based dVRK: hardware interface, PID-based low-level control, high-level customized control, teleoperation and application layer [48].

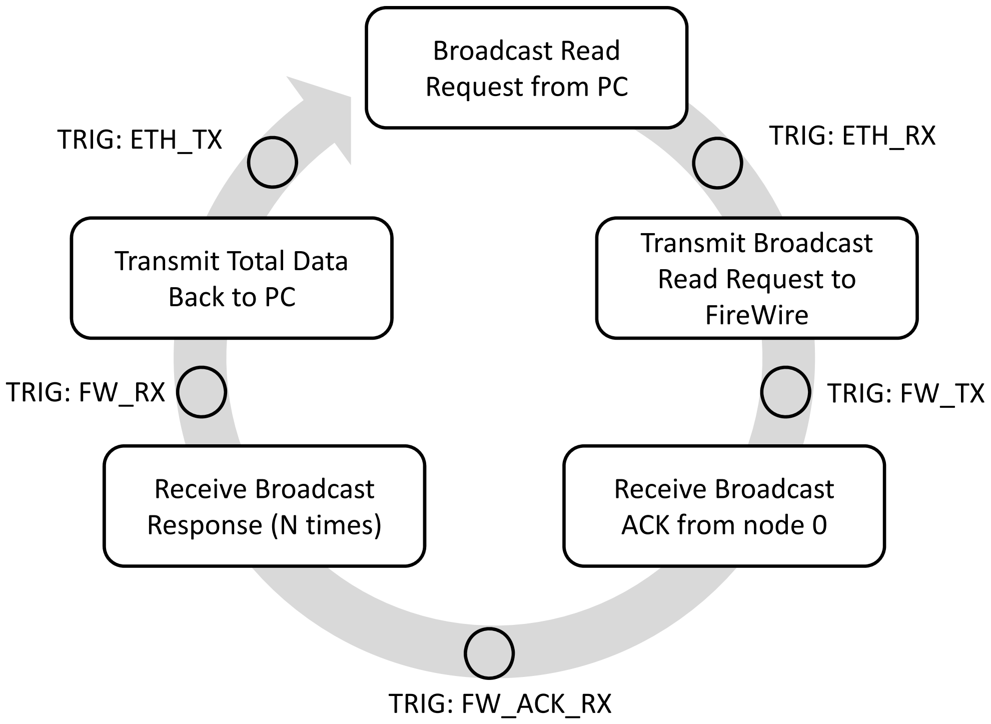

Broadcast Read Request from PC
TRIG: ETH_RX
Transmit Broadcast Read Request to FireWire
TRIG: FW_TX
Receive Broadcast ACK from node 0
TRIG: FW_ACK_RX
Receive Broadcast Response (N times)
TRIG: FW_RX
Transmit Total Data Back to PC
TRIG: ETH_TX

**Figure 2.23:** Finite State Machine for control loop, with num_node FireWire nodes.

The introduction of the Ethernet-to-FireWire bridge requires an Ethernet interface instead of FireWire in the hardware interface level; this Ethernet interface is provided by the C++ library *pcap*. An *Eth1394Port* is created to represent the node in the bridge-based design. An abstract base class *BasePort* is introduced so that both *Eth1394Port* and *FirewirePort* can inherit from it and provide the same functionality. Class *AmpIO*, which represents an FPGA/QLA node, is unchanged, thereby keeping the upper software layers intact.

The Ethernet interface library *pcap* is directly available for Linux and OS X. For Windows, a slightly modified library, *winpcap*, is utilized, which provides the same methods for Ethernet port operation. Portability between different operating systems

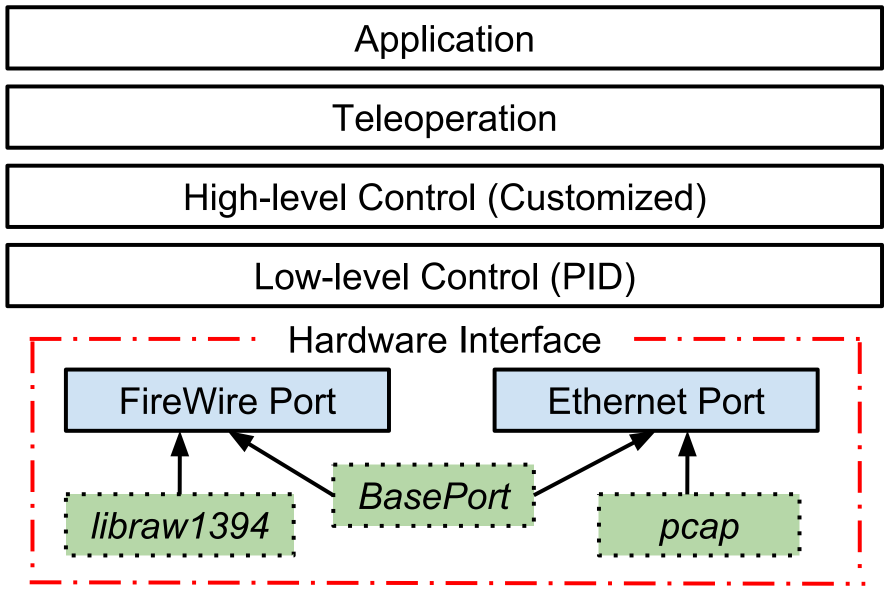


**Figure 2.24:** Software Architecture

is guaranteed by the cross-platform support of the *pcap* library, which is a significant improvement compared to the FireWire-based dVRK, which required the *libraw1394* library that is only readily available on Linux. To achieve best performance of the system, a real-time environment composed of a real-time operating system and real-time Ethernet driver is required. Our software can be quickly ported to such platforms with the ubiquitous support for the *pcap* library; for example, Xenomai with the RTnet driver or Matlab Simulink Real-Time.

### 2.6.3 Experiments

System performance experiments are conducted in the following three aspects. First, we measure the round-trip time of the standard FireWire protocol *quadlet* *read*, using a real-time operating system and real-time Ethernet driver. Timing characteristics of the bridge-based data transmission are compared with the FireWire-only transmission. Following that, control loop performance of both systems is tested and discussed. Furthermore, the cross-platform support for an Ethernet-based design is demonstrated.

#### 2.6.3.1 FireWire Transaction over Ethernet

As presented in the *System Overview*, the previous FireWire-based design achieves a good average timing performance, however, the system reliability suffers from lack of support for a real-time PC FireWire driver. The introduction of the Ethernet-to-FireWire bridge aims to improve the system performance by taking advantage of the prevalent real-time Ethernet driver. First, the round-trip time of the basic *quadlet read* transaction for both system configurations is measured by averaging over 5000 transactions, as shown in Figure 2.25. An Ethernet sniffer (tcpdump) based on the *pcap* library is used to capture the timestamp of Ethernet frames. A round-trip of a quadlet read transaction includes an Ethernet quadlet read request initiated by the PC controller and a corresponding response transmitted from the slave node. For both designs, the testing environment is set up on a real-time operating system (Xenomai 2.6.3 real-time framework based on Linux). The bridge design uses a real-time Ethernet driver (RTnet 0.9.12), while the FireWire-based design uses the standard (non-real-time) FireWire driver. Instructions for setting up the

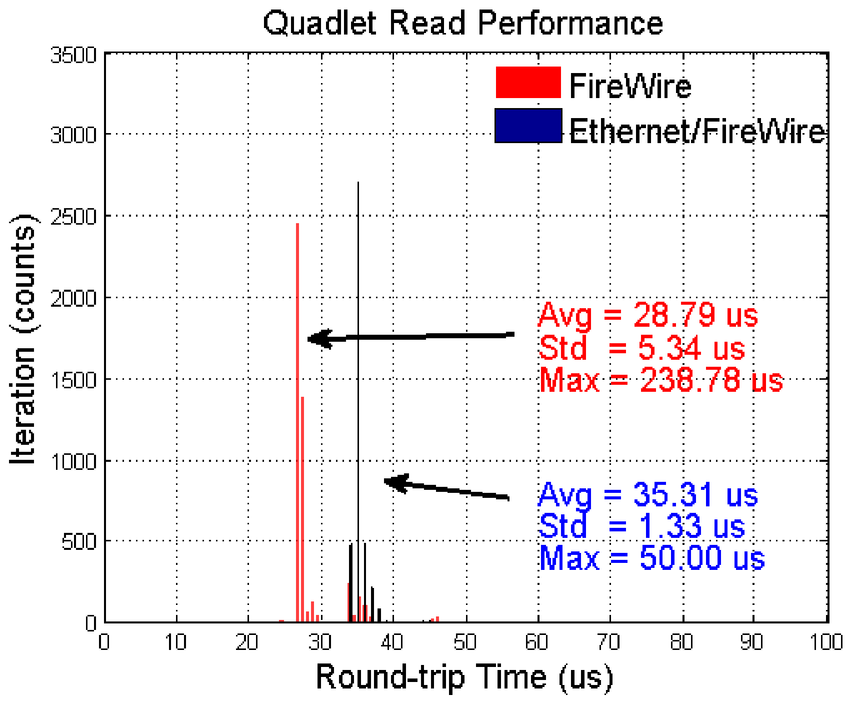


**Figure 2.25:** Quadlet read transaction times, tested on Xenomai real-time operating system. FireWire uses standard (non-real-time) FireWire driver, whereas Ethernet/-FireWire uses RTnet real-time driver.

Xenomai/RTnet system are provided in Appendix A.

The average execution times for the FireWire-only and Ethernet-to-FireWire Bridge designs are 28.79 $\mu s$ and 35.31 $\mu s$, respectively. The average time for the bridge-based design is longer due to the two additional Ethernet transmissions. But, the maximum time for the *quadlet read* transaction is significantly reduced from 238.78 $\mu s$ for the FireWire-based design to 50 $\mu s$ for the bridge-based design. This is primarily due to the use of the real-time Ethernet driver.

#### 2.6.3.2 System Performance

We measured the control loop performance of the bridge-based design and compared it with the FireWire-only design, as shown in Figure 2.26. The system performance includes the I/O time of Ethernet and FireWire in an 8-node system, which is typical for the control of the dVRK. Broadcast transfers are employed in both systems to maximize control efficiency. The test environment is the same as for the quadlet read, except that the FireWire-only design is tested with the generic Linux kernel rather than with Xenomai. In our experience, Xenomai and Linux-generic produce similar I/O times, since the primary cause of timing variations appears to be the non-real-time FireWire driver used in both cases.

The Ethernet I/O costs about 47.76 $\mu s$ in the system loop, which is higher than the Ethernet I/O time for a *quadlet read* transaction due to the larger payload. As expected, the bridge-based design has more consistent timing measurements than

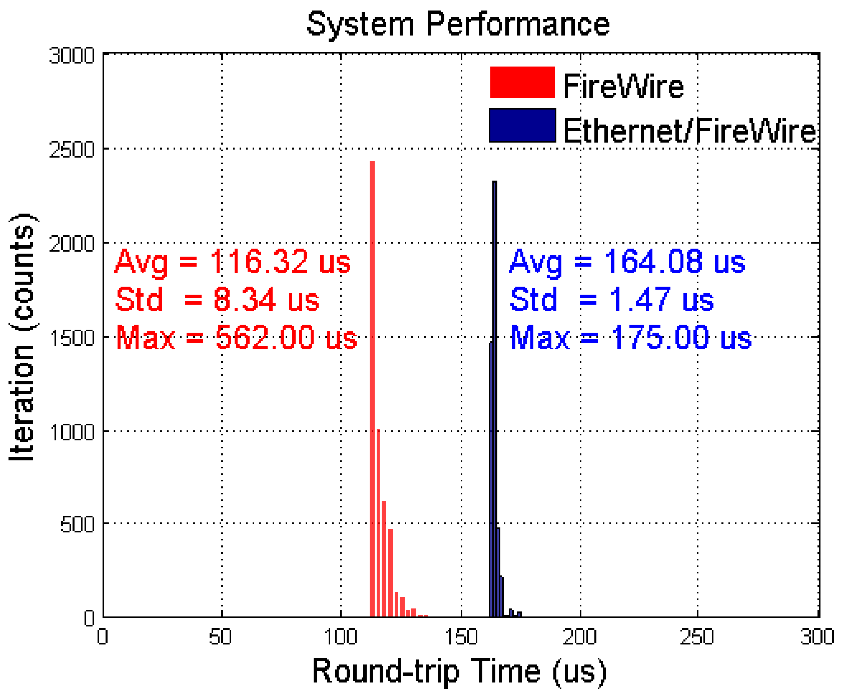


**Figure 2.26:** I/O Time of FireWire and Ethernet/FireWire bridge in an 8 FPGA-QLA board system (standard dVRK setup); FireWire tested on generic Linux, whereas Ethernet/FireWire tested on Xenomai with RTnet driver.

the FireWire-only design. The standard deviation of the bridge-based design is 1.47 $\mu s$, and the maximum time cost is 175.00 $\mu s$, which is less than one-third of the FireWire-only design. With PC computation time added, the complete control loop for the bridge-based design is less than 200 $\mu s$, which is sufficient for 5 $kHz$ control. Though the control frequency of the FireWire-based design is higher on average, the performance is not as deterministic due to the lack of a real-time FireWire driver.

The introduction of the Ethernet-to-FireWire bridge separates the FireWire subsytem, which can then be implemented entirely on the FPGA. This makes it easier to support our custom broadcast protocol and avoid problems that we faced with some PC FireWire chipsets and drivers. On the PC, this design benefits from the availability of a real-time Ethernet driver.

#### 2.6.3.3 Cross-platform Capability

The prevalence of Ethernet ports and software support (e.g., *pcap* library [13]) renders the bridge-based design as a cross-platform solution. This test measures the timing performance of *quadlet read* and *broadcast read* for an 8-node system using different operating systems, as shown in Figure 2.27. Xenomai is a real-time framework for Linux; real-time drivers such as RTnet are supported on the Xenomai platform. System performance is less satisfactory on non-real-time platforms such as Linux Generic and Apple OS X. For the Windows operating system, an echo test reveals that it takes approximately 2.34 $ms$ for two Windows controllers to communicate

| | | Xenomai | Linux | OS X |
|---|---|---|---|---|
| Quadlet Read ($\mu s$) | Avg | 35.3088 | 260.8400 | 306.7500 |
| | Max | 50.0 | 744.0 | 414.0 |
| | Std | 1.3337 | 16.0609 | 30.7173 |
| Broadcast Read ($\mu s$) | Avg | 73.5246 | 262.7728 | 559.6167 |
| | Max | 84.0 | 660.0 | 636.0 |
| | Std | 1.5074 | 14.2807 | 36.5718 |

**Figure 2.27:** Ethernet quadlet read/broadcast read timing data on Xenomai, Linux and OS X

through the raw Ethernet protocol. These tests verified the cross-platform capability of our design, but also demonstrated the importance of a real-time platform for a control system. Thus, cross-platform capability is more important for different real-time operating systems, such as Xenomai, Matlab Simulink Real-Time, and QNX.

#### 2.6.3.4 Ethernet Bridge Timing Model

We model the round-trip cycle time for the bridge-based design as:

$$T_{bridge} = T_{sw} + T_b + N\frac{8\left(S_w + S_r\right)}{BW_e} + N\frac{8S_w}{BW_f} + NT_{sf} \tag{2.9}$$

where $BW_f$ is the FireWire bandwidth (400 Mbps for IEEE-1394a), $T_b$ is the bridge delay, $T_{sf}$ is the time required for each slave to broadcast its data to the bridge node (5 $\mu$s), and $S_w$ and $S_r$ are the sizes (in bytes) of the write and read packets, respectively. We assume that the bridge delay is constant because it can immediately begin processing an incoming (Ethernet or FireWire) packet and start transmitting the outgoing (FireWire or Ethernet) packet. Furthermore, because it is implemented in an FPGA and uses a 25 MHz 16-bit parallel interface to the Ethernet MAC, we assume that $T_b$ is negligible.

For the da Vinci Research Kit, the packet sizes (not including headers, checksums, etc.) are $S_r = 68$ bytes and $S_w = 16$ bytes. In anticipation of an expected change to the dVRK system write packet to include the control register (quadlet), $S_w = 20$ bytes is used instead of $S_w = 16$ bytes in the following analysis and experiments. Using these values, and the other values presented above, produces the following linear equation:

$$T_{bridge} = T_{sw} + 12.44N \qquad (\mu s) \tag{2.10}$$

We measured the round-trip time for the Ethernet/FireWire bridge for setups with 4, 5, 6, 7, and 8 nodes. Performing a linear regression yielded a slope of 16.08 $\mu$s/node and an intercept of 35.12 $\mu$s. Thus, we estimate $T_{sw} = 35.12 \mu s$. We note that the measured slope (16.08) is larger than the computed slope (12.44), which indicates that our model may be missing some sources of delay or that some of our parameter estimates may be inaccurate.

### 2.6.4 Discussion

We developed an Ethernet-to-FireWire bridge that enables real-time control of a distributed system from a central PC. Real-time control is possible because the number of Ethernet transactions can be limited to two or three (depending on the protocol), regardless of the number of distributed nodes. In this manner, our system offers benefits similar to EtherCAT but utilizes only commodity network protocols (Ethernet and FireWire) and thus our complete hardware/software design is available open source.

The addition of the Ethernet-to-FireWire bridge node and associated real-time driver improves I/O performance, significantly reducing the maximum I/O time at the cost of a slight increase in the average I/O time (48 $\mu s$ for the dVRK System). From a systematic level, the bridge acts as a buffer or switch between two fieldbuses, Ethernet and FireWire. FireWire is designed as a real-time control fieldbus, however, real-time performance is only guaranteed within the embedded subsystem. Ethernet is not intrinsically designed as a real-time transmission media, but has a wide range of real-time support benefitting from its ubiquitous applications. The bridge approach leverages the strengths of two different transmission media (Ethernet and FireWire), while compensating for the drawbacks of each to achieve high bandwidth hard-real-time control performance. Although demonstrated on the da Vinci Research Kit, this approach is generally applicable to other systems.

The success of this bridge design has led us to adopt an upgraded design of the FPGA board with an additional integrated Ethernet port, as shown in Figure 2.28. With this upgrade, any FPGA board can serve as a bridge node. When a frame from the control computer arrives, the FPGA parses the FireWire packet inside the

Ethernet frame and responds to the packet directly if itself is targeted (i.e., the bridge's node ID matches the FireWire packet's destination node ID). Otherwise, it forwards the packet to the FireWire network, waits for responses and forwards any response packet from other FireWire nodes. This approach eliminates the need for a separate bridge board, and the board remains backward compatible. There is an ongoing effort to upgrade the existing FPGA firmware to support this bridge feature.

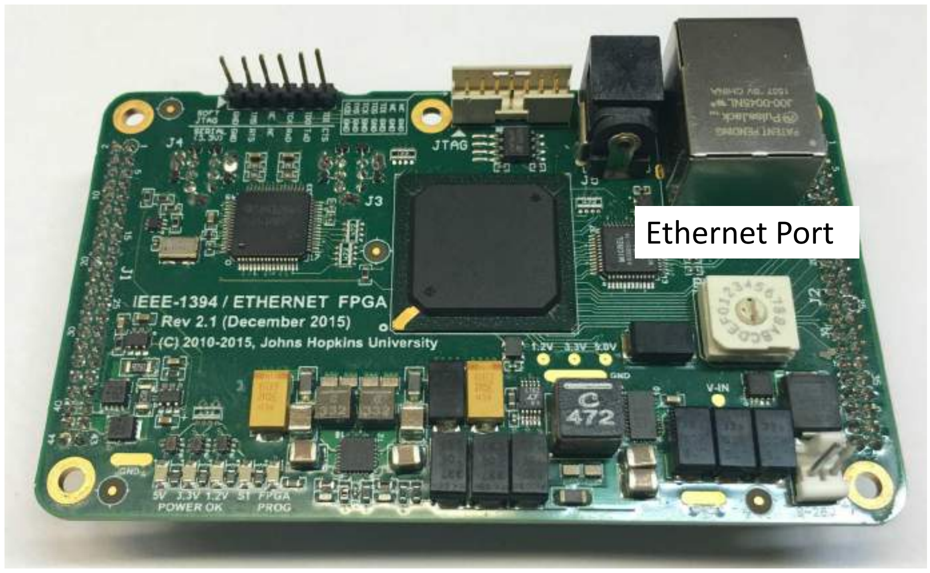

Ethernet Port
IEEE-1394 / ETHERNET FPGA
Rev 2.1 (December 2015)
(C) 2010-2015, Johns Hopkins University
JTAG

**Figure 2.28:** Second generation IEEE-1394 FPGA board with an integrated Ethernet port

## 2.7 Performance Comparison with EtherCAT

We selected FireWire in 2006 and developed a broadcast protocol to further improve its performance, but the obvious question is whether this is still a good choice, given the wider deployment of other Ethernet based fieldbus systems, particularly EtherCAT [73, 51]. As in the robotics field, several groups have reported using EtherCAT as their fieldbus of choice [96, 40]. In this section, a performance comparison study is presented.

### 2.7.1 Introduction to EtherCAT

In 2003, the EtherCAT protocol was initially introduced by Beckhoff Automation GmbH at the Hannover Fair and the standard has now been opened up and handed off to the EtherCAT Technology Group (ETG). Among all industrial real-time Ethernet systems, EtherCAT delivers the most deterministic response (100 axes in 125 $\mu s$) and has reached the tipping point for market acceptance [51].

Similar to the FireWire broadcast protocol, EtherCAT is a link layer level protocol. As shown in Figure 2.29, with EtherCAT, several slave nodes are typically networked with a single bus master node in a ring topology. During each cycle, the EtherCAT frame initiated by the master is passed through the next slave node, which extracts relevant output data and stuffs its own input data into the packet at a predefined location “on-the-fly”, until it reaches the end of the chain and is sent back to the master. These cyclic frames are referred to as Process Data in EtherCAT terminology. EtherCAT also supports a mailbox-based protocol for acyclic data exchange (Service Data).

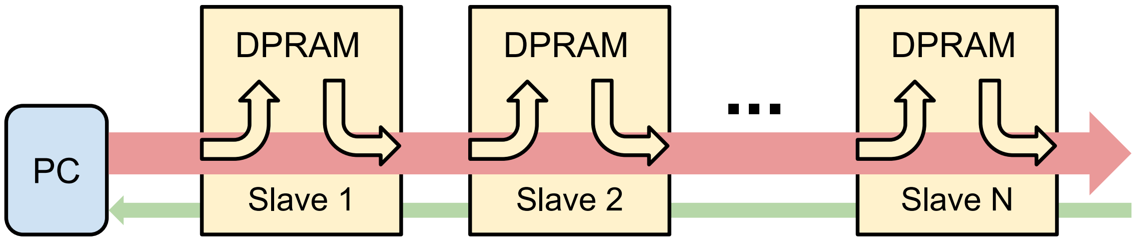


**Figure 2.29:** An example EtherCAT system

At the physical layer, EtherCAT relies on standard Ethernet and transmits frames with the standard Ethernet telegram structure at 100 Mbps, but with an entirely different mode of operation. At each cycle, control packets are not sent to each slave node separately as in other approaches, but rather utilize a single telegram with the headers and process data of all stations defined in consecutive sub-telegrams. The benefits of the approach are two-fold: 1) it minimizes the number of transactions on the master node, which typically is a conventional computing platform and therefore subject to software-induced latency, and 2) it increases the user data rate as the Ethernet protocol requires a minimum of 64 bytes payload size and typical control packets are small (below 15% in motion control applications [51]). If a Process Data packet exceeds the maximum Ethernet payload, it is distributed across multiple EtherCAT frames.

This special mode of operation requires a special hardware implementation on slave nodes. In normal operation, an Ethernet slave typically has only one Ethernet port, and receives and transmits data packets through the same port (typically half-duplex). An EtherCAT slave, on the other hand, generally uses two separate ports, one for receiving and one for forwarding. A special EtherCAT Slave Controller implements the receive, process “on-the-fly” and forward link layer protocol. It is typically realized as an Application-Specific Integrated Circuit (ASIC) or as a FPGA core module for the best timing performance. On the master side, EtherCAT can utilize any off-the-shelf Ethernet card and it is typically implemented as a software stack that configures and manages its communication. For example, the TwinCAT 3 from Beckhoff Gmbh is developed as an extension of Microsoft Visual Studio and runs on a Windows computer.

### 2.7.2 EtherCAT Timing Performance

As the first step of this comparison, we model, measure and verify the EtherCAT timing performance. Although several documents have reported EtherCAT timing performance [73, 51, 69], they either neglected the practical software induced latency [51], or used a non-ideal platform (JAVA). Prytz’s analysis between EtherCAT and PROFINET IRT stayed at the theoretical level (models), without measurements from hardware [69]. In our test setup, a Xenomai (2.6.3) patched Linux (Kernel version 3.5.7) computer with the RTnet [52] real-time Ethernet driver is used as the

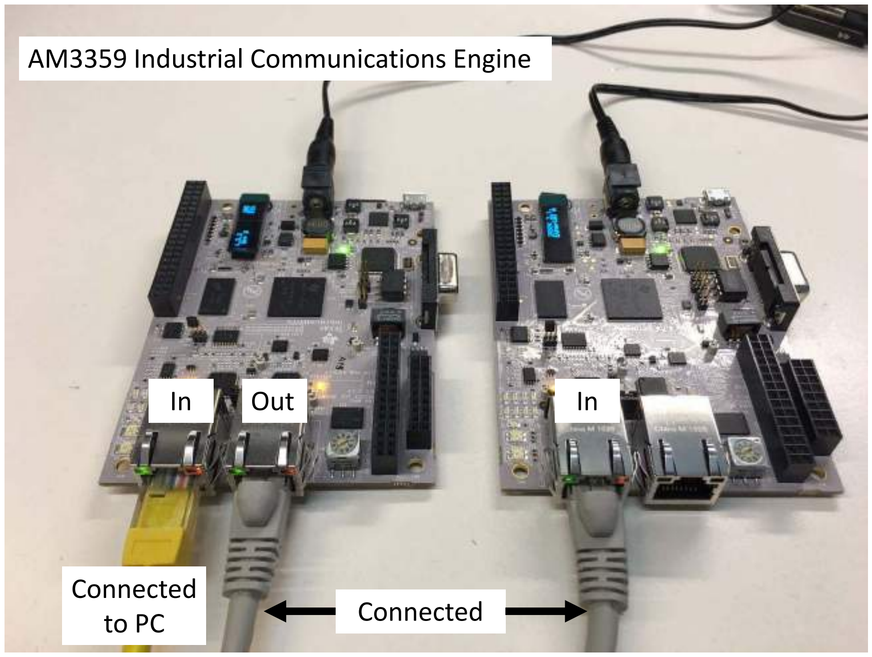


**Figure 2.30:** EtherCAT slave test board: AM3359 Industrial Communications Engine (TMDSICE3359) from Texas Instruments (TI). The control PC is connected to the left board's In port and the two boards are daisy-chained.

master control computer and the AM3359 Industrial Communications Engine (TMDSICE3359) from Texas Instruments (TI) Incorporated (see Figure 2.30) is selected as the EtherCAT slave device. For the master software stack, we used Simple Open EtherCAT Master (SOEM) from the Open EtherCAT Society due to its openness and support for Linux. The SOEM master stack has been ported to support the Xenomai real-time kernel.

#### 2.7.2.1 EtherCAT Timing Model

Prytz [69] presented a model for the round-trip time on EtherCAT. This model assumes a master forwarding time based on the packet size and Ethernet bandwidth (e.g., 100 Mbps), a maximum delay of the master PHY (expected to be less than 0.5 $\mu$s), and a 1 $\mu$s forwarding time per slave node (this includes cable and slave PHY delays). The model does not, however, consider software overhead. We add a constant term, $T_{sw}$, to model software overhead. Essentially, we assume that the software overhead due to the increase in packet size (as the number of nodes is increased) is negligible compared to the total software overhead. This is a reasonable assumption because the amount of time required to copy packet data from one memory location to another is small compared to other tasks done by the operating system and driver, such as context switches. Thus, we can simply model the round-trip EtherCAT timing as:

$$T_{ecat} = T_{sw} + N \times \frac{8\,(SZ_{out} + SZ_{in})}{BW_e} + N \times T_{slave}, \tag{2.11}$$

where $N$ is the number of slave nodes, $SZ_{out}$ is the number of bytes written from the PC to each slave, $SZ_{in}$ is the number of bytes sent by each slave to the PC, $BW_e$ is the Ethernet bandwidth in Mbps (100), and $T_{slave}$ is the forwarding time of each EtherCAT slave node. This equation assumes the use of a single packet, which is reasonable given the number of bytes required in our application. It also does not include the overhead for the standard Ethernet header and CRC, which are the same for all Ethernet-based protocols and captured in $T_{sw}$, or the overhead for the EtherCAT header, which is assumed to be negligible.

#### 2.7.2.2 Model Parameter Measurement

As shown in the previous section, three parameters are of interest, namely, software latency ($T_{sw}$), EtherCAT node latency ($T_{slave}$) and transmission speed ($BW_e$). Experimentally, we measured and identified these three parameters.

**Bandwidth ($BW_e$) and Software Latency ($T_{sw}$)**: From FireWire testing, we learned that fieldbus medium transmission can be lower than nominal speed. One reason is that some details, such as the time required to transmit the frame header and CRC, is not included in the model. To identify EtherCAT transmission speed (nominally 100 Mbps) and software latency, we used a setup with a single EtherCAT slave board, loaded slave firmwares with different payload sizes (8/10/12/14/16/18/20 quadlets) and measured the cycle time for each payload size. The timing measurement program runs 100,000 cycles and outputs the mean cycle time. As shown in Figure 2.31, EtherCAT transmits data at 93.10 Mbps ($BW_e$ = 93.10 Mbps), slightly under the nominal 100 Mbps, and has a software latency of 26.79 $\mu s$ ($T_{sw}$ = 26.79 $\mu s$).

**EtherCAT Slave Forwarding Latency** $T_{slave}$: TI's EtherCAT Slave Controller solution is based on its Programmable Realtime Unit (PRU) co-processors and has a reported end-to-end forwarding latency of less than 700 ns [35]. To measure it experimentally, we measured the cycle time on setups with 1 to 4 boards while keeping the overall EtherCAT packet size unchanged. For example, in a one slave board setup, the slave firmware has a payload size of 20 quadlets and in a two board setup, one of them has a payload of 18 quadlets and the other one has 2 quadlets. This avoids

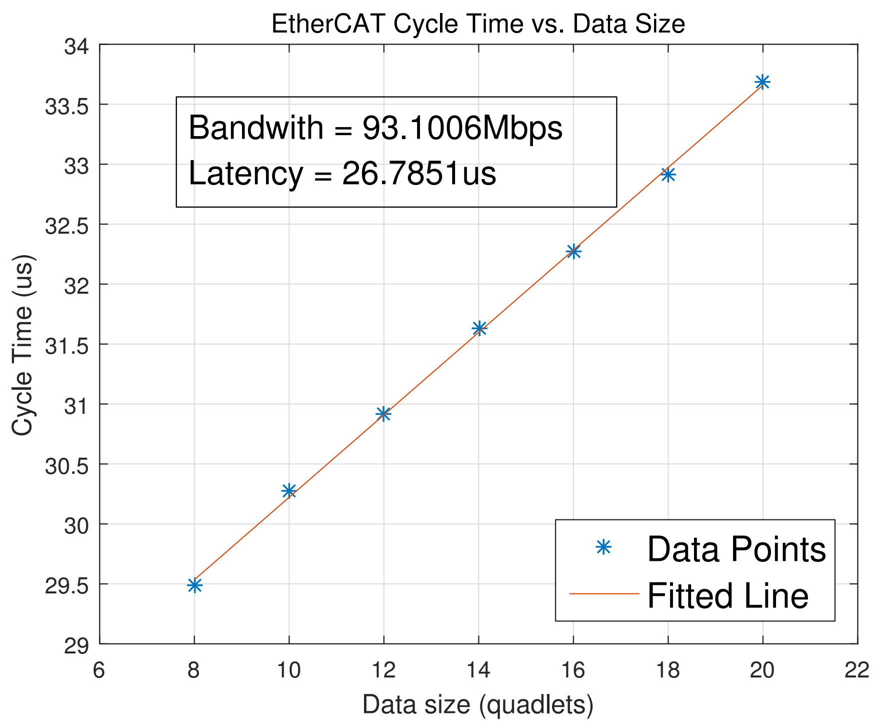


**Figure 2.31:** EtherCAT timing performance: data size (quadlets) versus cycle time ($\mu s$)

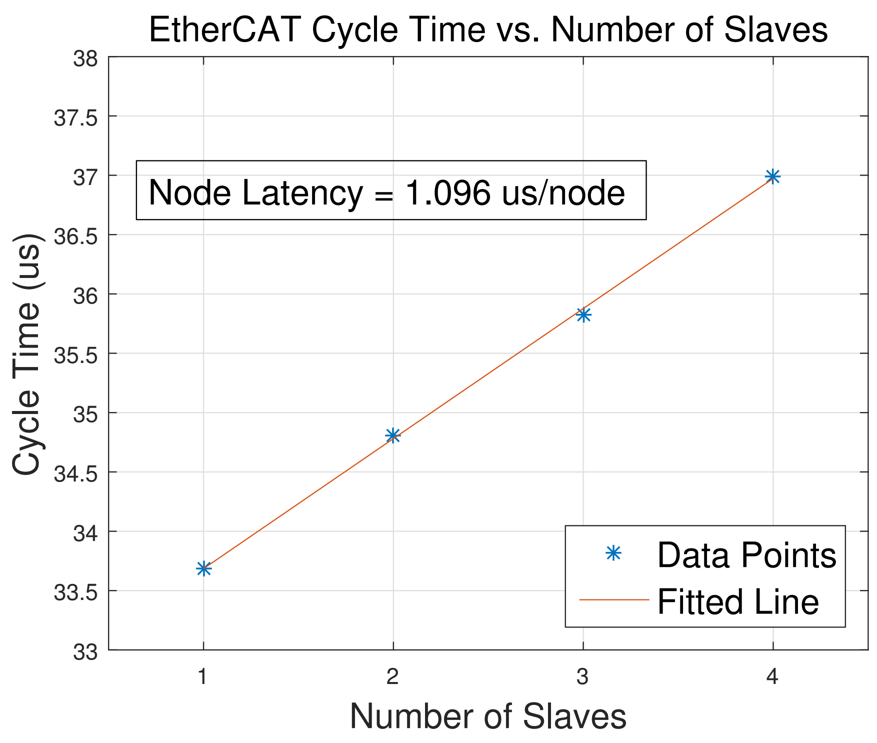


**Figure 2.32:** EtherCAT slave forwarding latency: number of nodes versus cycle time ($\mu s$)

timing differences that could otherwise be introduced by different Ethernet frame sizes. Figure 2.32 plots cycle time versus number of boards. By fitting a line, we found that the forwarding delay of each node is 1.10 $\mu s$ ($T_{slave} = 1.10\ \mu s$), which is consistent with the 1 $\mu s$ value reported by Prytz [69] and slightly higher than the claimed 700 ns delay. The difference may come from Ethernet cable latency as well as limited data samples.

#### 2.7.2.3 EtherCAT Model Verification

The EtherCAT model uses the same dVRK packet sizes as the Ethernet Bridge presented in Section 2.6.3.4, which are $SZ_{out} = 20$ bytes and $SZ_{in} = 68$ bytes. Using

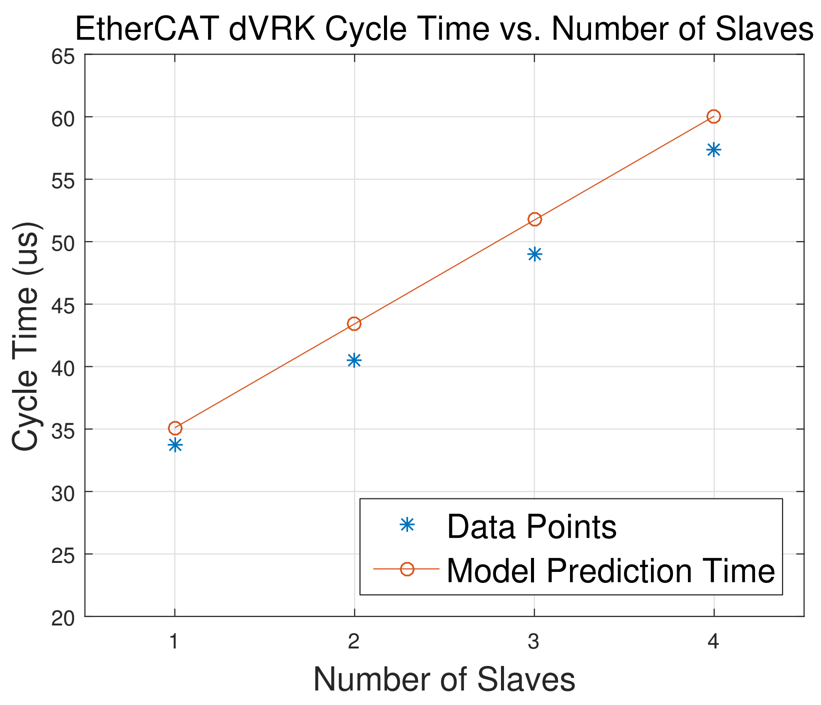


**Figure 2.33:** EtherCAT performance: cycle time ($\mu s$) versus number of nodes

these values, and the measured $T_{sw}$, $BW_e$ and $T_{slave}$ values, Equation (2.11) can be rewritten as:

$$T_{ecat} = 26.79\mu s + N \times 8.662\mu s \tag{2.12}$$

This equation is the model of EtherCAT timing data for da Vinci Research Kit. For verification, we loaded firmware with the same payload size into TI's TMDSICE3359 boards and measured the cycle time on 1 to 4 boards, as shown in Figure 2.33. The blue star represents the cycle time collected on the real hardware and the red circle is the predicted cycle time using Equation (2.12). The model matches the measured data well, with a slightly higher prediction, which, we suspect, is from measurement errors due to the small number of boards.

### 2.7.3 Time Performance Comparison on dVRK

We compare timing data collected with hardware, as well as predicted data using models. To obtain a fair comparison, we assume that all implementations use a single node to control four robot axes. Figure 2.34 plots data collected from hardware and Figure 2.35 uses predicted data from the models developed in earlier sections. All three protocols can support systems with large numbers of nodes and show good scalability. The Ethernet/FireWire bridge design has a higher slope (i.e., the time increase when an additional board is added), due to the extra forwarding step involved. The FireWire broadcast protocol has a higher overhead, but can scale better compared with EtherCAT thanks to the higher transmission speed (400 Mbps vs. 100 Mbps). For large systems, the FireWire broadcast protocol can outperform the EtherCAT protocol.

### 2.7.4 Discussion

The study showed that the proposed FireWire protocol and Ethernet/FireWire bridge provide performance comparable to EtherCAT. All three can provide sufficient performance for high-rate control. FireWire has a higher bus bandwith (400 Mbps for IEEE-1394a and typically 800 Mbps for IEEE-1394b) compared to EtherCAT, which is limited to 100 Mbps, but this is not likely to be significant.

There are several obvious benefits to using EtherCAT: (1) while many computers

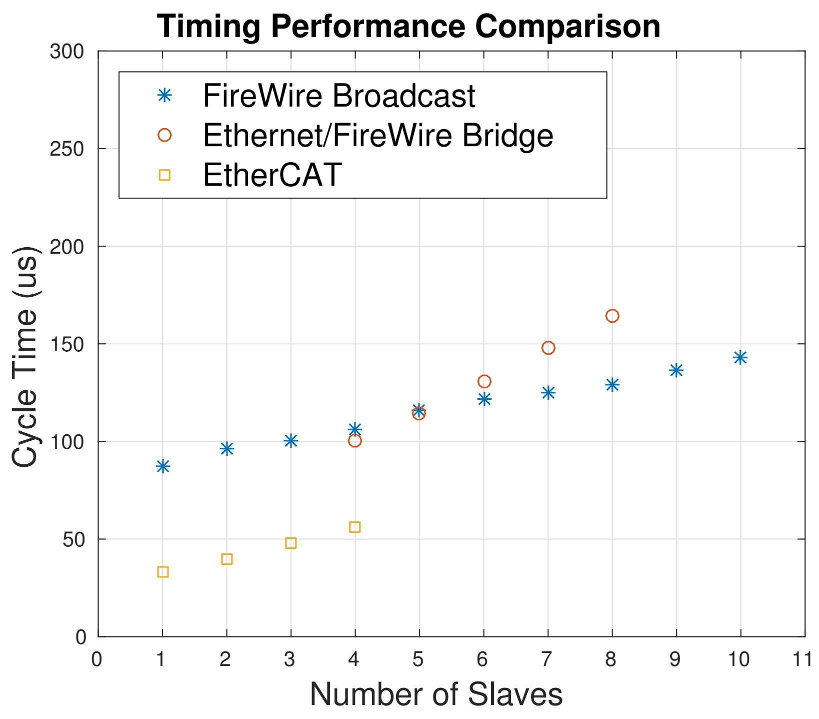


**Figure 2.34:** Comparison of EtherCAT, FireWire Broadcast and Ethernet/FireWire Bridge performance using data measured on physical hardware.

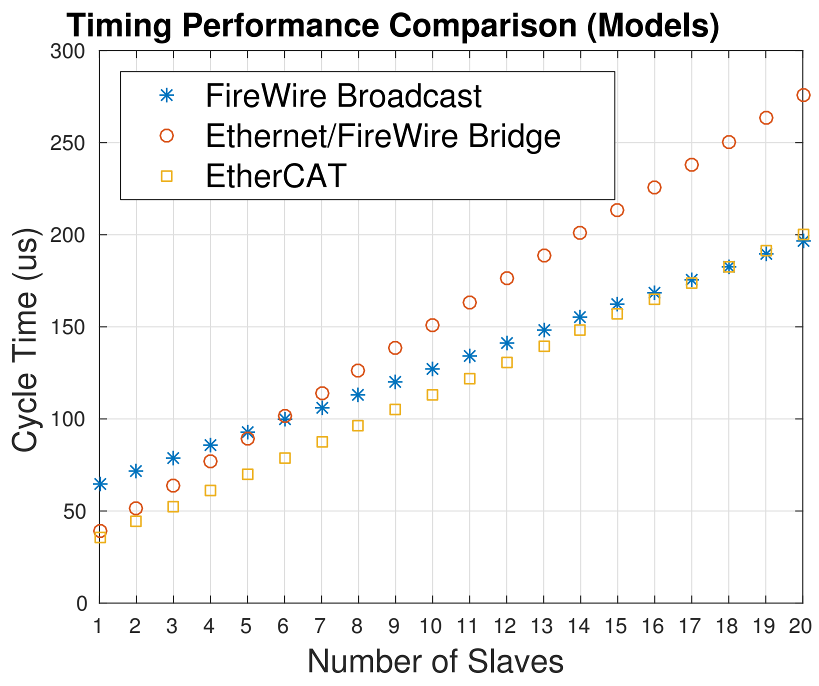


**Figure 2.35:** Comparison of EtherCAT, FireWire Broadcast and FireWire Bridge performance based on models

have a FireWire port, it is not as ubiquitous as an Ethernet port; (2) EtherCAT uses standard Ethernet cables, which are more easily routed inside a robot mechanism (with an option for high-flex cables) and can be longer than FireWire cables; and (3) there are more vendors providing control components with EtherCAT interfaces. The first benefit is also provided by the Ethernet/FireWire bridge configuration.

At this point, we can identify a few advantages of our FireWire and Ethernet/-Firewire systems: (1) they are easier to dynamically reconfigure by connecting and disconnecting nodes, as compared to the configuration tools and files required for EtherCAT systems; and (2) they are completely open source and therefore simple and inexpensive for researchers to implement custom slave nodes or to customize the protocol. As an example, since we use broadcast to transmit status data from the FPGA, all FPGA nodes can receive these packets and have information about the whole robot system. Besides allowing any FPGA node to act as the Hub, this is potentially valuable in a multivariable control system. Another use of this information is to provide an extra safety feature (e.g., power shutdown if another FPGA node fails).

From the control perspective, there is an important difference between the protocols. As shown in Figure 2.36, at the beginning of a control cycle, an EtherCAT based controller first processes existing data in the In buffer, which is received from previous cycle, then does control computation, then initiates an EtherCAT transmission. This transmission sends out control commands and fetches feedback. However,

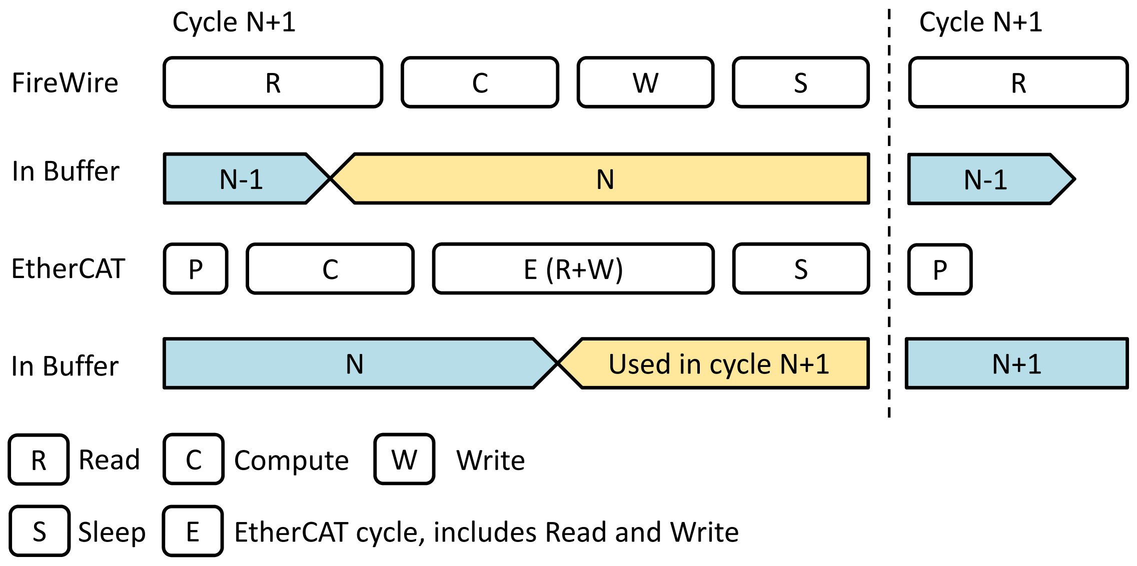


**Figure 2.36:** Comparison of read buffer timing between FireWire broadcast and EtherCAT protocol

the feedback is stored in the buffer and is used in the next control cycle. This means that the control computation is always using an older feedback data. This can be problematic, especially at a slower update rate. The FireWire broadcast protocol, on the other hand, fetches and uses the latest input data for control computation and transmits control output commands immediately. In fact, we can use the broadcast write command packet as a read query packet as well and get an even better timing performance.

## 2.8 Conclusions

In summary, this chapter presents the system aspect of a scalable, high-performance control architecture motivated by the dVRK. In historical context, we reviewed the rationale for the selection of the distributed I/O and centralized computation architecture and revealed that the major limitation when scaling to large systems is the number of transactions from the control computer. This led to the development of the first contribution: a broadcast-based communication protocol for scalable real-time performance. This protocol utilized the broadcast and peer-to-peer capabilities of the FireWire bus. In a control cycle, the PC broadcasts a read query packet containing a sequence number and a board exist mask to all control nodes (FPGA boards). Upon receiving the query packet, each node transmits its status using an asynchronous broadcast block write packet at its pre-allocated time slot. All status broadcast packets are received and cached by each FPGA node, so that all nodes maintain a copy of the entire robot status. Then, the control PC reads the status of all nodes from any FPGA board. This protocol is similar to the FireWire isochronous transfer mode, except that it can have an arbitrarily specified frequency. After the PC computes the control commands, it transmits a broadcast packet that is received by all nodes. Compared with protocols such as Ethernet POWERLINK, this protocol reduces the number of transactions from the control PC to two for reading feedback and one for writing commands. Also, the optimization on the FPGA minimizes timing gaps between FireWire frames and makes the use of concatenated asynchronous transactions possible. In general, this protocol can be applied to any fieldbus with broadcast and peer-to-peer transfer capabilities. For other fieldbuses, reducing the number of transactions and optimizing the link layer at the hardware level can be applied as general strategies to design an efficient I/O protocol. This contribution has been published in [19]. **Credit:** Zihan Chen.

In Section 2.6, a bridge design is proposed to enable real-time control over a conventional Ethernet interface, which is a more common interface on modern computers and has up-to-date Linux real-time driver support. When compared to the state-of-the-art EtherCAT fieldbus, our approach shows comparable performance. In this work, we demonstrated a strategy to support hard real-time control using a common interface (Ethernet) on an existing control network design. This strategy can be applied to support other existing hardware using other fieldbusses. This work also contributed an open-source implementation that converts an Ethernet interface to a Firewire network. Although this implementation is by no means complete and does not support FireWire isochronous transfers, it serves as an example and a starting point. To our knowledge, no such converter is available open-source or commercially. This bridge design was published in [70]. **Credits:** Developed in a collaboration with Long Qian. The general concept and design was developed by Zihan Chen. Long Qian implemented the initial bridge FPGA board firmware and helped with experiments.

The two contributions solved the identified performance problem and provided a solution for high-performance scalable control of high DOF robot systems. However, it is desirable to re-evaluate the use of FireWire compared to other fieldbus options, especially EtherCAT. We presented a study comparing the timing performance of the broadcast protocol, the bridge design and EtherCAT. The study was done both analytically and experimentally. The result shows that the FireWire broadcast protocol, with or without the Ethernet/FireWire bridge, has comparable performance. In addition, the timing analysis of EtherCAT is a contribution of this work, as previous studies were either performed using a non-optimal platform [73] or purely analytically [69]. **Credits:** Zihan Chen.

# Chapter 3

# Software Architecture

This chapter presents a software architecture that supports high-performance, low-level control as well as flexible, high-level ROS-based multi-process control. The architecture is specifically used for the da Vinci Research Kit (dVRK), but could be more generally applied to other robot systems.

## 3.1 Introduction

We laid out key requirements that influenced our choice of a centralized processing and distributed I/O architecture, and presented a broadcast-based protocol that enables IEEE-1394 (FireWire) to serve as a high-speed fieldbus that scales to a multi-robot system, such as dVRK. We also presented the concept of a bridge design, that couples a convenient interface, such as Ethernet or USB, with a high-speed real-time fieldbus, such as IEEE-1394. This design ensures that at the hardware level, the architecture can scale to support multiple high DOF robots and allows robot systems to be reconfigured by changing the network cabling and safety/E-stop chain. A proper design of the software architecture is required to support this hardware architecture's high performance, scalability and reconfigurability. Meanwhile, the software architecture, as a programming interface, should provide a clean low-level interface as well as a flexible high-level interface.

A few software design considerations are as follows:

1. Real-time performance for high-frequency, low-level robot control.

2. Software support of the hardware scalability.

3. Easy reconfiguration, such as adding or removing arms or even splitting the system into multiple independent setups, preferably without the requirement of recompiling code.

4. Use of a familiar software development environment, such as Linux with the GNU Compiler Collection (GCC), for all levels.

5. Ability to integrate with other high-level robot components and development environments, such as MATLAB and Python, via middleware.

These considerations led to the use of C++ as the programming language and Linux as the operating system (ideally Xenomai [29] patched for real-time performance), though most of the software is portable to other platforms, such as VxWorks and QNX. The key layers of the software architecture, shown in Figure 3.1, derive from the following design decisions, which are presented in subsequent sections:

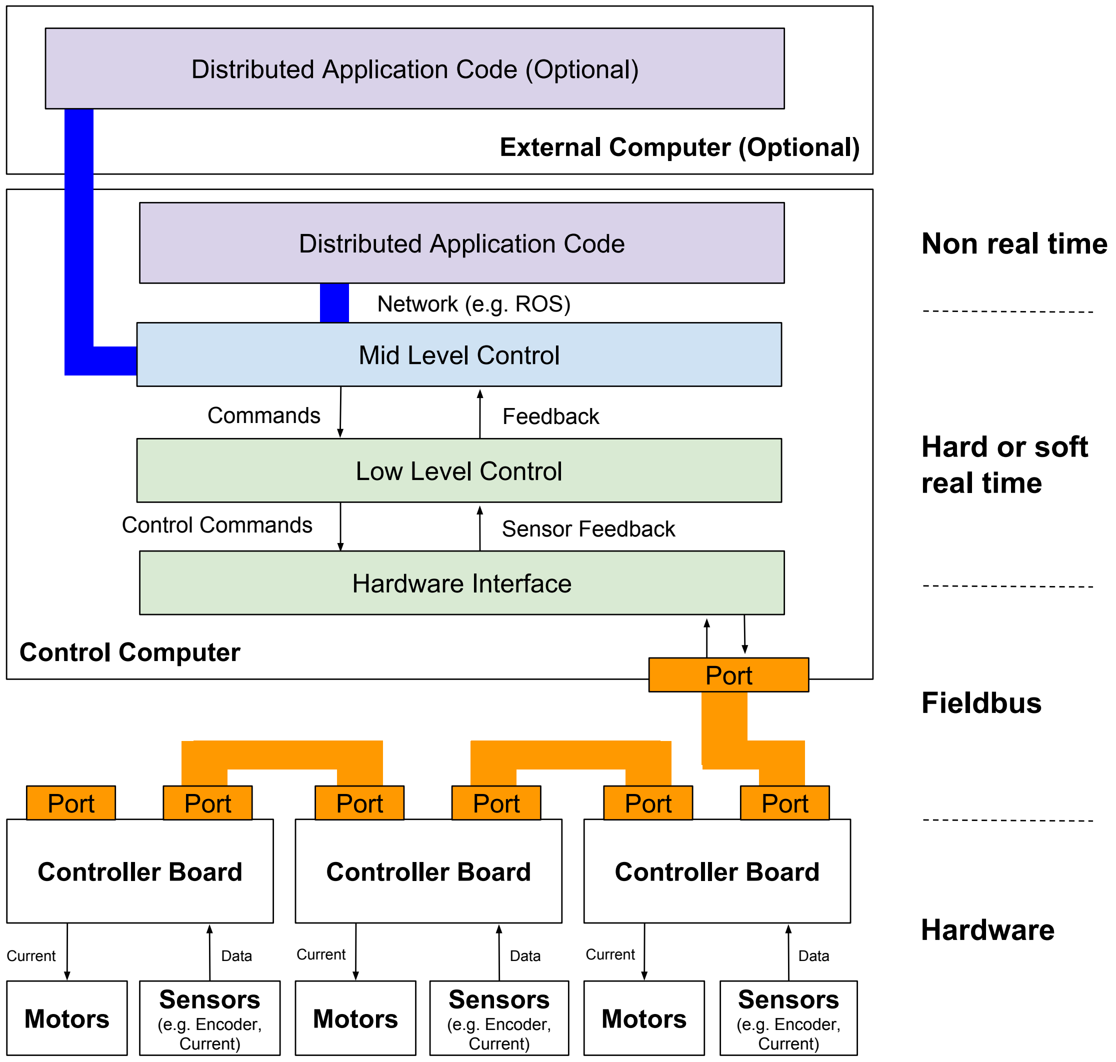


**Figure 3.1:** da Vinci Research Kit (dVRK) software control architecture

1. An efficient (low overhead) software interface to the fieldbus, which satisfies the requirements for scalability and reconfigurability. This is discussed in Section 3.4.

2. A real-time, component-based framework that enables high bandwidth, low latency control. Section 3.5 describes the design of the real-time software layer for the dVRK, which is based on the open source *cisst* libraries developed at JHU [23, 41].

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 [42] based on *cisst* and Internet Communications Engine (ICE), but has since transitioned to ROS [71], as discussed in Subsection 3.6.

## 3.2 Thesis Contributions

The da Vinci Research Kit software has been developed by several individuals and utilizes software infrastructure, such as the *cisst* libraries and Surgical Assistant Workstation (SAW), that have been developed over more than a decade. My contributions include: (1) design of the architecture, with Dr. Peter Kazanzides; (2) implementation of the initial hardware interface layer, which was subsequently updated by Anton Deguet, Jonathan Bohren, and others; (3) timing studies to compare communication performance of ROS and *cisst*, which justify the use of the *cisst* ExecIn/ExecOut interfaces for low-latency data exchange between the low-level control and hardware interface layers; and (4) implementation of the ROS interfaces, with assistance from Jonathan Bohren and Anton Deguet.

The research contribution is in the architecture, which demonstrates a design for scalable real-time control of multiple robots. While some aspects of the architecture are conventional, such as the use of hierarchical multi-rate control, the novelty is the way that it presents each robot as an independent entity, even though they share resources such as a single communication bus and single thread for low-level control. The work presented in this chapter was published in [18].

## 3.3 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 WAM [78] (Barrett Technology, Inc. Cambridge, MA) is a 7 degrees 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 a *Puck*, installed on each joint. These modules are interconnected 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, Bohren et al. [9] and Lages et al. [55] implemented control architectures that use ROS for the high level interface and the Open Robot Control Software (OROCOS) [11] 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-directional 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 [96]. 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 [32] 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 (UCSC) Bionics lab, and was based on Raven I developed at UW [59]. The UW/UCSC 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 125 $\mu s$ [27]. The Raven II has been integrated with ROS, which allows easy integration with other robotic software.

## 3.4 Low-Level Hardware Interface Layer (Fieldbus)

All robot control code interacts with the fieldbus through the Hardware Interface Layer. The layer is provided by a C++ library that enables direct access to the raw I/O data via the IEEE-1394 bus. This library has no external software dependencies, other than `libraw1394`, which is a standard Linux library for communication over IEEE-1394. Other drivers, such as RT-FireWire [100], could be used to obtain hard real-time performance (although RT-FireWire is no longer maintained). There is also a Microsoft Windows implementation of `libraw1394` [90]. The API consists of two main abstract base classes: a `BasePort` class to represent a hardware port (e.g., FireWire or Ethernet), and a `BoardIO` class to represent one FPGA node on the bus. Currently, the only derived class from `BoardIO` is `AmpIO`, which corresponds to the FPGA/QLA board set. Figure 3.2 presents a Unified Modeling Language (UML) class diagram of these base and derived classes. For a typical system, one port will connect to multiple FPGA nodes; thus the `BasePort` object maintains a list of `BoardIO` objects. The `BasePort` class contains two methods, `ReadAllBoards` and `WriteAllBoards`, which read all feedback data into local buffers and transmit all output data from local buffers, respectively. This allows the class to implement more efficient communication mechanisms, such as the broadcast write and consolidated read described in previous sections. The `AmpIO` API provides a set of functions to extract feedback data, such as encoder positions, from the read buffer, and to write data, such as desired motor currents, into the write buffer. All data types are unsigned integers because they are stored as counts (or bits) in FPGA registers.

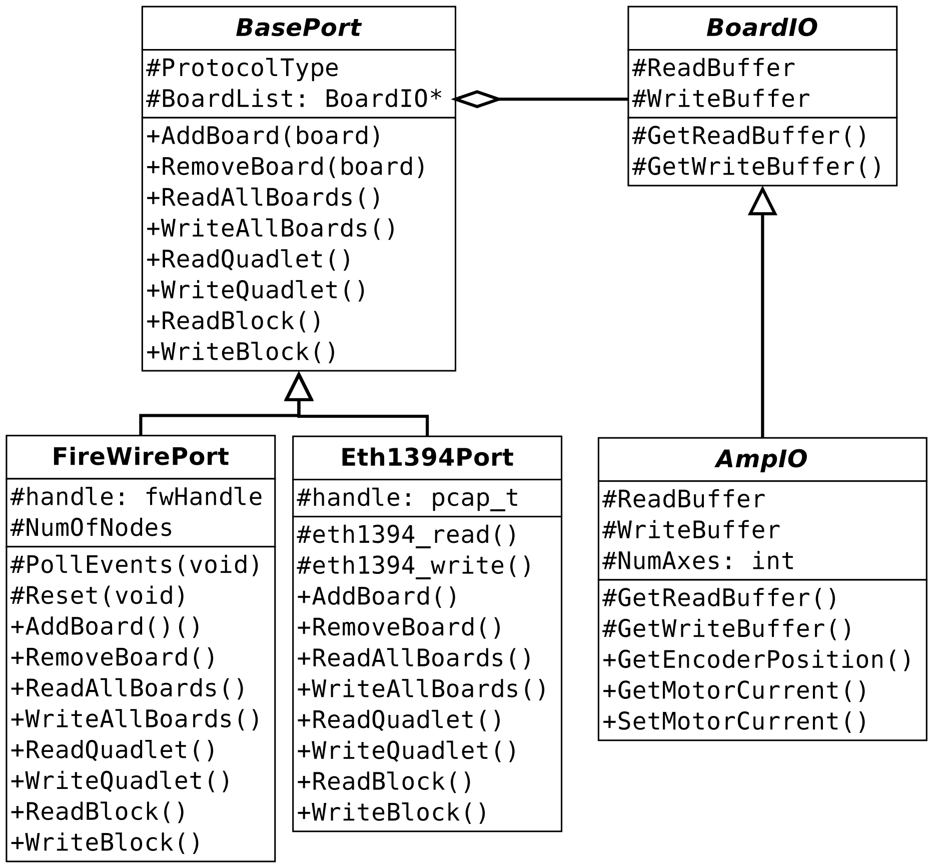


**Figure 3.2:** UML class diagram of interface software (subset of class members shown): the design can scale and support different field bus implementations as well as different board designs.

## 3.5 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 Figure 3.1. 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. My contributions are: (1) the Design Analysis of the inter-component communication mechanisms, presented in Section 3.5.2, which favors the use of the *cisst* synchronous (ExecIn/ExecOut) communication mechanism between components in the same process, and (2) the design and implementation of the *mtsRobotIO1394* class, described in Section 3.5.3, that manages the IEEE-1394 fieldbus and uses its *ExecOut* interface to synchronously execute the low-level control components (*mtsPID*) for each manipulator. Much of the remaining low-level and mid-level control software was implemented by Anton Deguet.

### 3.5.1 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 or sharing a single thread. 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.

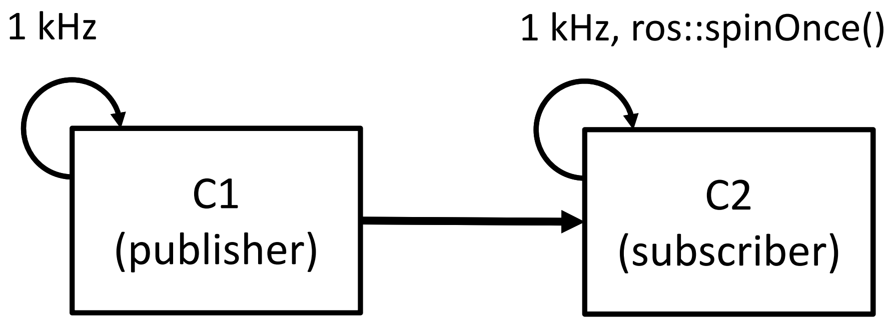
### 3.5.2 Design Analysis

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.

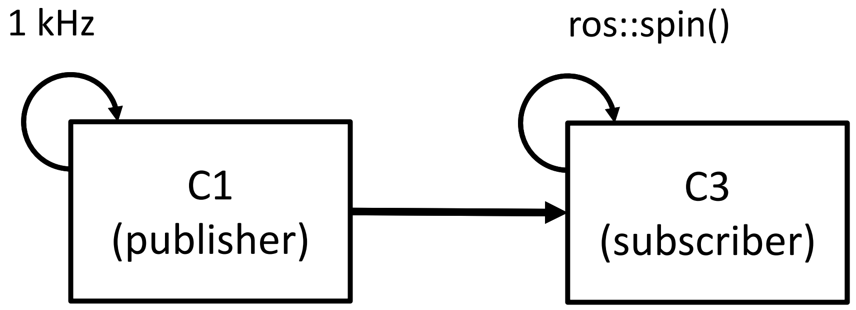
First, we consider the ability to handle data exchange between components with different execution rates in a timely and reliable manner. The key requirement is to deliver the latest data to the consumer component with minimum latency and overhead. In particular, we consider the case where the consumer 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. One option is to spawn a process for each component and rely on inter-process communication (IPC) mechanisms such as ROS for data exchange. Figure 3.3 shows two setups we evaluated. Both setups use the same publisher component C1, which is a ROS node, running at 1 kHz. The published messages from C1 are subscribed either by C2 running `ros::spinOnce()` (equivalent to periodically polling) or C3 running `ros::spin()`. Timing data is collected by time stamping a ROS message before it is published, and computing the difference between the wall clock time and the stamped time in the subscriber callback function. In a periodic polling setup, the communication latency is depdendent on the loop rate. When C2 calls `ros::spinOnce()` at 1 kHz (see Figure 3.4(a)), the



(a) Subscriber calls `ros::spinOnce()` at 1 kHz

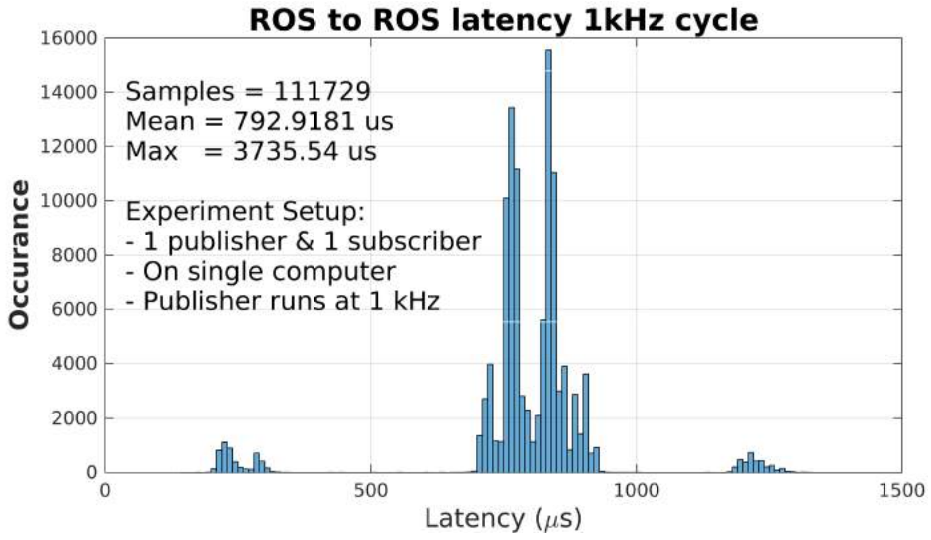


(b) Subscriber calls `ros::spin()`

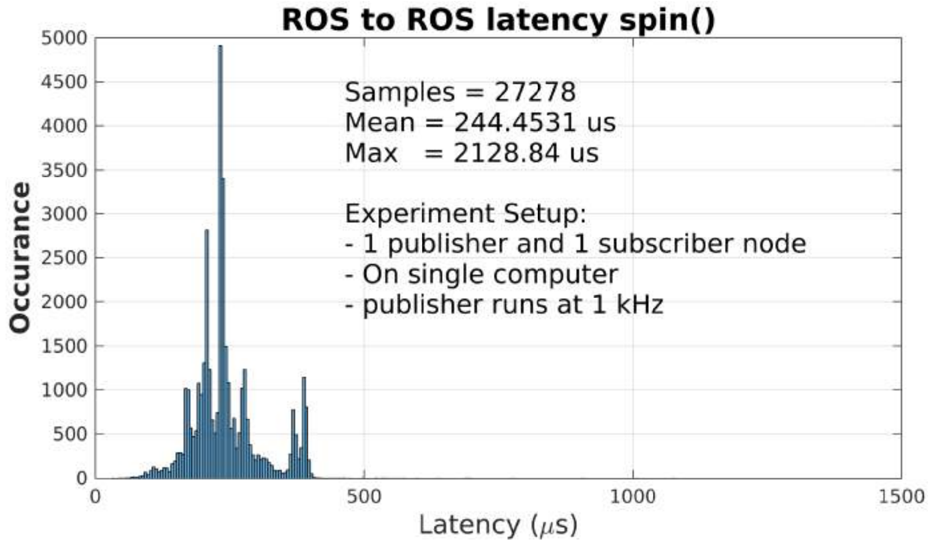
**Figure 3.3:** ROS system publisher/subscriber latency test setup. ROS node C1 runs at 1 kHz and publishes to ROS node C2 or C3. C2 calls `ros::spinOnce()` at 1 kHz, whereas C3 calls `ros::spin()` to wait for publisher.

mean latency between C1 and C2 is 792 $\mu s$, which is more than half the node update period. This can be improved by having the subscriber wait for the publisher, as demonstrated by ROS nodes C1 and C3, where C3 calls `ros::spin()`. Figure 3.4(b) shows that this decreases the mean latency to 244 $\mu s$, with a maximum latency of 2,129 $\mu s$. While the mean 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 requires the subscriber to wait for the publisher. 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.

A multi-threaded component-based robotic middleware, such as OROCOS from Katholieke Universiteit Leuven and *cisst* [23] from JHU, 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 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 3.5 shows the data transfer latency between two *cisst* components using the *ExecIn/ExecOut* feature. On average,



(a) Subscriber calls `ros::spinOnce` at 1 kHz



(b) Subscriber calls `ros::spin`

**Figure 3.4:** ROS system publisher/subscriber latency tests. 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.

the latency is 21.3 $\mu s$ with a maximum value of 115.2 $\mu s$. 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.

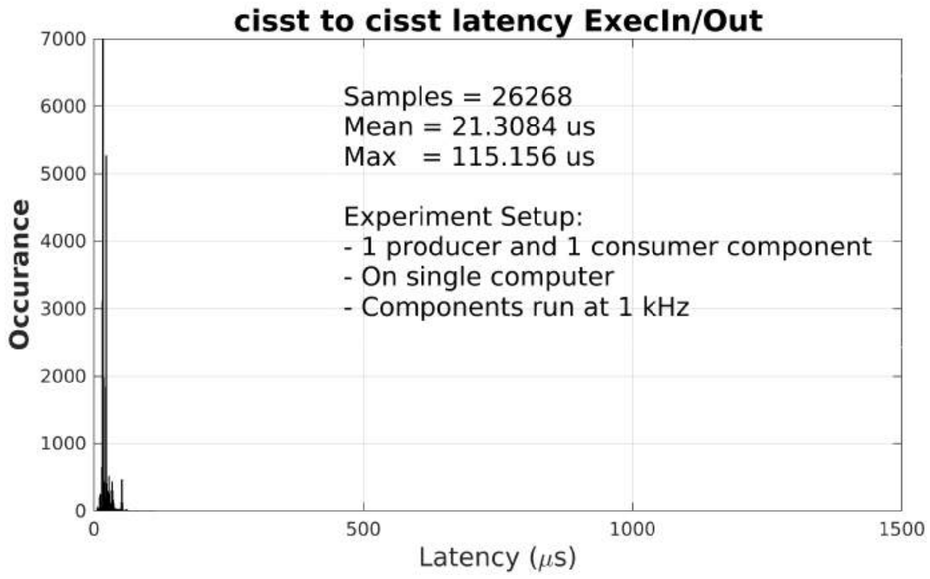


**Figure 3.5:** Communication latency in *cisst*, using *ExecIn/ExecOut* for synchronous communication; components execute at 1 kHz, same hardware/software setup as Figure 3.4(b).

## 3.5.3 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 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 3.6, the architecture consists of: (1) one hardware Input/Output (I/O) component, *mt-*

*sRobotIO1394* (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.

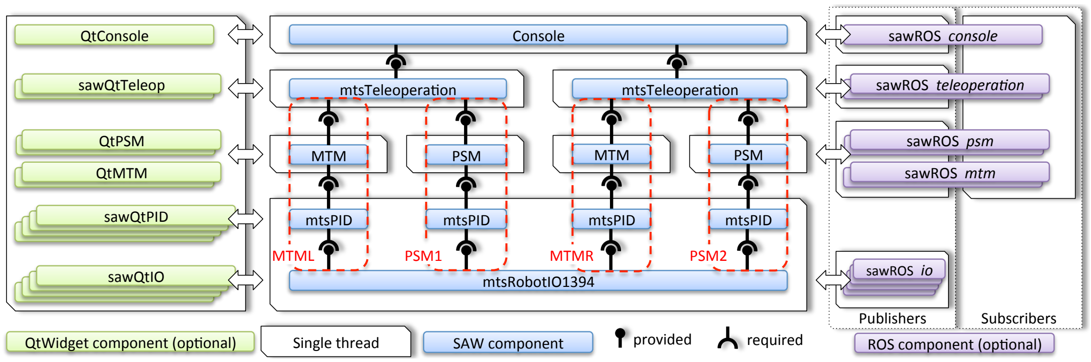


**Figure 3.6:** Robot tele-operation control architecture with two MTMs and two PSMs, arranged by functional layers and showing thread boundaries [48].

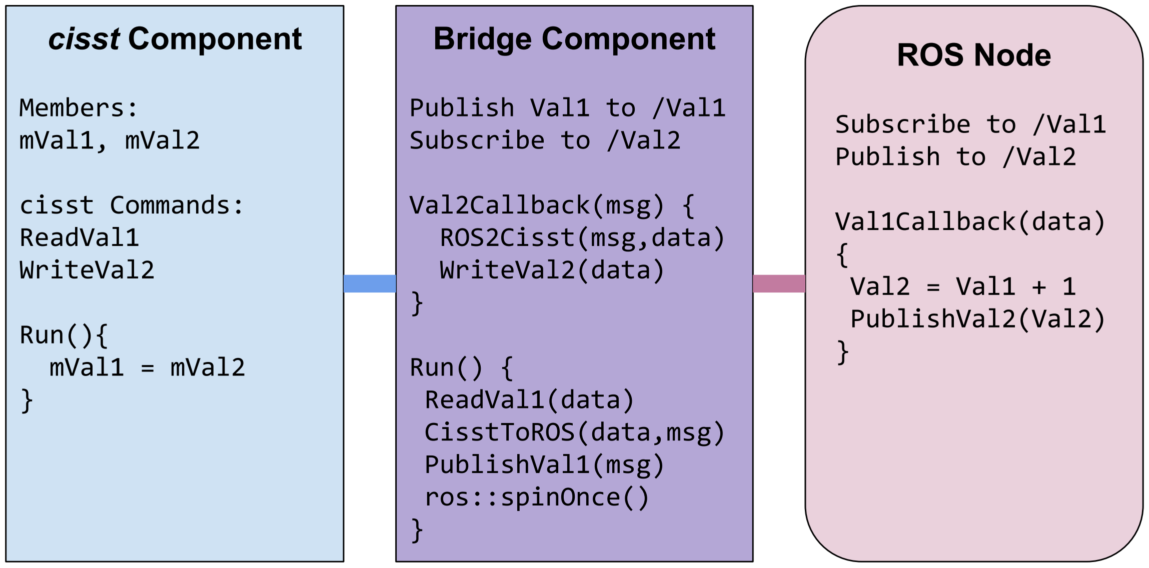
## 3.6 System integration via ROS interfaces

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 (wiki.ros.org/rviz) and MoveIt! (moveit.ros.org/). It 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 dVRK's ROS ecosystem.

### 3.6.1 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 software bridge component. The bridge component is both a periodic component (inherits from *mtsTaskPeriodic*) and a ROS node. As an *mtsTaskPeriodic* 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 Figure 3.7, 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.



```
Members:
mVal1, mVal2

cisst Commands:
ReadVal1
WriteVal2

Run(){
  mVal1 = mVal2
}
```

```
Publish Val1 to /Val1
Subscribe to /Val2

Val2Callback(msg) {
  ROS2Cisst(msg,data)
  WriteVal2(data)
}

Run() {
 ReadVal1(data)
 CisstToROS(data,msg)
 PublishVal1(msg)
 ros::spinOnce()
}
```

```
Subscribe to /Val1
Publish to /Val2

Val1Callback(data)
{
 Val2 = Val1 + 1
 PublishVal2(Val2)
}
```

**Figure 3.7:** *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`.

### 3.6.2 ROS Ecosystem

The dVRK ROS stack includes the cisst-to-ROS bridge as a package and a robot description package with ROS Unified Robot Description Format (URDF) files for the MTM, PSM and ECM. These URDF files are 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 [64] or alternate slave robot, such as the Raven-II [32]. 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.

## 3.7 Discussion and Conclusion

In this chapter, we presented a scalable, reconfigurable, real-time and ROS compatible software architecture for dVRK. The architecture includes 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 3.4 represent the transition between the distributed hardware layer and the real-time framework, whereas the cisst-to-ROS bridge defined in Section 3.6 provides the interface between the real-time framework and the ROS environment.

Although the architecture is designed to support dVRK, there are several concepts and lessons that can be generalized to other robots, including single robot systems:

1. The design pattern for sharing single resources such as a fieldbus and/or a computation thread among multiple robots while keeping them as independent entities, as summarized in Figure 3.2.

2. The separation of I/O and robot-specific control via the Hardware Interface Layer and the component-based design, and the use of a synchronous execution model (*ExecIn*/*ExecOut*) for efficiency. This can be generalized to other robots, including single robot systems. For example, in Laboratory for Computational Sensing and Robotics (LCSR), there are several robots that use Galil controllers (Galil Motion Control, Rocklin, CA), but they all have different C++ components that interface via Ethernet/PCI to the Galil controllers. Applying this concept, they can share a common *mtsGalilControllerIO* component that handles the I/O and then synchronously invoke robot-specific computations via the *ExecIn*/*ExecOut* interfaces.

This software stack has been used extensively among the dVRK community and some researchers have made contributions, including a ROS physics simulation and a MATLAB Simulink®to C++ interface [76]. We are also aware of an ongoing effort for Real-Time ROS (RTROS) [14], using the Ach library [22]. In this case, it would be possible to replace *cisst* component-based communication with Real-Time ROS (RTROS). The key benefit of this approach is that it would provide an identical ROS Application Programming Interface (API) while still meeting hard real-time constraints and providing sufficient performance (average 45.96 $\mu s$ for publisher/subscriber [14]). Therefore, researchers who are already familiar with the ROS API can easily modify the low-level components without learning *cisst*. 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.

# Chapter 4

# Application to Virtual Fixture Assisted Suturing

This chapter presents an application of the high-performance architecture to semi-autonomous teleoperation; in particular, a suturing task in Robotic Minimally Invasive Surgery (RMIS). This includes research contributions in the development of virtual fixtures for the needle passing and knot tying sub-tasks, with a multi-user study to verify their effectiveness.

## 4.1 Introduction

MIS is beneficial to patients due to the smaller incisions and faster recovery times. However, surgeons face the challenge of a limited and constrained workspace with loss of direct visualization. The development of robot-assisted surgery via the da Vinci surgical system [37] has addressed these challenges by providing the surgeon with wristed instruments, augmented stereo vision and better ergonomics. Even then, the dexterous manipulation involved in suturing and knot tying is challenging [44], making them mentally demanding and time consuming tasks in MIS. As a result, these skills are an important component of the training program of Fundamentals of Laparoscopic Surgery [67] and similar robot-assisted surgery training programs.

Some researchers have attempted to automate part of this challenging task using learning by demonstration algorithms. Mayer et al. [62] used a supervised learning algorithm on recorded trajectories from an experienced surgeon and generated a semi-automated procedure that can be “played back” by the robot at a later time, thus allowing automatic task completion. Similarly, Schulman et al. [82] used a trajectory transfer algorithm by performing a non-rigid registration between a demonstration trajectory, generated by a human, and a test scenario. A slightly different approach is to define the task analytically. Jackson and Cavusoglu [38] split the suturing task into five steps: Needle Approach, Needle Bite, Needle Reorientation, Needle Regrasping and Needle Follow Through, providing a path planning algorithm for each step. In a subsequent paper, Chow et al. [21] presented a vision guided automatic knot tying system, where the robot automatically ties a knot at a user selected position.

Although the fully autonomous and semi-autonomous approaches are promising, they will continue to be a challenge for the foreseeable future due to technical difficulties and regulatory concerns. Specifically, the suturing task in these works has been oversimplified and will require additional work to transfer to a real surgical setting. We have taken a surgeon-in-the-loop approach, where “virtual fixtures” (e.g., [74, 45, 47, 97]) are used to reduce uncertainty in motion, thereby improving operation accuracy and reducing mental stress on the surgeon.

Kapoor et al. [45] presented a constrained optimizer framework to define virtual fixtures for a suturing task (catering to the needle alignment sub-step and bite sub-step) with a cooperative robot. Their experimental evaluation was limited to measuring the deviation of the performed trajectory from the ideal path. Later, the authors [47] applied the framework to a knot positioning task (the last step of the knot tying task), which requires a multi-robot cooperation. Virtual fixtures were used to maintain certain spatial relationships between the two cooperative robots. In a follow-up paper, Xia et al. [97] extended this approach to a master/slave teleoperation robot (da Vinci Surgical Robot) with the same knot positioning task. However, the authors only presented an experimental protocol without any experimental data proving its effectiveness.

In this chapter, we implemented and validated virtual fixtures to assist the user during the needle passing and knot tying sub-tasks on a teleoperated robotic system. Section 4.2 gives an overview of contributions. In Section 4.3, we present a new approach to define virtual fixtures in the task frame, along with an explanation of the needle passing and knot tying virtual fixtures. Section 4.4 describes the system implementation details and the validation experiment and user study. The outcomes of the study and a discussion of the experimental findings is provided in Section 4.5. Finally, we conclude this chapter by discussing possible future research directions in Section 4.6.

## 4.2 Thesis Contributions

This application contains several research contributions. First, it introduced a novel multi-level haptic rendering mechanism featuring a low-level fast updating haptic rendering based on simple models, and an easy to program interface for high level behaviors. Another contribution is the design of two virtual fixtures: one that guides a circular motion for the needle passing task and another plane virtual fixture that constrains motion onto the plane for the knot tying task. The last contribution is the experimental evaluation of the two virtual fixtures via a multi-user study. This work was developed in collaboration with several individuals. Anton Deguet and Preetham Chalasani assisted with software development, Dr. Anand Malpani helped in evaluating and selecting virtual fixtures and Dr. S. Swaroop Vedula helped on user study design and data analysis. Much of the content of this chapter was published in [20].

## 4.3 Virtual Fixtures

Virtual Fixtures are used to augment sensory information by constraining or guiding motion in order to improve performance of a surgeon in direct or remotely manipulated tasks. In this paper, we implement a couple of virtual fixtures on the master side manipulators to enhance the accuracy and efficiency of the surgeon performing a suturing task. The da Vinci surgical system is highly optimized to follow motions of the master manipulator. Thus, we chose to implement these fixtures on the master side of the robot. Constraining the surgeon's hand motion by master-side VFs does not alter the basic position-based telemanipulation control loop of the robot and therefore does not interfere with the safety and system assurance associated with the basic telemanipulation function.

In the following subsections, we will discuss the implementations of the VF in detail – by describing the suturing task (Section 4.3.1), by providing the generic formulation of an impedance type VF (Section 4.3.2), and, by formulating the subtask specific VF for the needle passing and knot tying steps (Sections 4.3.3 and 4.3.4, respectively).

### 4.3.1 Task Description and Analysis

Suturing is an important step in multiple surgical procedures and an integral component of surgical skills training curricula. It is also considered difficult to master in terms of dexterity and time consuming tasks in MIS [77]. Although specialized instruments like Endo Stitch$^{TM}$ (Covidien, Medtronic), are used in traditional MIS to reduce operating time, the conventional suturing and knot tying technique remains the most popular and cost effective method [66]. The suturing step often requires multiple attempts by the surgeon, which extends the operating time [77]. We argue that providing virtual fixture assistance will increase the accuracy in task performance, and significantly decrease the time per stitch. This technique can be used to train novice surgeons.

### 4.3.2 Impedance Virtual Fixture

We have used an impedance-type VF, wherein forces are exerted on the surgeon's hands to provide guidance. In our experience, these forces do not interfere significantly with basic control stability, and the slave manipulator simply follows the master motions. Furthermore, we can easily combine different assistance behaviors like gravity compensation with an impedance VF by simply adding the desired joint torques that were computed for each case, as shown in Figure 4.1.

To define the VF controller behavior, the teleoperation component sets a force/-torque compliance frame $F_c = [R_c, \vec{p}_c]$ defined in the master base frame, together with position stiffness gains $\vec{k}^{(+)}, \vec{k}^{(-)}$, position damping gains $\vec{b}^{(+)}, \vec{b}^{(-)}$ and force bias terms $\vec{g}^{(+)}, \vec{g}^{(-)}$. Similarly, we have the orientation torque bias terms $\vec{\tau}^{(+)}, \vec{\tau}^{(-)}$ and orientation stiffness factors $\vec{k}_o^{(+)}, \vec{k}_o^{(-)}$. In the above, the $^{(+)}$ and $^{(-)}$ superscripts

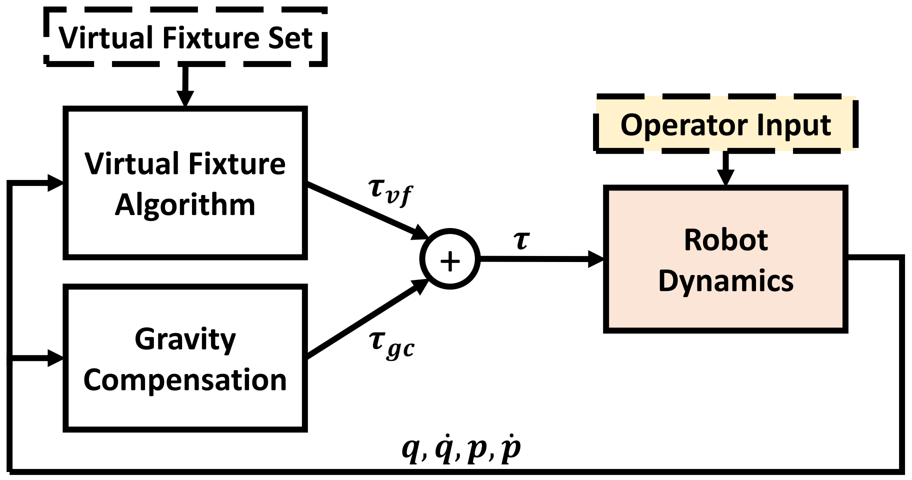


**Figure 4.1:** Impedance Type Controller :
$\mathbf{q}$ - joint position; $\dot{\mathbf{q}}$ - joint velocity; $\mathbf{p}$ - Cartesian position; $\dot{\mathbf{p}}$ - Cartesian velocity; $\tau$ - total joint torque applied to robot; $\tau_{vf}$ - joint torque from virtual fixture controller; $\tau_{gc}$ - joint torque from gravity compensation.

denote whether the parameter applies to motions/forces in the positive or negative directions, respectively. Desired forces and torques applied on the master tip are represented by $\vec{f}$ and $\vec{t}$, respectively. Given the current velocity $\dot{p}$, Algorithm 1 presents the pseudo-code for computing the desired force and torque that should be applied on the master tip.

One advantage of our design for VF is that it permits very fast haptic rendering of discontinuous impedance environments in the local configuration space near the slave end effector, also allowing a very versatile description of local VF behavior. Further, it permits simple combinations of VF elements. Although the low-level VFs directly supported by this formulation are very simple, more complex VFs may be implemented by updating the parameters at a reasonably fast update rate that is nevertheless slower than the very fast rates associated with haptic rendering. In our current implementation, we use a haptic update rate of 1 KHz and a VF update rate of 500 Hz.

**Algorithm 1** **Impedance Virtual Fixture**

```
1: if Enabled then
2:     # —— Force ——
3:     $\vec{q} = F_c^{-1}\vec{p} = R_c^{-1}(\vec{p} - \vec{p_c})$          ▷ Position Error
4:     $\vec{v} = R_c^{-1}\dot{p}$          ▷ Velocity in compliance frame
5:     for $i \in \{x, y, z\}$ do
6:         if $\vec{q_i} \leq 0$ then
7:             $\vec{g_i} = \vec{g}_i^{(-)} + \vec{k}_i^{(-)}\vec{q_i} + \vec{b}_i^{(-)}\vec{v_i}$
8:         else
9:             $\vec{g_i} = \vec{g}_i^{(+)} + \vec{k}_i^{(+)}\vec{q_i} + \vec{b}_i^{(+)}\vec{v_i}$
10:        end if
11:    end for
12:    $\vec{f} = R_c\vec{g}$          ▷ Desired force
13:
14:    # —— Torque ——
15:    $\triangle R = R_c^{-1}R$
16:    Compute $\vec{\theta}$ such that $exp(skew(\vec{\theta})) = \triangle R$
17:    for $i \in \{x, y, z\}$ do
18:        if $\vec{\theta}_i \leq 0$ then
19:            $\vec{\tau_i} = \vec{\tau}_i^{(-)} + \vec{k}_{oi}^{(-)}\vec{\theta}_i$
20:        else
21:            $\vec{\tau_i} = \vec{\tau}_i^{(+)} + \vec{k}_{oi}^{(+)}\vec{\theta}_i$
22:        end if
23:    end for
24:    $\vec{t} = R_c\vec{\tau}$          ▷ Desired torque
25: end if
```

An example of a one-sided forbidden region plane virtual fixture is shown in Figure 4.2. A 3D plane can be defined by a point $p_{plane}$ and a vector $N_{plane}$, normal to the plane. The force/torque frame is defined with its origin at $p_{plane}$ and its Z axis as $N_{plane}$. The X and Y axes can be chosen freely as long as they form a right-handed coordinate frame. For example, if the positive and negative force stiffness gains are set to $[0, 0, 0]$ and $[0, 0, 500]$ respectively, the user is free to move on one side of the plane while feeling a force pushing him/her away from the forbidden side of the plane.

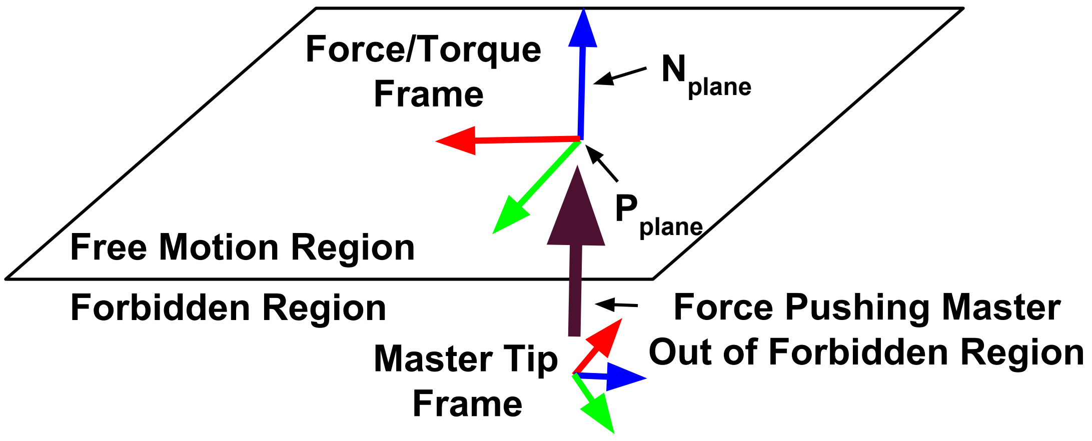


**Figure 4.2:** Plane forbidden region virtual fixture

## 4.3.3 Needle Passing Virtual Fixture

In the needle passing sub-task, we provide a three-phase virtual fixture: (i) to bring the needle to a desired orientation, (ii) to guide the user to the entry point, and (iii) to guide the user to pierce through the tissue in a constrained circular motion. As the desired rotation and position are known to the system, transitions between the three virtual fixture phases are automatic based on rotation and position errors without any additional user interaction via a foot pedal, for example.

In this study, we used a 3D-printed needle holder (Figure 4.6) to ensure that the needle was held consistently in a known relative pose across a user's trials (task repetitions). Although the use of a needle holder prevents the needle from piercing through the tissue, and thus makes it impossible to complete a suture, this approach allows us to factor out the effect of variability in the environment on the task performance and focus our analysis on evaluating the design and effectiveness of the virtual fixture. The needle tip frame can be computed using robot forward kinematics and the known needle holder design. Optionally, computer vision techniques can be used to determine the relative pose of the needle with respect to the robot tool tip frame; however, that is outside the scope of this work.

The user starts the virtual fixture by pressing the `CAMERA` pedal. Once the foot pedal event is detected, a force/torque compliance frame is defined at the current gripper position with a precomputed orientation based on the ideal entry pose for the needle. Positive gain values with zero offset are set for all three axes for the force and torque frames. This virtual fixture will apply a force and torque on the master manipulator to guide the user to orient the needle. The system constantly monitors the orientation error and moves on to the second phase if the error is within a set threshold value – determined empirically to ensure accuracy but at the same time allow smooth phase transition. In the second phase, the position of the force/torque compliance frame is changed to the desired position, while orientation and gain values remain unchanged.

After the needle is in the correct pose, the third phase virtual fixture is enabled to guide the user to pierce through the tissue. We define a circle with its origin at the center of the needle and the radius same as the needle radius to constrain the

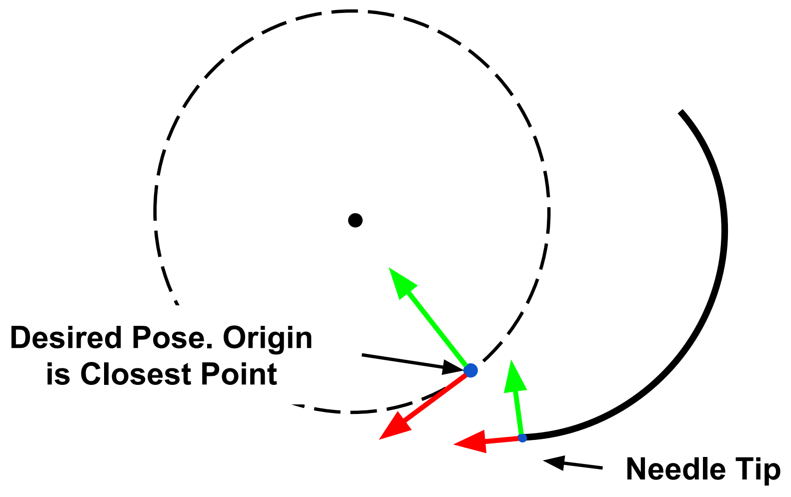


**Figure 4.3:** Needle passing circular motion virtual fixture.

needle driving motion. The virtual fixture frame is updated every single loop by setting the origin as the closest point on the desired circle (see Figure 4.3), the Y axis points towards the center of the circle, and the Z axis is normal to the plane of the circle. The virtual fixture position gains along the Y and Z axes are set to large values to maintain the needle along the circle, while the X axis gain is set to zero to allow the user to freely move along the needle and the circle. Orientation gains are set to large values for the three axes, enforcing the needle tip to move along the predefined circular path. With these settings, the needle motion is constrained along the circle with correct orientation. The gain values were determined empirically – strong enough to provide effective guidance, but soft enough so that the user can over-power it when necessary to compensate for small modeling errors.

### 4.3.4 Knot Tying Virtual Fixture

The surgeon's knot has three elements: a two loop knot followed by two single loop knots. Here we only focus on defining the virtual fixture for the two loop knot. A similar technique can easily be applied to the other single loop knots. One of the challenges faced by a novice user in knot tying is to successfully loop the thread twice around the instrument tip without slippage.

We use a plane virtual fixture (Figure 4.4) to provide assistance to the novice user during the looping action. The plane virtual fixture is located at the clevis point of the instrument in the non-dominant hand (the one on which the loop will be laid upon) with a normal along the instrument pointing direction. This is a constraining two-handed virtual fixture that allows the dominant instrument tip to move freely in the plane defined by the virtual fixture, and pulls it back towards the plane if there is any out of plane motion. A user can enable this virtual fixture by pressing and holding the `CAMERA MINUS` foot pedal. The virtual fixture gets activated only when the dominant instrument tip is close to the virtual fixture plane, in order to avoid a sudden large force from being applied to the user and resulting in an undesirable motion. The Z axis for the virtual fixture is aligned with the instrument pointing direction. The gain values are set to be large along the +Z and -Z axes, and zero on the X and Y axes.

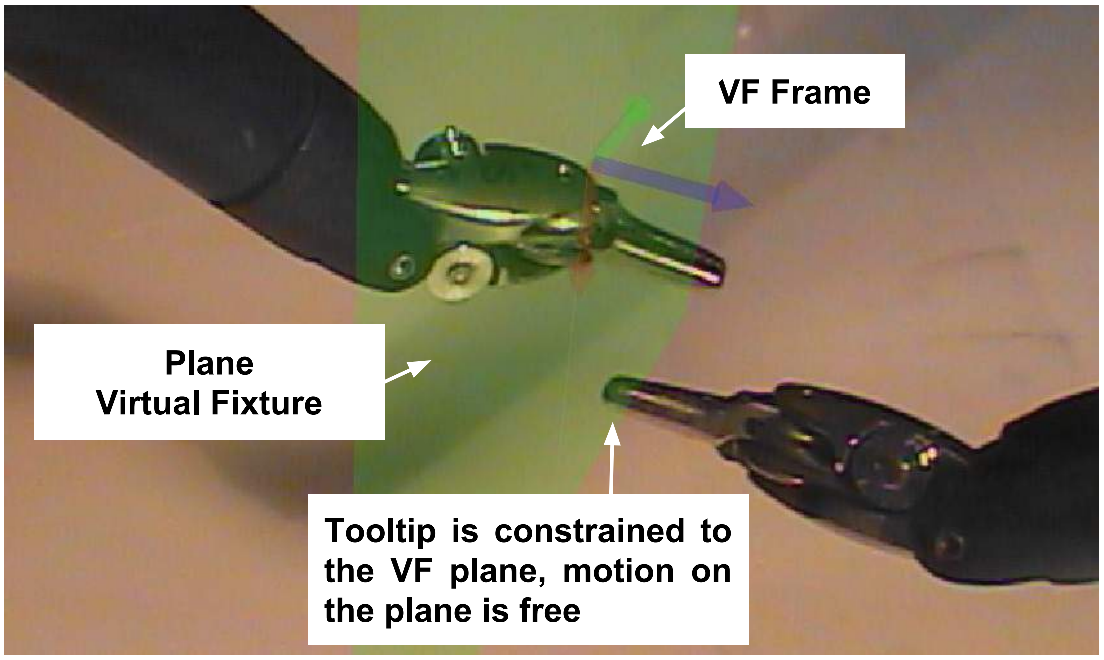

VF Frame
Plane Virtual Fixture
Tooltip is constrained to the VF plane, motion on the plane is free

**Figure 4.4:** Knot tying virtual fixture: a constraining plane (green) with its force/-torque virtual fixture frame. It is shown here for demonstration purposes. Users can feel it haptically, but do not see the augmented plane visually.

## 4.4 Experiment

We conducted a user study to evaluate the effects of virtual fixture assistance in teleoperated robotic suturing tasks of needle passing and knot tying. The implementation of the system architecture and the execution of the user study, along with the collected data and analyses are described in detail in the following sections.

### 4.4.1 System Implementation on da Vinci Research Kit

We have implemented the described virtual fixture framework on the hardware and software architecture proposed in this thesis. As shown in Figure 1.1, the setup includes two MTMs, two PSMs, a Foot Pedal Tray, and a High-Resolution Stereo Viewer (upgraded to two flat panel displays configured to 1024 $\times$ 768 resolution, as compared to the standard dVRK systems that use CRT displays with 640 $\times$ 480 resolution). The different pedals on the foot tray can be pressed by the operators to trigger different events that change the control system states. Our stereo vision system comprises of two SONY lipstick cameras, which are connected to the dVRK stereo viewer console. A stereo camera calibration procedure is performed to obtain the intrinsic/extrinsic parameters for the camera that are later used for image rectification and camera registration. We note that although the stereo baseline and lens-to-tissue distances may be different if a regular da Vinci stereo endoscope is used, the stereo visualization provided is similar enough to allow us to evaluate our virtual fixture strategy.

The robot control is based on the software architecture proposed in chapter 3, and the application specific high-level software is programmed in C++ and interfaces to the robot via the ROS interface as summarized in Figure 4.5. The Task Node contains the task-specific algorithm that defines and publishes virtual fixture commands to

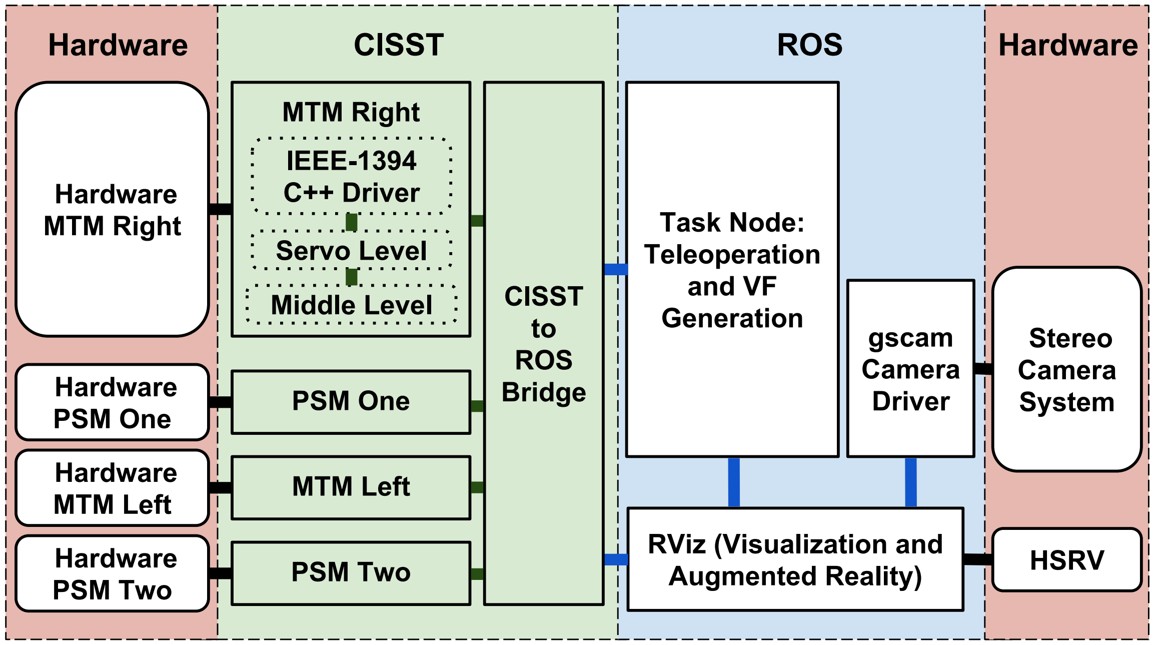


**Figure 4.5:** Block diagram showing hardware/software connection, software components implemented in both *cisst* and ROS environments

the masters based on task state. The RViz node is a ROS visualization software for visualizing robot configuration, displaying the live stereo video stream (grabbed with the *gscam* package [39]) and adding augmented reality markers for visual guidance.

## 4.4.2 Needle Passing Sub-task

As shown in Figure 4.6, a 15 mm thick tissue phantom is used in the experiment. This phantom has a stiff top layer simulating the epidermis and a soft dense foam layer simulating the dermis. A large needle driver instrument from Intuitive Surgical Inc. is chosen to operate a 3/8 circle 26 mm reverse cutting suture needle fixed in a 3D printed needle holder. The needle holder is designed to tightly fit the large needle driver gripper to ensure that the needle cannot move with respect to the instrument during the test. On the tissue phantom, the target entry and exit points are marked with dark color dots such that they can be easily identified through the stereo viewer on the master console. The operating area is placed at the center of the camera view. Before each trial, the instrument is moved to the same starting pose.

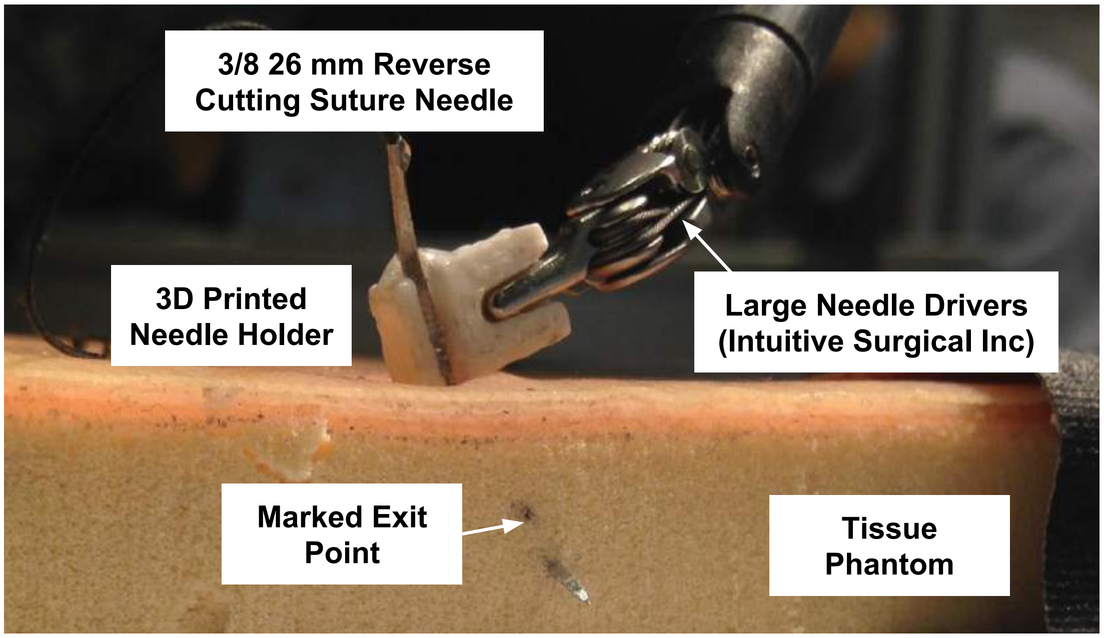

3/8 26 mm Reverse Cutting Suture Needle
3D Printed Needle Holder
Large Needle Drivers (Intuitive Surgical Inc)
Marked Exit Point
Tissue Phantom

**Figure 4.6:** Test setup for needle passing task, including tissue phantom, suturing needle with needle holder and a large needle driver instrument

### 4.4.3 Knot Tying Sub-task

Similar to the needle driving task, two large needle driver instruments (Intuitive Surgical Inc.) are installed on the patient side manipulators. Users teleoperate, with position scale of 0.3, the two instruments from the master console with visual feedback through the stereo viewer. Before the task, a suture thread of 18 cm total length was

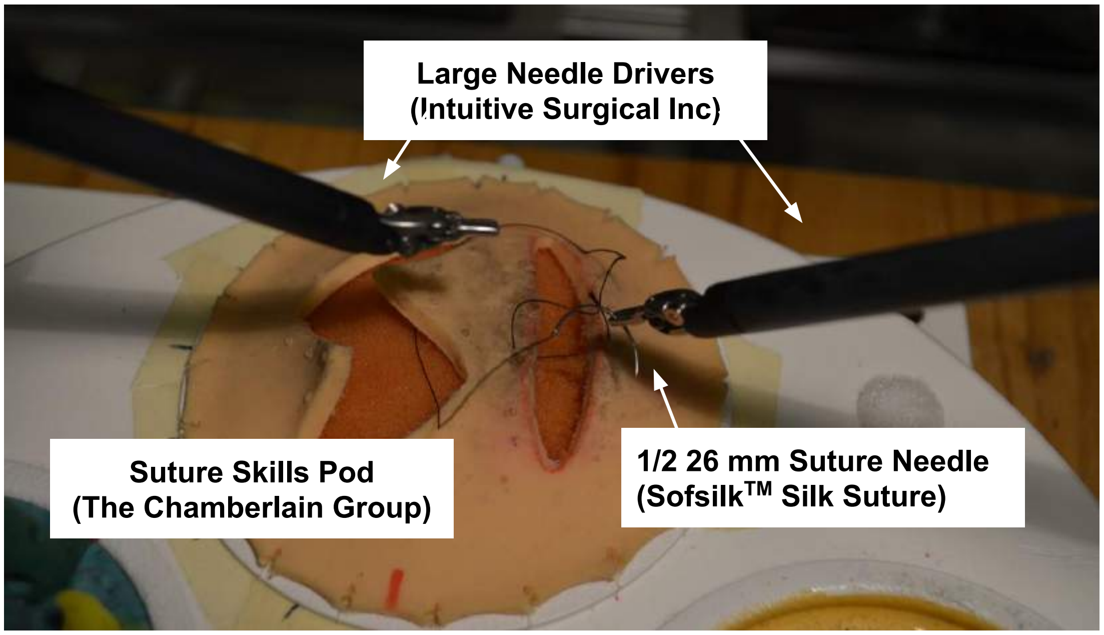


**Figure 4.7:** Test setup including a suture skills pod (phantom), a suture needle and two large needle driver instruments

prepared on a suture skills training pod (The Chamberlain Group Inc.) commonly used for robotic surgery training. As shown in Figure 4.7, the suture has a 3 cm tail left on the right side of the tissue for easy grasping, 2 cm between the entry and exit points on the phantom, and a 13 cm loose end for the knot tying operation. Again, all the robot arms are configured to the same starting poses.

### 4.4.4 Test Procedure

The user study was performed with volunteer users recruited from a population of graduate and undergraduate students at the Johns Hopkins University (JHU) and with none to little experience in teleoperated robotic suturing. Tests were approved by the JHU Homewood Institutional Review Board (HIRB00002925). A total of 14 participants, 13 right-handed and 1 ambidextrous, completed the trials (12 male, 2 female). None of the volunteers have neurological disorders, or uncorrected vision problems that may negatively affect performance.

The experiment was divided into four parts: an introduction section, needle passing sub-task section, knot tying sub-task section and the subjective evaluation section. Users started the experiment with an introduction to the suturing task by watching a video of simulated surgical suturing using the *da Vinci* skills simulator and a brief introduction with hands on time to become familiar with basic da Vinci operation such as teleoperation and use of the clutch pedal. Before each sub-task, users were given sub-task specific instructions and guidance on how to use the sub-task specific virtual fixture before the trials. The subjects then practiced two non-recorded trials, with one in each control mode (freehand and virtual fixture assisted), to understand the system and the sub-task. After this, users performed 4 consecutive trials in each control mode. The order in which the test conditions were performed was balanced and randomized between users to cancel the learning effect. Users took a break between the two sub-tasks. Users completed a NASA TLX survey [33] after each sub-task and a subjective evaluation questionnaire at the end of the entire trial. The whole experiment lasted about 1.5 hours per user. None of the users verbalized fatigue as corroborated by the self-reported TLX survey.

### 4.4.5 Data Collection and Analysis

The states of the MTM and PSM robots (500 Hz), video stream (30 Hz) of the stereo cameras and foot pedals events were logged for all trials. For the needle passing task, exit points were also recorded with high-definition cameras, from which the exit error was measured. The performance of the needle passing task was evaluated in terms of exit error and task completion time. For the knot tying task, task completion time, total needle trajectory and number of times the suture slipped during the loop were used as performance evaluation metrics. For subjective evaluation, the standard NASA TLX survey is adopted to evaluate operator workload and the subjective evaluation questionnaire covers perceived difficulties during the suturing task, user preference on control mode and suggestions on how to improve robotic assistance.

## 4.5 Results and Discussion

This section reports and discusses the results of the user study, including both sub-tasks. There were multiple trials for each test condition, so a two-way repeated measure analysis of variance (ANOVA) is performed, where the test mode (freehand and virtual fixture-assisted) is the first independent variable and the user is treated as a random variable. For TLX workload analysis, a paired t-test is used.

### 4.5.1 Needle Passing Sub-task

#### 4.5.1.1 Statistical Analysis

We analyzed total task completion time from the start of needle motion until the needle tip pierces through tissue. Figure 4.8(a) shows a boxplot of the total task completion time. The overall mean total task completion time was 18.17 seconds. The introduction of virtual fixture assistance reduced the mean task completion time by 15% from 19.67 s to 16.67 s. There was a significant effect of virtual fixture assistance ( $F_{1,84} = 5.62$, $p = 0.02$) and user ( $F_{13,84} = 6.33$, $p < 0.01$) on total task completion time. The interaction effect between virtual fixture assistance and user was also significant ($F_{13,84} = 3.23$, $p < 0.01$).

The measured exit error is defined as the distance between the marked target point and the needle exit point. Figure 4.8(b) shows a boxplot of the measured exit error. The mean exit error when virtual fixture assistance is enabled was 0.88 mm, while the mean error in freehand mode was 3.38 mm. The effect was significant ( $F_{1,84} = 155.36$, $p < 0.01$ ). The effect of users was also significant ($F_{13,84} = 2.63$, $p < 0.01$), but there was no significant interaction effect ($F_{13,84} = 1.05$, $p = 0.41$).

#### 4.5.1.2 Trajectory Analysis

In comparing freehand to virtual fixture modes, we noticed that in freehand mode, users tend to first move down in a straight line with the intent to pierce the needle

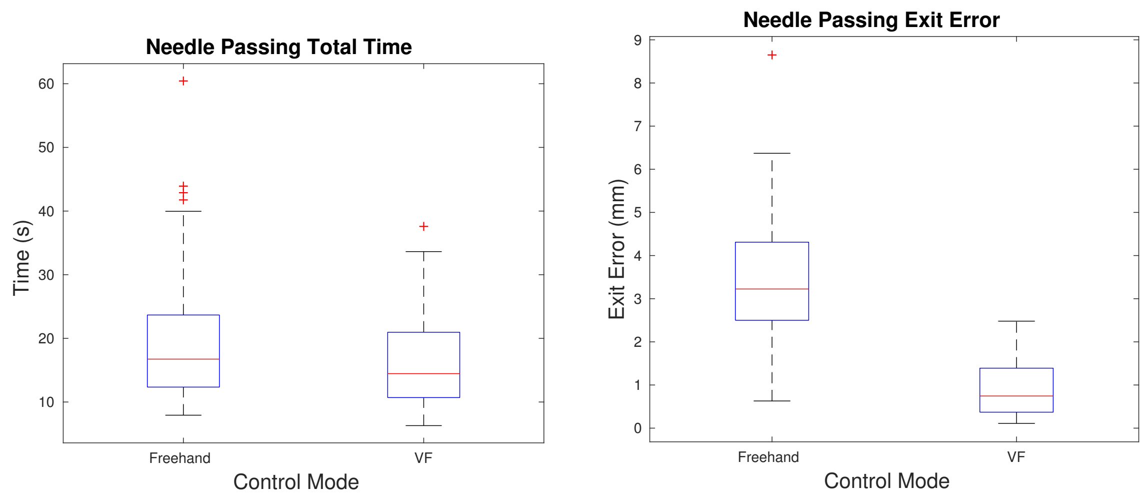
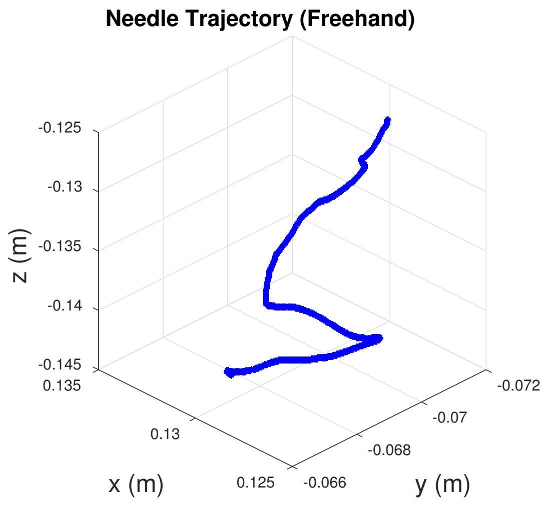


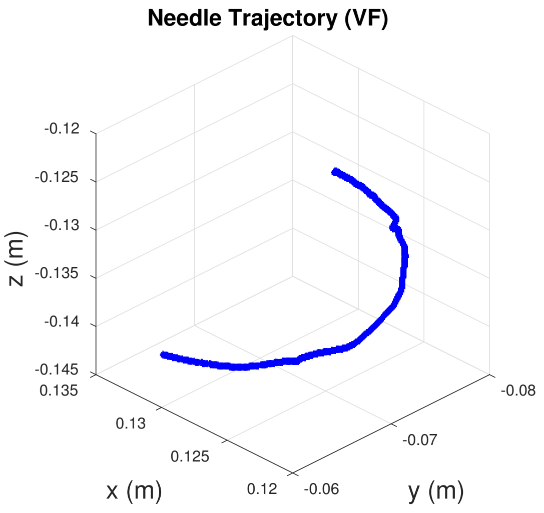
**Figure 4.8:** Boxplots showing the needle passing task completion time under two test conditions (left) and needle exit point error (right). For all boxplots in this chapter, the red centerline represents the median, the upper and lower edge of each box corresponds to 25th and 75th percentiles, and the whiskers extend to the most extreme data points. The red cross points are considered outliers [28].

through the tissue before they even start rotating the needle. Figure 4.9(a) illustrates this tendency. With virtual fixture assistance, the user moves and rotates the master simultaneously, resulting in a smooth circular motion as shown in Figure 4.9(b), which can effectively reduce stress on the tissue.

Another important finding is that, in freehand mode, users tend to re-adjust the needle trajectory when they find that the needle exit point might be far away (e.g., 5 mm) from the target exit point by pulling the needle out vertically and redoing the operation, which could potentially increase the possibility of tearing tissues and explain the longer total task completion time.



(a) Freehand



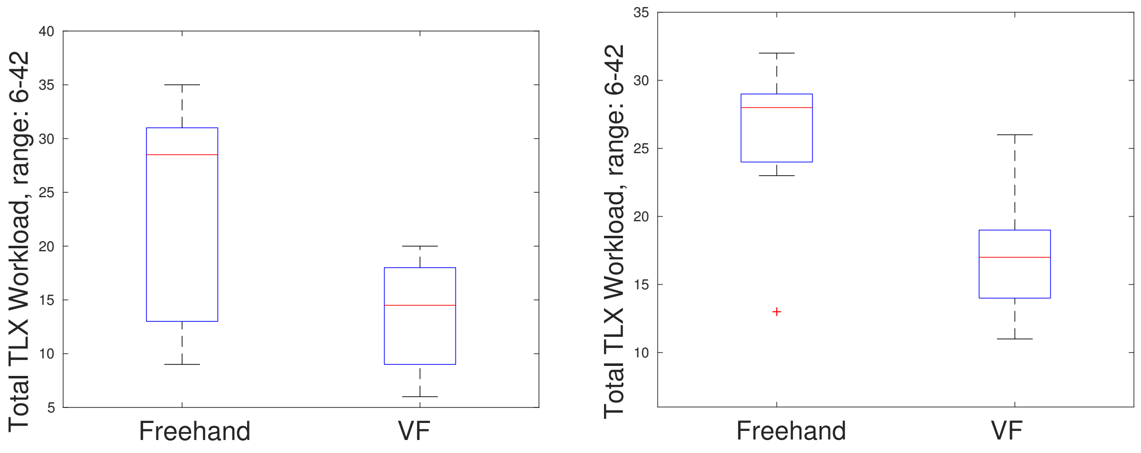
(b) Virtual fixture assisted

**Figure 4.9:** Comparison of needle passing trajectories: left is needle trajectory in freehand motion, right is trajectory from the same user with virtual fixture assistance.

#### 4.5.1.3 Operator Workload

The boxplot shown in Figure 4.10(a) summarizes the overall workload (the sum of the responses to all categories). The mean workload for freehand mode was 23.86 compared with 13.57 for virtual fixture assistance (the scale ranges from 6 to 42, with lower numbers indicating lower workload) and a paired t-test reveals that the beneficial effect of virtual fixture assistance is significant (p = 0.0005).

The radar plot in Figure 4.11 shows the mean values of each workload category self-reported by the users in the NASA TLX survey. Virtual fixture assisted needle passing resulted in less workload than the freehand mode in all categories.



(a) Needle Passing (b) Knot Tying

**Figure 4.10:** Boxplot showing the total operator workload as self-reported via the NASA TLX survey for each test mode. Left is from the needle passing sub-task and right is from the knot tying sub-task.

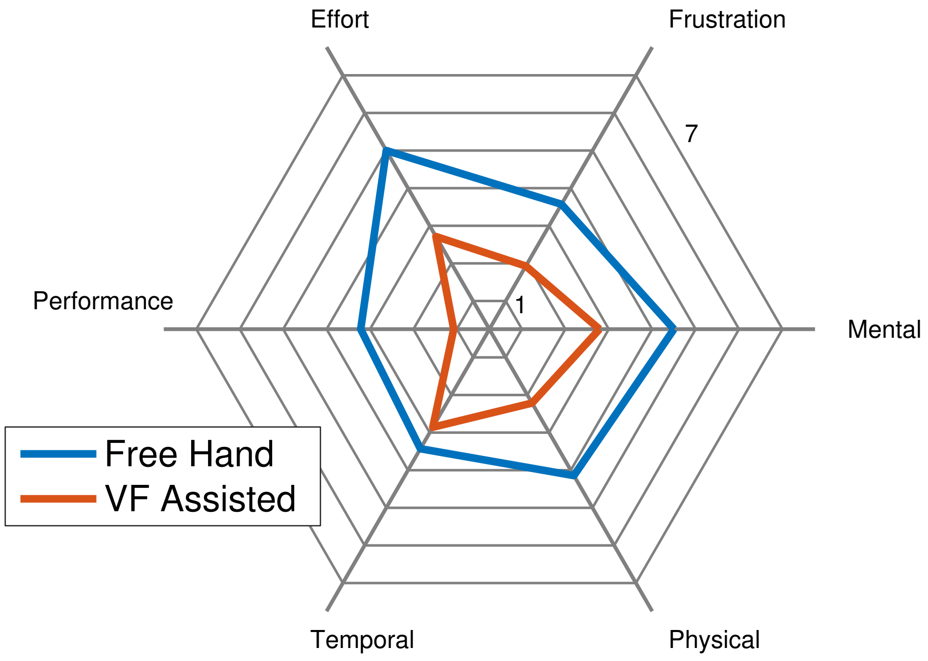


**Figure 4.11:** Needle passing task, NASA TLX survey radar plot of average categorical workload as self-reported by the users. Workloads increase from the center.

#### 4.5.1.4 Subjective Evaluation

Participants unanimously opted for virtual fixture-assisted mode rather than freehand mode for the needle passing task. The ability to find the right entry orientation was the favorite feature. One user also suggested that a text hint on when to start and stop virtual fixture assistance can be helpful.

### 4.5.2 Knot Tying Sub-task

#### 4.5.2.1 Errors (Number of Slips)

The motivation behind the plane virtual fixture is to prevent the slip events common for novice users. Our findings show that the plane virtual fixture does help reduce the average number of slips per trial from 1.5 in freehand mode to 0.34. The effect of virtual fixture assistance is significant ($F_{1,84} = 28.87$, $p < 0.01$). The data do not provide statistically significant evidence that the overall skill level of the "novice" participants was the same ($F_{13,84} = 1.47$, $p = 0.15$) or that the amount of improvement was uniform across users ($F_{13,84} = 1.17$, $p = 0.31$). Therefore, it is possible that different users will benefit from virtual fixture assistance by different amounts. It needs to be pointed out that not all slips happened during the suture wrapping process. In fact, a few slips happened when the user had already finished the loop and was trying to reach and grab the suture tail, but failed due to lack of good depth perception with the pair of small video cameras that we used. This issue can be alleviated by providing a better stereo visualization system.

#### 4.5.2.2 Task completion and trajectory length

Noticeable benefits in task completion time were observed when using virtual fixture assistance ($F_{1,84} = 16.15$, $p < 0.01$). Similarly, a statistically significant difference in PSMs trajectory length (both of PSM1 and PSM2 trajectory) was also found ($F_{1,84} = 11.35$, $p < 0.01$ for PSM1, $F_{1,84} = 12.39$, $p < 0.01$ for PSM2). Figure 4.12 shows boxplots of task completion time versus trial number from two different test orders. Users who did freehand mode first demonstrated a clear learning curve. Especially for the first trial, the mean completion time was 67.62 seconds for the freehand user compared with 49.71 seconds for the virtual fixture-assisted user. Although the difference is not statistically significant, it still suggests that virtual fixture assistance can potentially help novice users complete the task in a more timely manner.

#### 4.5.2.3 Operator Workload

A summary of the overall workload (the sum of the responses to all the categories in the TLX survey) is shown in Figure 4.10(b). The overall mean operator workload is 26.57 for freehand mode and 16.93 for virtual fixture-assisted mode. The effect of assistance was significant (p = 0.0001054). Figure 4.13 shows a radar plot of the mean ratings for each workload category. The radar plot shows that the virtual

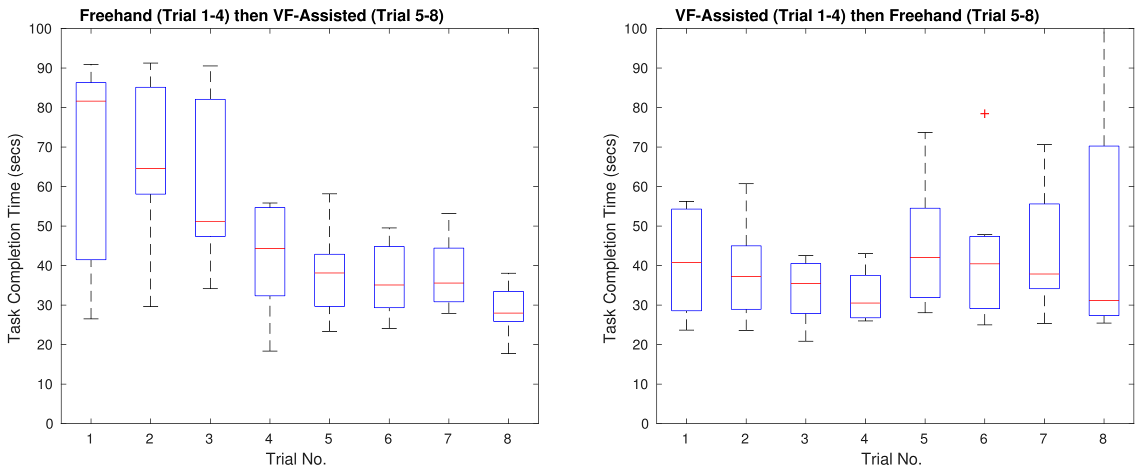


**Figure 4.12:** Boxplots of knot tying task completion time versus trial number. The left figure is from users who did freehand mode first followed by virtual fixture-assisted mode and the right figure from users with the opposite order.

fixture assisted mode has a lower perceived workload in every one of the five categories. Also, compared with the needle passing sub-task, the knot tying task is a much more challenging task physically and mentally and requires more effort to finish.

#### 4.5.2.4 Subjective Evaluation

In the questionnaire, 12 out of 14 users indicated that they prefer virtual fixture assistance to freehand mode. One of the two users who preferred freehand mode stated that the reason was that freehand mode felt less constrained. Another user suggested that giving a little more time to get used to the virtual fixture during the pre-evaluation phases of the experimental protocol might have improved performance.

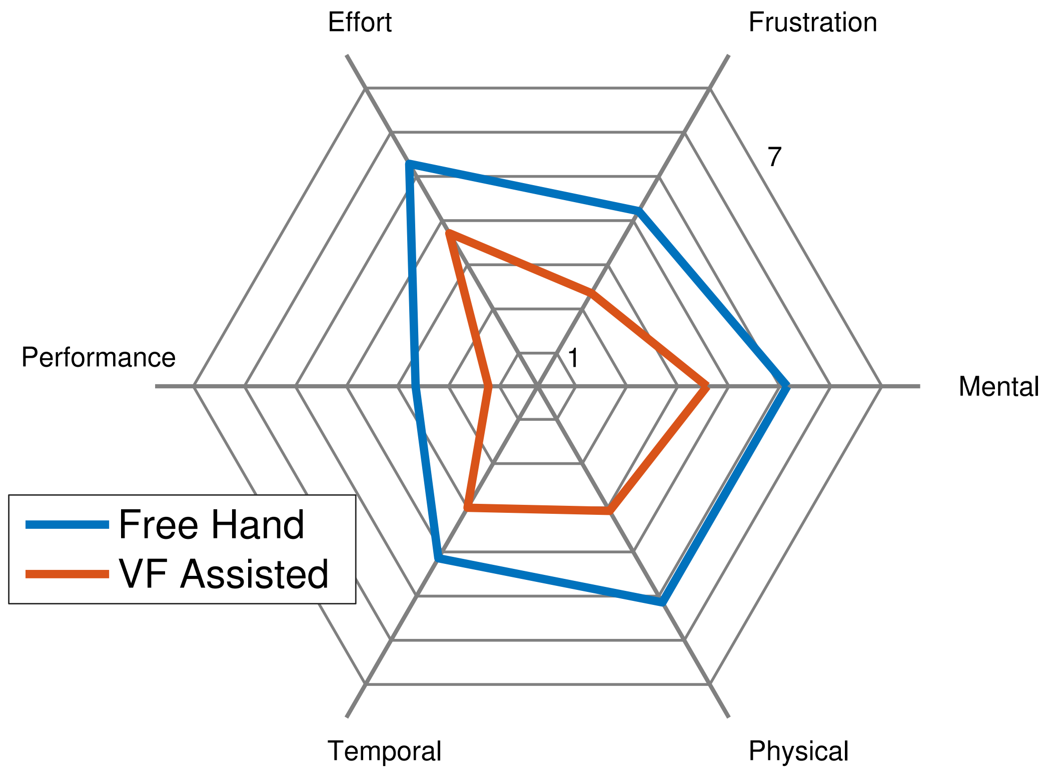


**Figure 4.13:** Knot tying task, NASA TLX survey radar plot of average categorical workload as self-reported by the users. Workloads increase from the center.

## 4.6 Summary and Future Work

This chapter has described the implementation of virtual fixture assistance on a dVRK and reports the results of a user study to compare the performance of virtual fixture assistance and freehand teleoperation in both needle passing and knot tying suturing sub-tasks.

Despite bringing the benefits of small incisions and fast recovery times to the patient, MIS presents a constrained workspace and limited vision feedback for the surgeon. In particular, suturing remains the most demanding and time consuming task even with a teleoperated surgical system. This work addresses these problems by providing impedance type virtual fixtures to assist the surgeon to complete the task in an accurate and efficient manner. These fixtures are applied on the master side without breaking the direct master to slave teleoperation link. In particular, a force compliance frame with force/torque gains and offsets is defined in the master workspace based on the current state of the task. This approach may have significant value in introducing virtual fixture assistance in complex telesurgical systems in which maintaining the integrity of a high bandwidth master-slave control loop is vital.

The results of our user study indicate that virtual fixtures can significantly improve needle exit accuracy, thus reducing tissue tearing pressure. During the knot tying task, virtual fixture assistance reduces average task completion time, total trajectory length and number of slips during the task.

One interesting aspect of knot tying is that it is an inherently *two-handed* task.

Accordingly, we developed a two-handed virtual fixture to prevent the thread from slipping off the gripper during both the thread wrapping and tail grasping phases of tying the knot. The success of this strategy illustrates the potential importance of such two-handed, coordinated virtual fixtures in such tasks, and we plan to investigate these further for other tasks.

For haptic rendering, although the current Algorithm 1 is fast and versatile, the force and torque representations are independent and decoupled; i.e., the force output ($\vec{f}$) is only dependent on the position error $\vec{q}$ and its derivative and the torque output only on the rotation error. This works fine for relatively simple virtual fixtures such as the plane virtual fixture used in the knot tying sub-task, however, it can become a limitation if we want to specify a virtual fixture with a coupled force and torque rendering. One example is to render a haptic assistance cue for driving a screw, which involves both the rotational torque and a pushing force along the screw axis. To overcome this, a coupled representation with full 6 DOF can be used to represent renderings that require simultaneous positional and rotational haptic cues.

Future work includes the development and evaluation of different approaches for the knot tying virtual fixture. In this regard, it is possible to explore the trade-offs involved in different virtual fixture implementations and gains for these tasks. Similarly, we recognize that computer vision techniques can be employed for needle tracking and improving augmented reality performance. Another area of future work is incorporating computer vision methods into our system to identify the needle, thread, and tissue locations to help generate the virtual fixture constraints. Here, it is also possible to use the stereo endoscopic camera available with the “classic” da Vinci system in our laboratory, which runs the same open-source software environment. Another future work item is to investigate the use of the third slave manipulator on the da Vinci system to assist the surgeon, e.g., by assisting with tissue alignment for needle passing or by grabbing the thread end and feeding it to the surgeon during knot tying.

# Chapter 5

# Application to Teleoperated Space Robotics

This chapter presents another application of the architecture towards semi-autonomous teleoperation, which is time-delayed teleoperation of a robotic arm for satellite servicing.

## 5.1 Introduction

With the retirement of the space shuttle program, teleoperation of robotic spacecraft from earth has become the primary option for satellite servicing missions. The challenge, however, is the existence of signal delays that are typically on the order of several seconds. These delays increase the difficulty of teleoperation, especially when the remote robot is in contact with the environment. This is the case for the task we are considering, which is to cut the tape that secures a patch of insulation over the satellite access panel.

Xia et al. [99] have previously proposed a model-based delay-tolerant control approach using virtual fixtures with haptic feedback, hybrid position/force control and environment modeling update, that is robust to registration error. Previous evaluations of this approach, however, either consisted of a single user [99] or compared two system configurations that differed only in a single detail [93]. For example, Vozar et al. [93] evaluated user performance and workload conditions with and without hybrid position/force control, as well as with and without a 4 second round-trip delay. However, model-based haptic feedback was turned off during this study as it would only apply to one test condition. Later, a nonholonomic constraint and a nonholonomic virtual fixture (without haptic feedback) were proposed [92] and evaluated experimentally in a four-user pilot study.

Multi-user experiments with system configurations that differ only in a single feature are crucial for identifying the value of that specific feature, but they do not answer the question whether the assistance system provides a benefit with respect to a baseline case of no assistance. It is our belief that a model-based method may perform poorly unless it contains a critical set of features. Thus, in this chapter, we first develop an integrated system that combines some previously presented and newly developed methods and then experimentally verify this system against a baseline case where no assistance is provided (except for a force safety threshold to prevent damage to the equipment). The previously presented methods include use of a plane virtual fixture on the master manipulator [98], hybrid position/force control on the slave robot [99], a task monitor on the slave robot [56], and predictive display of the cutter on the master side [92]. The new methods include a feature that enables the operator to define and adjust the line virtual fixture, and augmented overlay of the measured force and task monitor output. For the first time, we also consider a delay configuration where the telemetry delay is significantly less than the video delay, which is reflective of realistic earth-to-space communication in some scenarios. The results show that the assistance can significantly reduce task completion time and operator workload and that the task monitor can successfully detect failure events such as bunching of the tape during cutting. The results also indicate that the model-based approach can potentially take advantage of the lower telemetry delays to provide an even greater benefit.

The rest of the chapter begins with a summary of research contributions and then is organized as follows: First, the robot system and technical approach are detailed in Section 5.3. Second, the system implementation, the test setup and test procedure and metrics are described in Section 5.4. Section 5.5 presents experimental results and offers a discussion of the findings. Finally, Section 5.6 summarizes the chapter and discusses limitations and topics for future work.

## 5.2 Thesis Contributions

The research contribution of this application includes the development of a line virtual fixture with augmented reality, a test for different time delay configurations and a multi-user study that shows the advantages of this system. This application utilizes the high performance in low-level control for haptic rendering of virtual fixtures and the high level interface for integration with other components such as augmented reality and the slave robot, and is another demonstration of the effectiveness the proposed architecture.

Credits: Zihan Chen developed the line virtual fixture rendering, along with a user interface for defining and adjusting it, conducted the user study and analyzed data. Several components of the control architecture were proposed by other individuals [98, 99, 56], but implemented (or reimplemented) and integrated by Zihan Chen. This work also uses infrastructure developed by others. Specifically, the satellite cutting test setup was developed by Vozar et al. [94, 93] and the open-source Orocos based controller for the slave robot was developed by Jonathan Bohren [9].

## 5.3 Technical Approach

The model-based approach is summarized in Figure 5.1. The operator teleoperates the remote slave robot with augmented vision feedback and haptic feedback based on a task model. Data exchanges between the ground and remote sides are delayed

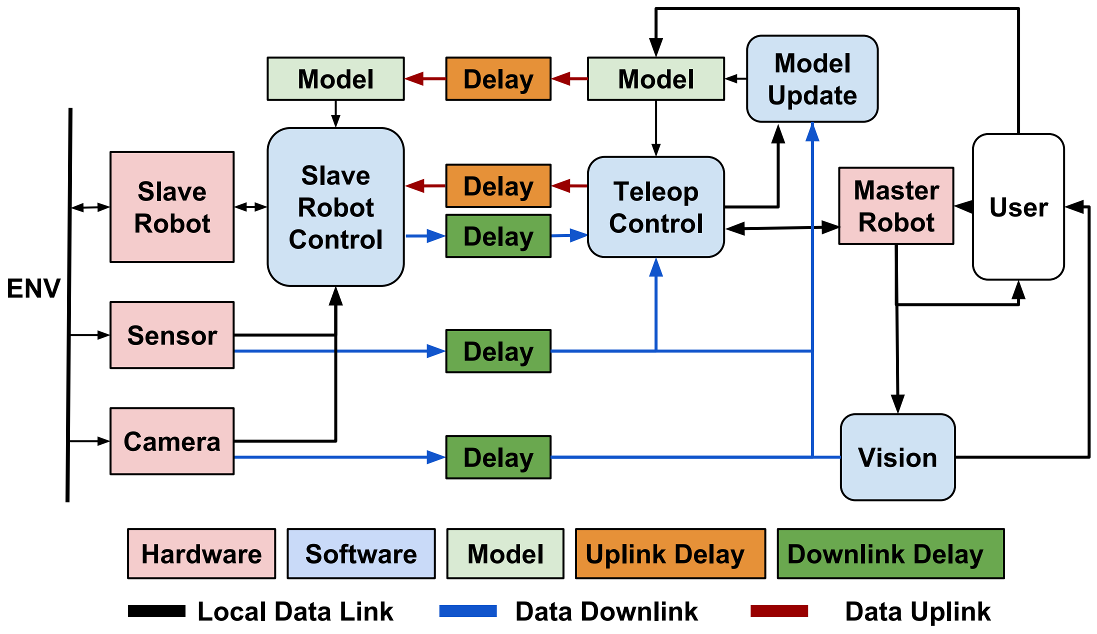


**Figure 5.1:** System Architecture Overview. The operator teleoperates the remote slave robot with augmented vision feedback and haptic feedback based on the task model. Data exchanges between the ground and remote side are delayed and the model is updated on the ground side based on delayed sensory feedback.

and the model is updated on the ground side based on delayed sensory feedback. The rest of the section describes each component of this approach including a plane virtual fixture, hybrid position/force controller, task monitor, line virtual fixture and predictive display in detail.

All of these components were developed in prior work by others. My contribution was to reimplement and/or integrate them into a single application, to develop an augmented overlay for the task monitor output, and to develop a novel method for specifying and adjusting the line virtual fixture.

### 5.3.1 Plane Virtual Fixture

Use of a plane virtual fixture was proposed by Xia et al. [98]. The motivation was that the task space (plate) is a planar surface and automatically aligning the cutter plane with the surface can help users by reducing their workload, thus improving overall performance. Although we assume that a reasonably good initial guess of the plane model is available either through initial registration or through CAD models at the beginning of the task, both orientation and positional errors can exist in the plane model. Methods to compensate for this registration error are presented in the next section.

### 5.3.2 Hybrid Position/Force Controller and Registration Update

In subsequent work, Xia et al. [99] reported using a standard hybrid position/-force controller (HPFC) [72] on the remote robot, where the normal to the plane model defines the direction of force control. Once running, the HPFC is capable of compensating for positional registration error between the real world and the virtual plane.

Plane orientation errors can be corrected by updating the plane normal based on the cutting trajectory, as described by Xiao et al. [57]. Figure 5.1 shows two models in the system, namely, the plane model on the remote side used by HPFC and for cutter orientation alignment and the plane model on the ground used for master side haptic rendering. On the ground, the slave plane model is transformed to the master workspace using

$$N_m = R_s^m \cdot N_s \tag{5.1}$$

$$P_m = P_{mc} + \frac{dist}{gain} \times N_m \tag{5.2}$$

$$dist = |(P_s - P_{sc}) \cdot N_s|, \tag{5.3}$$

where $N_s$, $P_s$ are the plane normal unit vector and plane point in the slave coordinate frame, $N_m$, $P_m$ are the plane normal unit vector and point in the master coordinate frame for haptic rendering, $P_{mc}$ is the current master tip position, $R_s^m$ is the rotation from the slave to the master coordinate frame, $P_{sc}$ is the slave cutter tip position, *dist* is the scalar distance between the cutter and slave plane model, and *gain* is the scalar teleoperation gain from master to slave.

The plane model is updated on the ground side based on sensory data and then transmitted to the remote side. In the case where the plane model is below the real plate, the robot on the slave side hits the plate before the user hits the haptic plane. The slave robot detects the contact by using force data and ignores further commands in the plane normal direction that would move it into the plate. On the ground (master) side, when this situation is detected, the slave side plane model and the master haptic model get updated. In the other case when the plane model is above the physical plane, the user feels the haptic plane first. In all cases, the user turns on the HPFC by pressing into the haptic plane and off by pinching and lifting the master input device at the same time. In the situation that the master CLUTCH pedal is released, the master haptic plane model position is updated to the master current position but the slave plane model is unchanged; this avoids sudden force applied to the master tip. Subsequently, a similar approach was taken by Chalasani et al. [16, 95] in teleoperation of robot instrument for tissue palpation with virtual fixture assistance.

### 5.3.3 Safety and Task Monitoring

As a safety feature, the force/torque measurement on the slave side is monitored locally. When the norm of force vector is higher than a threshold (20N), the slave robot transitions to a safe mode, where only gravity compensation is performed.

Beyond safety monitoring, the force data is also used in a task monitoring component. Kandaswamy et al. [43] developed a model for anomaly detection and identified parameters based on offline experiments. Xiao et al. [56] made this parameter estimation an online process. In this study, we favored the simple fixed parameter model from [43] due to its robustness. The force in the cutting direction was estimated using

$$F_{est} = \mu_k \times |F_n| + F_c, \tag{5.4}$$

where $\mu_k$ is the kinetic friction coefficient and $F_c$ is the nominal cutting force. The parameters used in this study are $\mu_k = 0.48$ and $F_c = 4$ for the tension-based strategy (cutter pulling up against tape seam while cutting) and $\mu_k = 0.56$, $F_c = 4$ for the compression-based strategy (cutter pushing down against satellite surface while cutting), which are identified in [43]. The task monitor can output three possible states as summarized in Table 5.1, where $F_{diff}$ ($F_{diff} = F_{est} - F_{measured}$) is the difference between the measured and estimated forces in the cutting direction, $F_{sliding}$ is the lower threshold for the SLIDING event and $F_{bunch}$ is the upper threshold that triggers the BUNCH event. If the measured force is significantly lower than the estimated force, it indicates that the cutter may have slipped out of the tape seam and may only be sliding along the surface (not cutting). If the measured force is too high, it indicates that the tape may have bunched up during cutting. Based on the previous study [43], these threshold values are chosen to be 30% higher than the estimated force for $F_{bunch}$ and 30% lower than estimated force for $F_{sliding}$, respectively.

The *Task Monitor* runs on the remote side with a motivation to potentially stop action in case of an adverse event. The results of the task monitor, including force difference and task state, are transmitted (with telemetry delay) to the master side to be displayed to the user, as shown in Figure 5.2. The display includes a color coded text (green for NORMAL, orange for SLIDING, red for BUNCHING) indicating the current cutting state and a gauge bar visually shows the difference between the measured and estimated cutting force (the black bar) in real time. The force difference

**Table 5.1:** List of task monitor states.

| No. | Force Condition | State | Text Color |
|---|---|---|---|
| 1 | $F_{diff} < F_{sliding}$ | SLIDING | Orange |
| 2 | $F_{diff} > F_{bunch}$ | BUNCH | Red |
| 3 | Others | NORMAL | Green |

bar operating in the green region indicates NORMAL cutting state, the red portions on the left and right indicate the SLIDING and BUNCH states, respectively. To prevent sudden jumps in the black force bar caused by noise in the force sensor, the force difference is low-pass filtered before being displayed to the user.

### 5.3.4 Line Virtual Fixture

Given that the task is to cut the insulation seam, which is a straight line, to further remove uncertainty from the system, a line virtual fixture with haptic feedback is also provided to the user. The user can enable the line virtual fixture if the HPFC is turned on and the cutter is pressed against the plate. The fixture line is initialized with a position by projecting the current cutter position onto the plane model and a direction of the current cutter X axis (the pointing axis) onto the same plane model. In this mode, both position and orientation of the da Vinci master manipulator are locked and the user controls the pointing direction of the virtual line by rotating the final axis (roll) of the master robot.

When defined, the master to cutter mapping is modified such that moving forward on the master side commands the cutter in the line fixture direction and moving left/right on the master side moves the cutter in the direction that is orthogonal to the line virtual fixture direction. Similar to the plane virtual fixture, a haptic rendering of the line is also presented to the user to guide his/her motions. This is a soft virtual fixture and the gains are chosen empirically such that it is strong enough to guide users and soft enough to allow them to overpower the virtual fixture and give lateral commands. This assistance can be further augmented by choosing different gains on different moving directions [92], that is, with smaller gain in the lateral direction.

### 5.3.5 Predictive display

A predictive cutter position is displayed to the user using a blue augmented reality marker overlay. This display appears once the teleoperation enters FOLLOW mode. The predicted cutter position starts off at the current cutter position and is updated every loop cycle by integrating the command twist sent to the slave. This predicted cutter pose is then used for overlaying the augmented blue cutter. The predictive position is motivated by the desire to provide “real-time” feedback to the user. However, discrepancy between the predicted position and the actual cutter position can occur due to multiple factors: First, the integration is done in different components, one on the ground in the teleoperation component and the other on the remote side. Second, the existence of HPFC and other controllers on the slave side implies that the

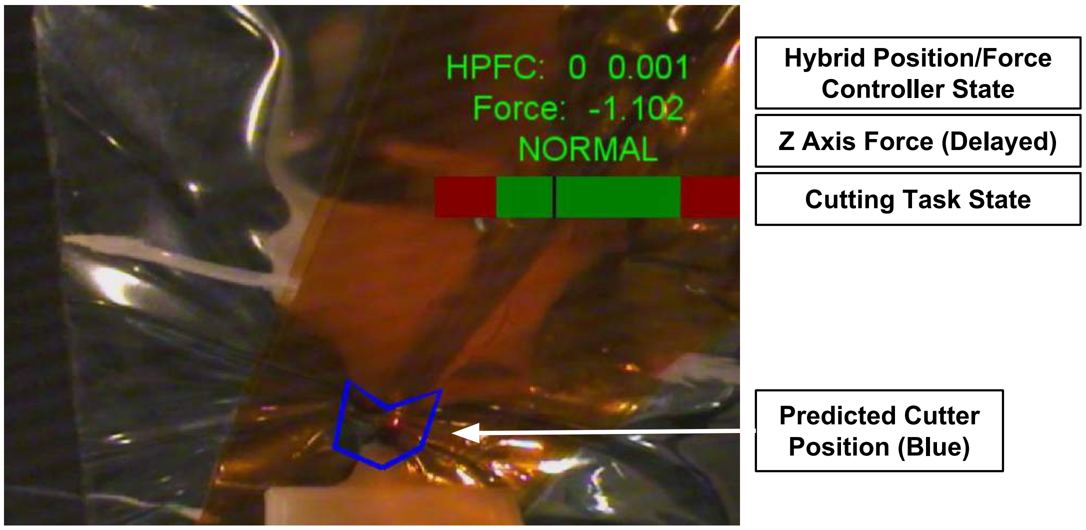


**Figure 5.2:** Screen shot of the operator view in assisted mode.

twist command received might be modified locally. Finally, there is residual error in external camera calibration. The predictive position is synchronized with the delayed telemetry position from the remote site whenever the user enters the teleoperation mode or presses a clutch button to reposition the master. These events frequently occurred in our experiments and thus the discrepancy was minimal and did not affect the user's operation. One potential improvement would be a better method to prevent accumulation of error between the predictive position and the actual position.

## 5.4 User Study

To evaluate the effectiveness of machine assistance in the satellite servicing task with time delay, we conducted a user study that includes freehand and assisted modes with different delay configurations. This section reports the system implementation,

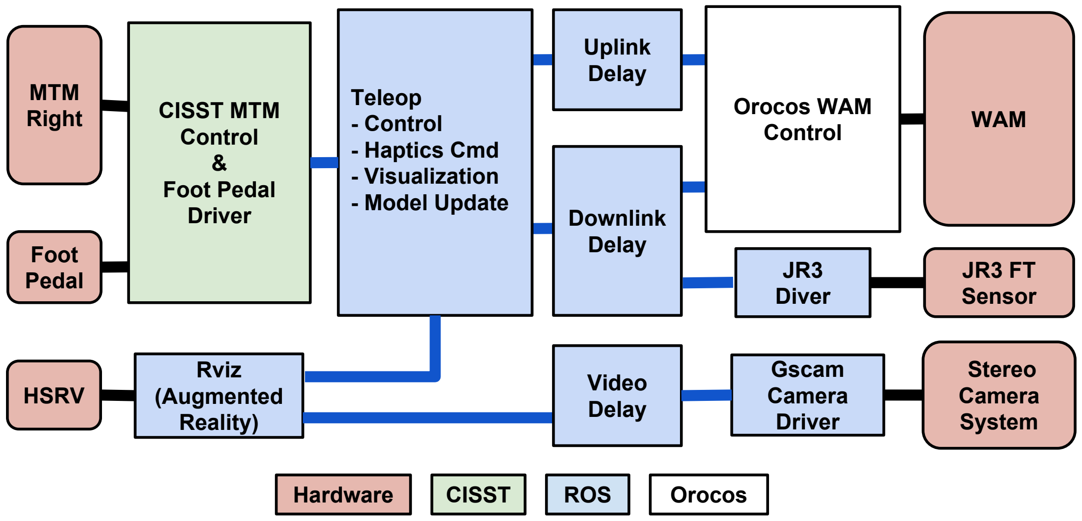


**Figure 5.3:** Block diagram showing hardware and software components in the system

operating conditions, and test procedure, followed by the data collection and analysis metrics.

### 5.4.1 System Implementation

The above discussed teleoperation system is implemented in a testbed at Johns Hopkins University (JHU). Figure 5.3 is a block diagram that summarizes the hardware, software system and their connections.

The master console from the dVRK platform provides the master input device for the user and consists of two 7 DOF da Vinci MTMs, of which only the right MTM is used, a Foot Pedal Tray, and a High Resolution Stereo Viewer (1024 x 768 in our system, though many other dVRK systems are 640 x 480). The buttons on the foot pedal tray can be triggered by the user as control input (e.g., to enable teleoperation, define and cancel the line virtual fixture). The dVRK is controlled by the custom IEEE-1394 (FireWire) controller described in Chapter 2, with the multi-rate component-based software architecture based on the open-source cisst/SAW [23] libraries on a Linux PC, as described in Chapter 3. The controller takes torque commands for each joint and the joint level servo control runs at more than 1 kHz for a good haptic performance. The middle level controller (e.g., kinematics) runs at 500 Hz and provides a ROS interface through a *cisst*-to-ROS bridge component.

A 7 Degrees Of Freedom (DOF) WAM robot is used as the slave device, emulating the servicing robotic arm in space, as shown in Figure 5.5. The WAM robot is mounted vertically on a bench built with frames from 80/20® Inc. Cutting is performed with a steel cutting tool manufactured by JWB manufacturing, Tempe, AZ with a shape and size designed based on a titanium cutting blade used by NASA Goddard Space Flight Center's RRM operation. A six-axis force-torque sensor (JR3 Inc., Woodland, CA) is mounted between the WAM wrist palm link and the cutting blade mount to get contact force measurement on the cutter and to enable active force control in the direction of the plane normal. The WAM robot is controlled using the Orocos Real Time Toolkit (RTT) [11] at 1 kHz.

The master and slave control PCs are connected in a local network and communication between them, including video and telemetry, is accomplished via ROS messages. The uplink, downlink and video delay that exist in the teleoperation between Earth and the satellite are implemented with the *TimeSequencer* filter from the ROS

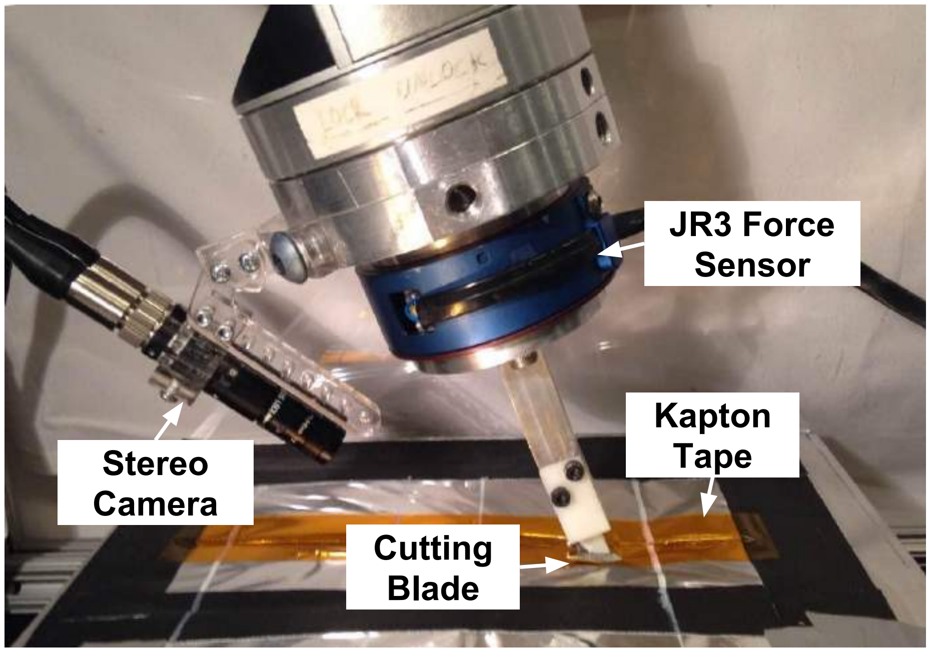


**Figure 5.4:** Cutting test setup including JR3 force sensor, cutting blade, Kapton tape and the stereo camera rig.

*message_filters* package. Delay configuration is described in the next subsection.

Our vision system consists of two SONY lipstick cameras (see Figure 5.4). A stereo camera calibration procedure is performed to obtain the camera intrinsic parameters that are later used for image rectification. The rectified stereo video with augmented reality overlays, generated with RViz (wiki.ros.org/rviz), are delayed and then displayed to the user in the dVRK master console stereo viewer.

For the cutting experiments, we used mockup MLI blankets constructed from representative industrial materials (Kapton tape, McMaster 7648A34) that resemble the physical properties of the space-qualified MLI materials (Figure 5.4). A total of 6 test strips are mounted on an aluminum plate in two layers. The design and construction of the mockup is described in detail by Vozar et al. [93].

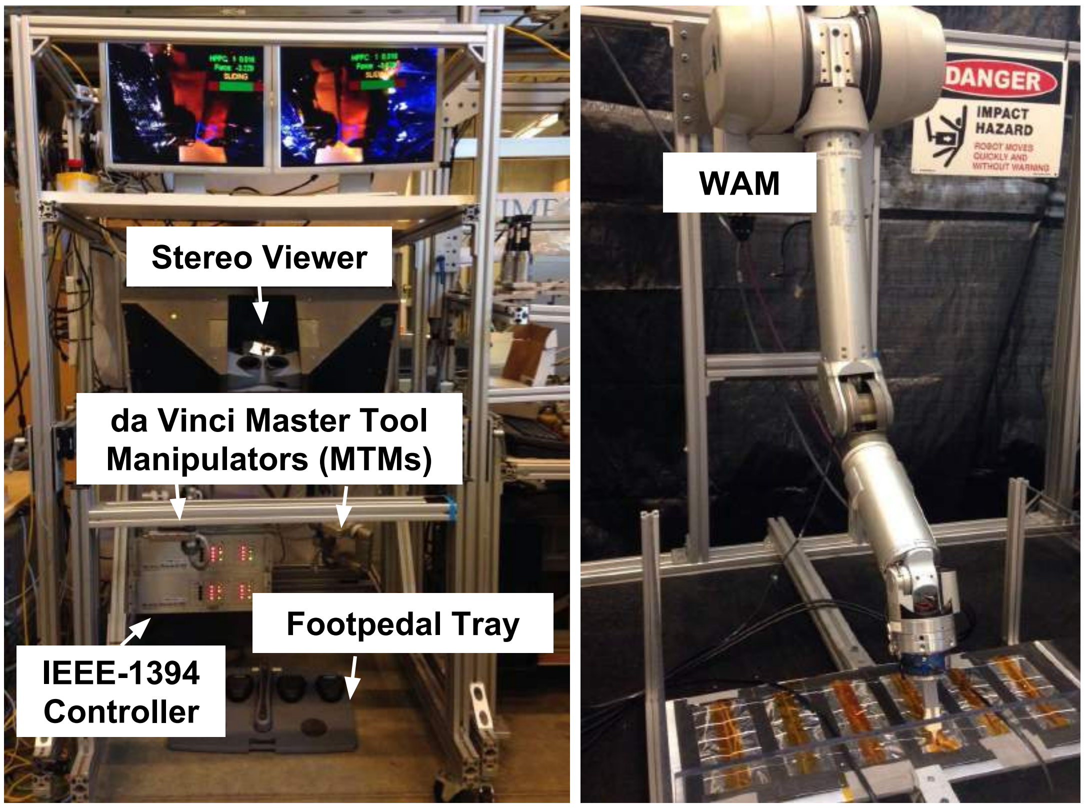

Stereo Viewer
da Vinci Master Tool Manipulators (MTMs)
Footpedal Tray
IEEE-1394 Controller
WAM
DANGER
IMPACT HAZARD
ROBOT MOVES QUICKLY AND WITHOUT WARNING

**Figure 5.5:** da Vinci master console (left). A Barrett Whole Arm Manipulator (right) robot shown with the test blanket setup.

### 5.4.2 Test Conditions

Teleoperation from Earth to space is subject to signal delays both uplink and downlink due to radio frequency (RF) signal time-of-flight propagation delays and, in most cases, the encoding/decoding and software delays in using the relaying system, such as NASA's advanced Tracking and Data Relay Satellite System (TDRSS) [10]. The telemetry and video data have different payload sizes and can have different delays due to the possible use of different communication routes, the compression and decompression of video data and other bandwidth induced latency. For example, the European Space Agency's (ESA's) HAPTICS-2 experiment [12] is reported to utilize a low-latency S-band link for telemetry data through the Russian module on the International Space Station (ISS) in addition to video transfers via the regular Ku-band communication through TDRSS. This link is direct line-of-sight communication and features little transmission delay, but has limited bandwidth and therefore is not suitable for transmitting large data such as videos. Even when telemetry and video data are transmitted via the same route, they can have very different delays. Bohren et al. [8] reported an average telemetry delay of 200 ms and average video delay of 2 s with peaks up to 6 s in a transatlantic teleoperation experiment. To evaluate how the different telemetry and video delays may affect the proposed strategy, two delay configurations are used. Table 5.2 summarizes the three test conditions used in this study. These conditions differ in control modes as well as delay configurations.

The first condition is the *Freehand* mode. In this mode, the user only has delayed

**Table 5.2:** Test conditions

| Condition | Mode | Uplink (s) | Downlink (s) | Video (s) |
|---|---|---|---|---|
| 1 | Freehand | 0.5 | 0.5 | 3.5 |
| 2 | Assisted | 0.5 | 0.5 | 3.5 |
| 3 | Assisted | 0.0 | 4.0 | 4.0 |

video feedback and a text displaying the Z axis of the delayed force measurement in the upper right corner of the screen. The basic teleoperation scheme described in Section 5.3 is employed, where the user manually controls all 6-DOF of the slave cutter. A safety monitor is deployed at the remote end. It monitors the force measurement and turns off the PID controller, such that the WAM controller only outputs joint torques for gravity compensation, which is considered to be a safe mode. The user is notified when the safety monitor triggers and can resume telemanipulation by repressing the *ENABLE* pedal. In terms of delay, a round-trip 1 second telemetry delay (uplink 0.5s, downlink 0.5s) and a 3.5 second video delay configuration is used. This is equivalent to the the overall 4s delay used in our previous work.

The assistance mode is enabled for both the second and third test conditions. In this mode, all the features including predictive display, hybrid position/force controller, plane virtual fixture, line virtual fixture and task monitoring are available to the user. The user can choose whether to use the line virtual fixture. The only difference between these two test conditions lies in the delay configuration. While the teleoperation performance is reported to be independent of the delay location (e.g., uplink vs. downlink) [84], the different delay configurations can possibly make a difference because the force text display and the task monitoring feedback are only delayed by 0.5s in the second condition and by 4s in the third condition. Also, the model update block only uses telemetry information and thus can update the slave model faster when the round trip telemetry delay is 1s.

### 5.4.3 Procedure

The user study was performed with volunteer users recruited from a population of graduate and undergraduate students at JHU and with variable experience in teleoperated robotic operation. Tests were approved by the JHU Homewood Institutional Review Board (HIRB00000701). A total of 12 participants, all right-handed, completed the trials (10 male, 2 female). All volunteers have no neurological disorders nor physical conditions that may negatively affect performance. Volunteers each received a $15 USD Amazon gift card for their participation.

The experiment was divided into four parts: an introduction section, a training and practice section, an experiment section and a subjective evaluation section. After finishing the pre-experiment survey, the user was given a brief introduction of the task background, the master and slave robots used in the experiment, and the task setup.

The practice section starts with the user operating in freehand mode to finish the approach sub-task (i.e., position the cutter in the tape seam) and cut half of the tape seam. Then, all the capabilities of the assisted system and their related user interface are introduced. After each capability is explained, the user gets the opportunity to practice it right away; this ensures that the user can learn and properly understand the feature and its user interface. After all the features are taught, the user is allowed to cut the remaining half of the practice strip in assisted mode.

Trials were then performed. The robot was positioned 1 inch above the cutting start line, which was indicated with a white line (see Figure 5.4). The test strip was pre-punched because the punch sub-task is a difficult task that requires training and we did not want to extend our trial to more than 90 minutes. The user then moved the cutter underneath the tape (approach sub-task) and cut from the start line to the finish line (also marked by a white line). The user completed a NASA TLX survey [33] after each test condition and a subjective evaluation questionnaire at the end of the entire trial. On average, the study took 90 minutes to complete.

### 5.4.4 Metrics

The states of the master and slave robots (500 Hz), video feeds (including augmented reality overlays) of the stereo cameras (30 Hz), task states (model updates) and foot pedal events were logged for all trials. The task completion time and operator workload were analyzed statistically. We did not conduct the edge roughness analysis, described in [93], as it is less relevant to the task compared with task completion time and operator workload.

## 5.5 Results and Discussion

The results for the multi-user study are reported here, together with a discussion of the results. A paired t-test is used for statistical analysis of task completion and operator workload.

### 5.5.1 Task Completion Time

The total task completion time starts from the moment the user starts moving the da Vinci MTM to the moment the cutter passes the finish line. The approaching sub-task and cutting sub-task are timed and analyzed separately. If an adverse event occurs during the cutting, the time spent to manually move the cutter out and re-position it is excluded from the cutting time. Defining or redefining the line virtual fixture is considered as part of the cutting sub-task and thus is timed.

The boxplot shown in Figure 5.6 summarizes the cutting sub-task completion time. The mean task completion time was 198.5s for freehand mode (Condition 1) compared with 145.25s for assisted mode with 1s telemetry delay (Condition 2) and 164.5s for assisted mode with 4s telemetry delay (Condition 3). There is a statistically significant difference between the freehand mode and assisted mode with 1s telemetry delay ($p<0.01$, power=0.858). However, the difference is not significant between the freehand mode and assisted mode with 4s telemetry delay ($p=0.166$, power=0.273).

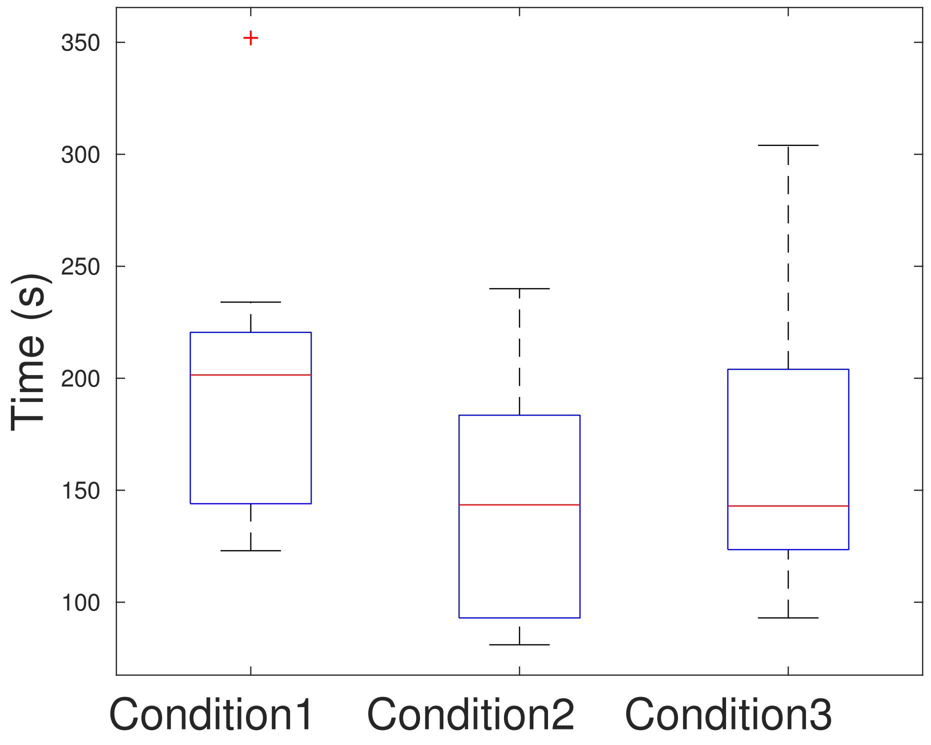


**Figure 5.6:** Boxplots showing the task completion time of the cutting sub-task for each test condition. For all boxplots in this chapter, the red centerline represents the median, the upper and lower edge of each box corresponds to 25th and 75th percentiles, and the whiskers extend to the most extreme data points. The red cross points are considered outliers [28]

### 5.5.2 Operator Workload

The boxplot shown in Figure 5.7 summarizes the overall workload (the sum of the responses to all categories). The mean workload for the freehand condition was 23.08 compared with 13.75 for the assisted mode (condition 2) with the same delay condition and a paired t-test reveals that the effect of assistance is significant (p = 0.00021, power = 0.9983). The effect of different delay configurations in assisted mode is also investigated by comparing condition 2 and condition 3 data and the result shows that there is no significant difference of operator workload (p = 0.91504, power = 0.0511).

The radar plot in Figure 5.8 shows the mean values of each workload category in the NASA TLX survey for each test condition. The assistance mode resulted in less workload than the freehand mode in all categories, regardless of delay configuration. Responses for the assisted mode with different delay configurations are fairly close in all categories.

Table 5.3 summarizes the paired t-test results. The overall workload is significantly different between freehand and assisted mode with 3.5s video delay, so are all the individual categories except for temporal. It is interesting to discover that the users did not perceive a significant difference in terms of time, but the analysis of task completion time did show a significant difference. Different delay configurations did not result in statistically significant differences in assistance mode.

**Table 5.3:** Paired t-test results of operator workload

| | **Condition 1 vs. Condition 2** | | | **Condition 2 vs. Condition 3** | | |
|---|---|---|---|---|---|---|
| | **h** | **p-value** | **power** | **h** | **p-value** | **power** |
| **Overall** | 1 | 0.0002 | 0.9983 | 0 | 0.9150 | 0.0511 |
| **Mental** | 1 | 0.0004 | 0.9949 | 0 | 0.1661 | 0.2732 |
| **Physical** | 1 | 0.0024 | 0.9456 | 0 | 0.1911 | 0.2468 |
| **Temporal** | 0 | 0.1360 | 0.3122 | 0 | 0.5863 | 0.0807 |
| **Performance** | 1 | 0.0063 | 0.8637 | 0 | 0.5863 | 0.0807 |
| **Effort** | 1 | 0.0014 | 0.9707 | 0 | 0.2750 | 0.1829 |
| **Frustration** | 1 | 0.0008 | 0.9869 | 0 | 0.1911 | 0.2468 |

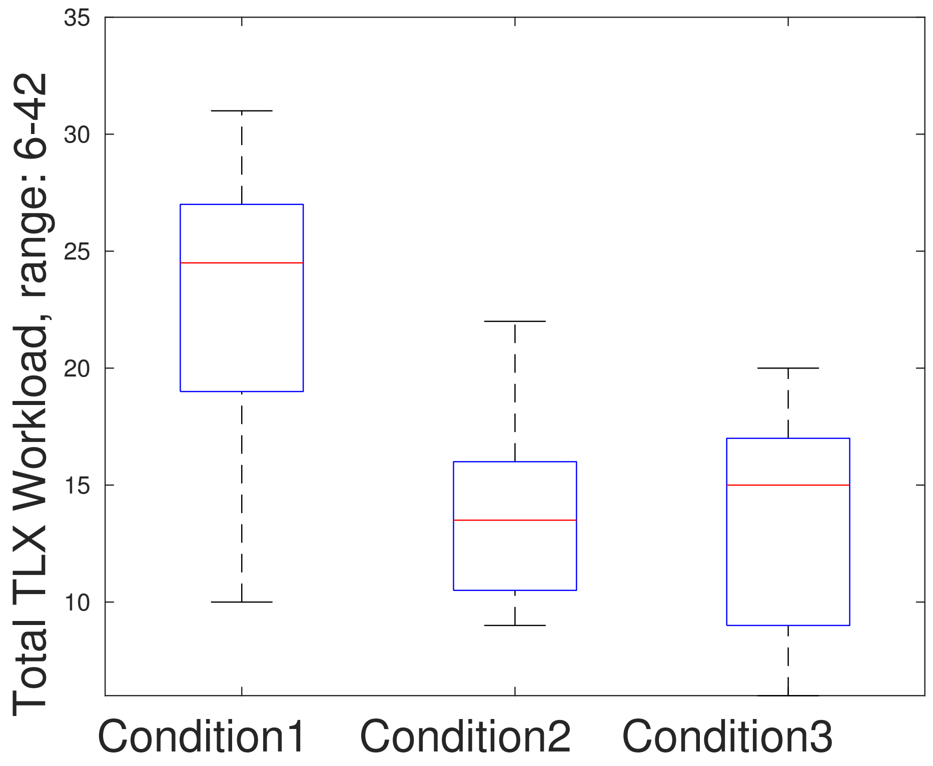


**Figure 5.7:** Boxplots showing the total operator workload as self-reported via the NASA TLX survey for each test condition.

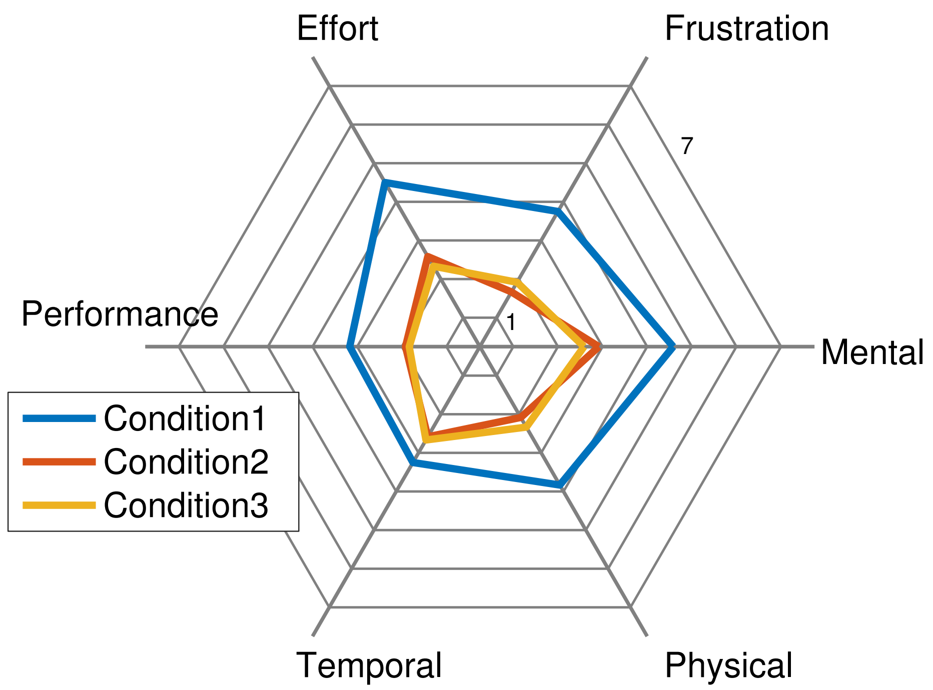


**Figure 5.8:** NASA TLX survey radar plot of average categorical workload as self-reported by the users. Workloads increase from the center (1 to 7).

## 5.5.3 Adverse Events

Here we examine adverse events that occurred under each task condition. Two types of adverse events occurred during the study: (1) failing to position the cutter in the approaching sub-task, and (2) bunching during the cutting sub-task. In total, there are 6 occurrences: 2 approaching sub-task failures and 4 bunching events. All 2 failures in the approaching sub-task happened during the freehand operation due to the fact that the users were not able to align the cutter and constantly triggered the safety threshold. 3 out of 4 bunching events occurred in the assistance mode with the line virtual fixture defined and 1 occurred in freehand mode. Further investigation is required to understand why all bunch events in assisted mode happened when the line virtual fixture was utilized.

### 5.5.4 Subjective Evaluation

While statistically all the assistance features are evaluated as an integrated system, some user feedback gave hints on how each feature helped them to complete the task. One user pointed out that the predictive cutter display also works as a visual feedback that helps the operator to avoid unintentional motion of the master manipulator. Another user stated that the haptic plane was helpful both during the approaching sub-task and in the cutting sub-task as he was more confident in taking bigger steps.

Users also gave some suggestions on system implementation. One user suggested to have an attractive force in the plane virtual fixture to keep the hand on the plane. Another user suggested to display a translucent overlay of the line virtual fixture once set. There was also feedback that constantly pressing a foot pedal to enable teleoperation was a little taxing.

### 5.5.5 Discussion

#### 5.5.5.1 Line Virtual Fixture

In trials using assistance, the users were told that the line VF feature is an optional feature and they can decide whether to use it. 9 out of 12 users used this feature (16 out of 24 assisted trials). The proposed user interface for defining a line virtual

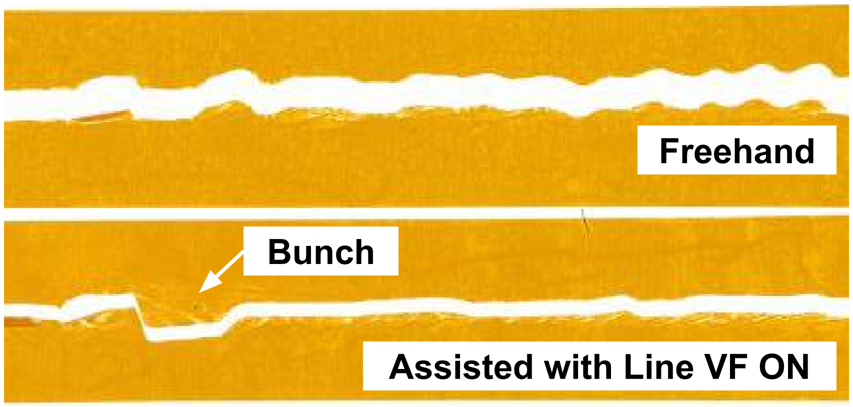


**Figure 5.9:** Comparison of tape cutting result from one user between freehand mode (top) and assisted mode with line VF turned on (bottom).

fixture is effective in terms of time. For trials using the line VF, average time spent on defining the VF is only 6.5% of the total cutting time. However, the effect of the line virtual fixture seems to be unclear. Figure 5.9 shows a cutting tape strip from user 10. The edge of the tape is much smoother when using the line virtual fixture compared with freehand mode. But on the other hand, all bunch events during assisted mode were with the line virtual fixture.

#### 5.5.5.2 Task Monitor

Despite the use of a simple model with fixed parameters, the *Task Monitor* worked well and was able to detect all the bunch events. There were two occasions where the *Task Monitor* produced a false positive. One of them happened right after a bunch event and the other one was when the cutter did not fully cut into the tape, but did not result in an actual bunch event. However, we noticed that users who encountered the bunch event did not pay attention to the displayed state (text overlay) and continued cutting. This is partly because we did not specifically ask the user to stop when he/she sees the warning, and partly because the user was not able to see the condition clearly due to the limited video quality. In the future, this *Task Monitor* would be used to automatically abort task execution on the slave side to prevent further damage.

#### 5.5.5.3 Cutting Strategy

When delayed video is the only feedback to the user, a move-and-wait strategy is commonly used [26]. For the 12 users, the mean wait time was 82.07 s for freehand mode and 63.66s for assisted mode. However, the difference is not statistically significant for this number of trials. Figure 5.10 shows representative velocity profiles from User 03. In freehand mode, the user clearly used the move-and-wait strategy and took a lot of small steps during the cut. The total wait time was 108.10 s (51.97% of the cut time). With the assistance mode (line VF turned on), the user took larger steps and only spent 45.12 s waiting.

Users seem to use different cutting strategies (compression versus tension-based) depending on test conditions. In the assistance mode, the HPFC used a compression-based approach. However, in freehand mode, 7 out of the 12 users chose (intentionally or unintentionally) the tension-based approach. No tearing error related to the tension-based approach was observed.

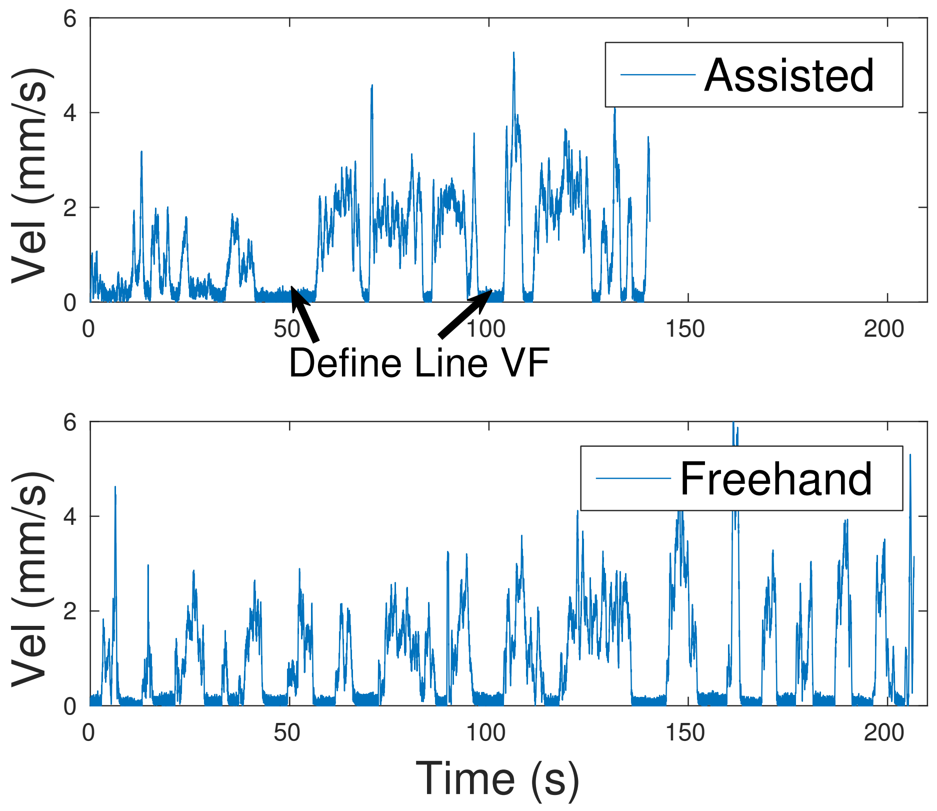


**Figure 5.10:** Velocity profile of trials from user 03: bottom is from freehand mode and top is from assistance mode (condition 2)

#### 5.5.5.4 Other observations

The performance of assistance mode heavily depends on training. There are also two users who spent more than 5 minutes but failed to position the cutter underneath the tape seam in freehand mode. Both of them successfully completed the task in a second trial.

## 5.6 Conclusions

This chapter has described the model-based telemanipulation framework. Although a few parts of the framework such as the model-based teleoperation had been presented before, we integrated a system that comprises all the previously proposed components and added a new component that enables the operator to define and adjust the line virtual fixture. We also added an augmented overlay of the measured force and task monitor output. This system allowed us to perform a multi-user study to verify our belief that a model-based method may perform poorly unless it contains a critical set of features. Compared with the previous study by Vozar et al. [93] that evaluated a specific feature, this study evaluates whether the assistance system provides a benefit with respect to a baseline case of no assistance. The results of the user study indicate that assistance can significantly reduce task completion time and overall operator workload.

This application is another demonstration of the effectiveness of the proposed architecture. Although Xia et al. [99] discussed rendering haptic feedback with a virtual fixture, the haptic feedback feature was not enabled in subsequent user studies due to delay between the teleoperation node and the master control node, as they were running on two separate computers. Using the proposed architecture, we can not only render the haptic feedback efficiently, but also run other components such as augmented reality and the teleoperation logic via a high level interface on the same computer.

One limitation of the work is the sample size, which did not allow us to conclude statistical significance in some cases. The other limitation is that due to the time constraint, some users were not able to fully master all the assistance features and were not using the system in the designed manner. Also, the punching sub-task was not included in the experiment for the same reason. Future work could include an evaluation of the impact of degraded force sensor feedback (as would be expected if operated in space) and a comparison of the impact of different features on task performance. Similarly, we recognize that computer vision techniques can be employed as sensory feedback for updating the line virtual fixture model and improving augmented reality performance.

# Chapter 6

# Conclusions

This thesis discusses the development of a scalable and real-time capable infrastructure to support high-performance control of high degrees-of-free systems, with the open source da Vinci Research Kit as the driving platform. The proposed architecture involved the design of an efficient broadcast-based FireWire protocol, the choice and implementation of a component-based multi-threaded software structure that maximizes performance. The timing performance of the proposed FireWire protocol has been measured, analyzed and compared to the state-of-the-art EtherCAT implementation. The architecture has also been demonstrated on the da Vinci Research Kit and applied in two semi-autonomous applications in the medical and space robotics domains. This chapter summarizes each chapter, discusses the thesis contributions and points out possible future works.

## 6.1 Summary of chapters

Chapter 1 gave an introduction of the multi-arm and high degrees-of-freedom da Vinci Research Kit platform and analyzed the challenges of designing a scalable and high performance architecture for such a system.

Chapter 2 focused on the hardware and firmware aspects of the proposed architecture. First, we reviewed the historical context that led to the selection of a distributed architecture based on the IEEE-1394a fieldbus. The initial protocol required one read and one write asynchronous FireWire transaction and did not scale well to a system with 39 DOF. To overcome this limitation, a broadcast-based communication protocol was proposed for scalable real-time performance. This protocol improved timing performance by reducing the number of transactions from the control PC to three, regardless of the number of slave nodes on the bus. The I/O time of a full dVRK drops from over 700 $\mu s$ to less than 200 $\mu s$. To further evaluate the performance of the proposed protocol, a comparison study with EtherCAT was presented. The proposed protocol showed comparable performance and is projected to outperform EtherCAT in large systems. To support the more prevalent Ethernet interface while minimizing hardware modification, an Ethernet-to-FireWire bridge design was introduced.

Chapter 3 presented a software architecture that supports high-performance, low-level control as well as flexible, high-level ROS-based multi-process control. The architecture includes a distributed hardware interface via a high-bandwidth, low-latency fieldbus, the use of a real-time component-based framework with multi-threading and thread-safe shared memory communication, and bridge components that provide interfaces between the real-time component-based framework and other systems such as ROS. The novel aspect of this architecture is that it presents each robot as an independent entity, even though they share resources such as a single communication bus and single thread for low-level control. This software was specifically used for the dVRK, but could be more generally applied to other robot systems.

Chapter 4 presented an application of the high-performance architecture to a surgical semi-autonomous teleoperation application. In this application, we developed virtual fixtures for the needle passing and knot tying sub-tasks of suturing. The virtual fixtures were implemented on the dVRK and tested with a multi-user study to verify their effectiveness. The experiment showed that the proposed virtual fixtures can improve needle passing exit accuracy, thus reducing tissue tearing pressure, and reduce the average knot tying task completion time and number of slips during the task. Although the virtual fixture used in the knot tying task is a simple constraining plane virtual fixture, it is a two-handed virtual fixture and illustrated the potential importance of such coordinated virtual fixtures in such tasks. This application involved four arms and a camera system. The proposed architecture provided a high-performance platform for the haptic rendering of virtual fixtures, as well as a ROS interface to develop behavior logic code in a fast manner.

Chapter 5 showed another application in the space robotics field. This chapter described the model-based teleoperation framework. The research contribution is the line virtual fixture together with a user interface that allows the user to define and adjust it on-the-fly. The chapter also reported the results of a multi-user study to compare the performance of assistance and freehand modes in a satellite servicing teleoperation task under 4 seconds communication delay. The results showed that the assistance can reduce the operator's workload. This application utilized the architecture, particularly the dVRK master console as the master input device, virtual fixture rendering interface, and the display for augmented video signals. The user study, however, is limited by the sample size.

## 6.2 Discussion and Future Work

The major contribution of this thesis is the architecture comprised of both the FireWire communication protocol and the software architecture that interfaces to the hardware. Although it was designed and implemented on the dVRK, it can be applied to other robot systems.

The two applications of this architecture towards semi-autonomous teleoperation demonstrated that the proposed architecture can support complex systems using multiple arms and is able to efficiently render haptic feedback at multiple kilo-hertz. However, an evaluation that focuses on the specific applications enabled by the hard real-time and ultra-high update frequency could be conducted.

Regarding the suturing application, one future research direction is on the haptic rendering mechanism. In the current framework, the force and torque representation are decoupled and specified independently. A coupled representation with full 6 DOF can be used to represent renderings that require simultaneous positional and rotational haptic cues. Another direction is to explore different virtual fixtures for the suturing task.

# Appendix A

# Appendix: How to Compile RTnet and Xenomai

In Chapter 2, we presented EtherCAT timing performance collected on a Xenomai patched Linux computer with the RTnet [52] Ethernet driver. Here we document how to setup a test computer and collect timing data.

Xenomai [29] provides a real-time extension kernel that is seamlessly integrated into Linux. In the test setup, Xenomai 2.6.3 and Linux kernel 3.5.7 are used. A tutorial showing how to patch, configure, compile and install a Xenomai-patched Linux kernel is available from Bohren [7].

RTnet [52] is an open-source, real-time networking framework for Xenomai-patched Linux. As the standard Ethernet protocol is non-deterministic, RTnet avoids unpredictable collisions and congestions using an additional protocol layer called RTmac, which controls the medium access. However, when used for real-time robot control applications, the connection between the control PC and the robot is typically a point-to-point connection; therefore, the use of RTmac is not necessary, and RTnet should be configured accordingly.

For this thesis work, RTnet 0.9.13 was used. To install RTnet, download RTnet 0.9.13 from www.rtnet.org, install `libncurses5-dev` using `apt-get`, then use `make menuconfig` to start configuration. In the build configuration step, select `Xenomai`, `TCP Support`, drivers (e.g., `New Intel Pro/1000` for Intel network card), `Real-Time Capturing Support` and optionally `RTnet Application Examples`. After configuration, call `make` to compile RTnet and then `sudo make install` to install the RTnet framework. An installation tutorial is also available in [36]. By default, a script `rtnet` is used to start and stop the RTnet driver stack with RTmac enabled by default. For our application, please use the bash script start_rtnet.sh to start RTnet. The script assumes the use of an Intel network card. If a different network card is used, an update to the default Linux network card driver module name and the RTnet real-time driver module name is required. In addition, update the MAC address in the last line of the script to match the network card in use.

SOEM is an open-source EtherCAT master stack. A version with Xenomai support is available at https://github.com/zchen24/SOEM/tree/thesis. Use `CMake` to configure and then compile the stack. An executable `thesis` is available to collect timing data.

# Bibliography

[1] N. Ahmidi, L. Tao, S. Sefati, Y. Gao, C. Lea, B. Bejar, L. Zappella, S. Khudanpur, R. Vidal, and G. D. Hager. A dataset and benchmarks for segmentation and recognition of gestures in robotic surgery. *IEEE Transactions on Biomedical Engineering*, 2017.

[2] D. Anderson. *FireWire System Architecture*. MindShare, Inc., Addison-Wesley, second edition, 1999.

[3] G. Baltazar and G. P. Chapelle. Firewire in modern integrated military avionics. *IEEE Aerospace and Electronic Systems Magazine*, 16(11):12–16, 2001.

[4] Barrett Technology, LLC. The Whole Arm Manipulator CAN message format. Technical report, Barrett Technology, LLC, 2012.

[5] J. Baumgartner and S. Schoenegger. POWERLINK and real-time Linux: A perfect match for highest performance in real applications. In *12th Real-Time Linux Workshop*, 2010.

[6] R. Bischoff, J. Kurth, G. Schreiber, R. Koeppe, A. Albu-Schäffer, A. Beyer, O. Eiberger, S. Haddadin, A. Stemmer, G. Grunwald, et al. The KUKA-DLR lightweight robot arm-a new reference platform for robotics research and manufacturing. In *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*, pages 1–8, Munich, Germany, June 2010.

[7] J. Bohren. Xenomai on Ubuntu 12.04 (LTS) – A Xenomai tutorial for roboticists. http://jbohren.com/tutorials/2014-09-27-xenomai-precise/, Sept. 2014.

[8] J. Bohren, C. Papazov, D. Burschka, K. Krieger, S. Parusel, S. Haddadin, W. L. Shepherdson, G. D. Hager, and L. L. Whitcomb. A pilot study in vision-based augmented telemanipulation for remote assembly over high-latency networks. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3631–3638. IEEE, 2013.

[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 Interface Conference on Human-Robot Interaction (HRI)*, pages 149–156, March 2016.

[10] D. L. Brandel, W. A. Watson, and A. Weinberg. NASA's advanced tracking and data relay satellite system for the years 2000 and beyond. *Proceedings of the IEEE*, 78(7):1141–1151, 1990.

[11] H. Bruyninckx. Open robot control software: the OROCOS project. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 3, pages 2523–2528. IEEE, May 2001.

[12] M. Bualat, W. Carey, T. Fong, K. Nergaard, C. Provencher, A. Schiele, P. Schoonejans, and E. Smith. Preparing for crew-control of surface robots from orbit. In *IAA Space Exploration Conference*, pages 1–7, 2014.

[13] T. Carstens and G. Harris. Programming with pcap @ONLINE.

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

[15] G. Cena, L. Seno, A. Valenzano, and S. Vitturi. Performance analysis of Ethernet Powerlink networks for distributed control and automation systems. *Computer Standards and Interfaces*, 31(3):566–572, 2009.

[16] P. Chalasani, L. Wang, R. Roy, N. Simaan, R. H. Taylor, and M. Kobilarov. Concurrent nonparametric estimation of organ geometry and tissue stiffness using continuous adaptive palpation. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4164–4171, May 2016.

[17] Z. Chen, A. Deguet, R. Taylor, S. DiMaio, G. Fischer, and P. Kazanzides. An open-source hardware and software platform for telesurgical robot research. In

*MICCAI Workshop on Systems and Arch. for Computer Assisted Interventions*, Sep 2013.

[18] Z. Chen, A. Deguet, R. Taylor, and P. Kazanzides. Software architecture of the da Vinci Research Kit. In *The First IEEE International Conference on Robotic Computing*, Taichung, Taiwan, Apr. 2017.

[19] Z. Chen and P. Kazanzides. Multi-kilohertz control of multiple robots via IEEE-1394 (firewire). In *IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6. IEEE, April 2014.

[20] Z. Chen, A. Malpani, P. Chalasani, A. Deguet, S. S. Vedula, P. Kazanzides, and R. H. Taylor. Virtual fixture assistance for needle passing and knot tying. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 2343–2350. IEEE, 2016.

[21] D.-L. Chow, R. Jackson, M. Cavusoglu, and W. Newman. A novel vision guided knot-tying method for autonomous robotic surgery. In *IEEE International Conference on Automation Science and Engineering (CASE)*, pages 504–508, Aug 2014.

[22] 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. *IEEE Robotics and Automation Magazine*, 22(1):76–85, 2015.

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

[24] S. DiMaio and C. Hasser. The da Vinci research interface. In *MICCAI Workshop on Systems and Arch. for Computer Assisted Interventions*, Midas Journal: http://hdl.handle.net/10380/1464, July 2008.

[25] M. Farsi, K. Ratcliff, and M. Barbosa. An overview of controller area network. *Computing and Control Engineering Journal*, 10(3):113–120, 1999.

[26] W. R. Ferrell. Delayed force feedback. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 8(5):449–455, 1966.

[27] 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*, pages 156–158, Long Beach, CA, Jan 2006.

[28] M. Frigge, D. C. Hoaglin, and B. Iglewicz. Some implementations of the boxplot. *The American Statistician*, 43(1):50–54, Feb. 1989.

[29] P. Gerum. Xenomai-implementing a RTOS emulation framework on GNU/Linux. *White Paper, Xenomai*, page 81, 2004.

[30] U. Hagn, R. Konietschke, A. Tobergte, M. Nickl, S. Jörg, B. Kübler, G. Passig, M. Gröger, F. Fröhlich, U. Seibold, et al. DLR MiroSurge: a versatile system for research in endoscopic telesurgery. *International Journal of Computer Assisted Radiology and Surgery*, 5(2):183–193, 2010.

[31] U. Hagn, M. Nickl, S. Jörg, G. Passig, T. Bahls, A. Nothhelfer, F. Hacker, L. Le-Tien, A. Albu-Schäffer, R. Konietschke, M. Grebenstein, R. Warpup, R. Haslinger, M. Frommberger, and G. Hirzinger. The DLR MIRO: a versatile lightweight robot for surgical applications. *Industrial Robot: An International Journal*, 35(4):324–336, 2008.

[32] B. Hannaford, J. Rosen, D. 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*, 60(4):954–959, Apr. 2013.

[33] S. G. Hart. NASA-task load index (NASA-TLX); 20 years later. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 50(9):904–908, Oct. 2006.

[34] IEEE-1394 Working Group. IEEE Standard for a High Performance Serial Bus and Amendments. *IEEE Std 1394-1995*, 1996.

[35] T. I. Incorporation. EtherCAT on Sitara AM335x ARM Cortex-A8 microprocessors.

[36] Institut des Systèmes Intelligents et de Robotique, Université Pierre et Marie CURIE. RTnet setup on Xenomai. `http://rtt-lwr.readthedocs.io/en/latest/rtpc/rtnet.html`. Accessed: 2017-10-25.

[37] Intuitive Surgical, Inc. The da Vinci Surgical System. `https://intuitivesurgical.com/products/davinci_surgical_system/`. Accessed: 2017-10-25.

[38] R. C. Jackson and M. C. Cavusoglu. Needle path planning for autonomous robotic surgical suturing. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1669–1675. IEEE, May 2013.

[39] G. Jay and C. Crick. gscam ROS wiki, 2012.

[40] I.-K. Jung and S. Lim. An EtherCAT based control system for human-robot co-operation. In *16th International Conference on Methods Models in Automation Robotics (MMAR)*, pages 341–344. IEEE, Aug. 2011.

[41] M. Y. Jung, M. Balicki, A. Deguet, R. H. Taylor, and P. Kazanzides. Lessons learned from the development of component-based medical robot systems. *Journal of Software Engineering for Robotics (JOSER)*, 5(2):25–41, Sept. 2014.

[42] M. Y. Jung, A. Deguet, and P. Kazanzides. A component-based architecture for flexible integration of robotic systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6107–6112. IEEE, Oct. 2010.

[43] I. Kandaswamy, T. Xia, and P. Kazanzides. Strategies and models for cutting satellite insulation in telerobotic servicing missions. In *IEEE Haptics Symposium (HAPTICS)*, pages 467–472, Feb. 2014.

[44] H. Kang and J. T. Wen. Robotic knot tying in minimally invasive surgeries. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 2, pages 1421–1426. IEEE, Oct. 2002.

[45] A. Kapoor, M. Li, and R. H. Taylor. Spatial motion constraints for robot assisted suturing using virtual fixtures. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, pages 89–96. Springer, Oct. 2005.

[46] A. Kapoor, N. Simaan, and P. Kazanzides. A system for speed and torque control of DC motors with application to small snake robots. In *IEEE International Conference on Mechatronics and Robotics (MechRob)*, Aachen, Germany, Sept. 2004.

[47] A. Kapoor and R. H. Taylor. A constrained optimization approach to virtual fixtures for multi-handed tasks. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3401–3406. IEEE, May 2008.

[48] P. Kazanzides, Z. Chen, A. Deguet, G. S. Fischer, R. H. Taylor, and S. DiMaio. An open-source research kit for the da Vinci® surgical robot. In *IEEE International Conference on Robotics and Automation (ICRA)*, May/June 2014.

[49] P. Kazanzides and P. Thienphrapa. Centralized processing and distributed I/O for robot control. In *IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, pages 84–88, Nov. 2008.

[50] J. Kerkes. Real-time ethernet. *Embedded Systems Programming*, 14(1):43–58, 2001.

[51] KINGSTAR. The Best Fieldbus For Your PLC: 5 Fieldbuses Compared For Industry 4.0. Technical report, KINGSTAR, Waltham, MA, 2015.

[52] J. Kiszka, B. Wagner, Y. Zhang, and J. Broenink. RTnet-a flexible hard real-time networking framework. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Catania, Italy, Sep 2005.

[53] R. Konietschke, U. Hagn, M. Nickl, S. Jorg, A. Tobergte, G. Passig, U. Seibold, L. Le-Tien, B. Kubler, M. Groger, et al. The DLR MiroSurge-a robotic system for surgery. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1589–1590. IEEE, May 2009.

[54] N. Korver. Adequacy of the Universal Serial Bus for real-time systems. Technical Report 009CE2003, University of Twente, 2003.

[55] W. F. Lages, D. Ioris, and D. C. Santini. An architecture for controlling the Barrett WAM robot using ROS and OROCOS. In *41st Interaction Symposium*

*on Robotics (ISR) and 8th German Conference on Robotics*, pages 1–8. VDE, 2014.

[56] X. Li and P. Kazanzides. Parameter estimation and anomaly detection while cutting insulation during telerobotic satellite servicing. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4562–4567. IEEE, Sept./Oct. 2015.

[57] X. Li and P. Kazanzides. Task frame estimation during model-based teleoperation for satellite servicing. In *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2016.

[58] S.-Y. Lin, C.-Y. Ho, and Y.-Y. Tzou. Distributed motion control using real-time network communication techniques. In *Proceedings of the Third International Power Electronics and Motion Control Conference (IPEMC)*, volume 2, pages 843–847. IEEE, Aug. 2000.

[59] 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*, 28(9):1183–1197, 2009.

[60] P. Mantegazza, E. L. Dozio, and S. Papacharalambous. RTAI: Real Time Application Interface. *Linux Journal*, 2000(72es), Apr. 2000.

[61] T. H. Massie and K. J. Salisbury. PHANToM haptic interface: a device for probing virtual objects. In *Proceedings of the ASME Winter Annual Meeting, Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, pages 295–299, Chicago, IL, Nov. 1994.

[62] H. Mayer, F. Gomez, D. Wierstra, I. Nagy, A. Knoll, and J. Schmidhuber. A system for robotic heart surgery that learns to tie knots using recurrent neural networks. *Advanced Robotics*, 22(13-14):1521–1537, 2008.

[63] O. Mohareri, C. Schneider, and S. Salcudean. Bimanual telerobotic surgery with asymmetric force feedback: A da Vinci® surgical system implementation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4272–4277. IEEE, Sept. 2014.

[64] A. Munawar and G. S. Fischer. Towards a haptic feedback framework for multi-DOF robotic laparoscopic surgery platforms. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1113–1118, Oct. 2016.

[65] A. Murali, S. Sen, B. Kehoe, A. Garg, S. McFarland, S. Patil, W. D. Boyd, S. Lim, P. Abbeel, and K. Goldberg. Learning by observation for surgical subtasks: Multilateral cutting of 3D viscoelastic and 2D orthotropic tissue phantoms. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1202–1209. IEEE, May 2015.

[66] N. T. Nguyen, K. L. Mayer, R. J. Bold, M. Larson, S. Foster, H. S. Ho, and B. M. Wolfe. Laparoscopic suturing evaluation among surgical residents. *Journal of Surgical Research*, 93(1):133–136, 2000.

[67] J. H. Peters, G. M. Fried, L. L. Swanstrom, N. J. Soper, L. F. Sillin, B. Schirmer, K. Hoffman, S. F. Committee, et al. Development and validation of a comprehensive program of education and assessment of the basic fundamentals of laparoscopic surgery. *Surgery*, 135(1):21–27, Jan. 2004.

[68] G. A. Pratt, P. Willisson, C. Bolton, and A. Hofman. Late motor processing in low-impedance robots: Impedance control of series-elastic actuators. In *Proceedings of the American Control Conference*, volume 4, pages 3245–3251, Boston, MA, June 2004.

[69] G. Prytz. A performance analysis of EtherCAT and PROFINET IRT. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 408–415, Hamburg, Germany, Sept. 2008.

[70] L. Qian, Z. Chen, and P. Kazanzides. An Ethernet to FireWire bridge for real-time control of the da Vinci Research Kit (dVRK). In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–7. IEEE, 2015.

[71] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler,

and A. Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, volume 3, 2009.

[72] M. H. Raibert and J. J. Craig. Hybrid Position/Force Control of Manipulators. *Journal of Dynamic Systems, Measurement, and Control*, 103(2):126–133, June 1981.

[73] S. G. Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov. Using real-time java for industrial robot control. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '07, pages 104–110, New York, NY, USA, 2007. ACM.

[74] L. B. Rosenberg. Virtual fixtures: Perceptual tools for telerobotic manipulation. In *Proceedings of IEEE Virtual Reality Annual International Symposium*, pages 76–82. IEEE, 1993.

[75] R. Rusu, I. Sucan, B. Gerkey, S. Chitta, M. Beetz, and L. Kavraki. Real-time perception-guided motion planning for a personal robot. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4245–4252, Oct. 2009.

[76] A. Ruszkowski, C. Schneider, O. Mohareri, and S. Salcudean. Bimanual teleoperation with heart motion compensation on the da Vinci® Research Kit: Implementation and preliminary experiments. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4101–4108. IEEE, May 2016.

[77] J. Ruurda, I. Broeders, B. Pulles, F. Kappelhof, and C. Van der Werken. Manual robot assisted endoscopic suturing: time-action analysis in an experimental model. *Surgical Endoscopy and Other Interventional Techniques*, 18(8):1249–1252, 2004.

[78] K. Salisbury, W. Townsend, B. Ebrman, and D. DiPietro. Preliminary design of a whole-arm manipulation system (WAMS). In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 254–260. IEEE, Apr. 1988.

[79] L. Santos and R. Cortesão. A dynamically consistent hierarchical control architecture for robotic-assisted tele-echography. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1990–1996. IEEE, 2014.

[80] M. Sarker, C. Kim, J. Cho, and B. You. Development of a network-based real-time robot control system over IEEE 1394: using open source software platform. In *IEEE International Conference on Mechatronics*, pages 563–568, July 2006.

[81] S. Schneider. Making Ethernet work in real time. *Sensors Magazine*, 17(11):22–39, Nov. 2000.

[82] J. Schulman, A. Gupta, S. Venkatesan, M. Tayson-Frederick, and P. Abbeel. A case study of trajectory transfer through non-rigid registration for a simplified suturing scenario. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4111–4117. IEEE, Nov. 2013.

[83] S. Sen, A. Garg, D. V. Gealy, S. McKinley, Y. Jen, and K. Goldberg. Automating multi-throw multilateral surgical suturing with a mechanical needle guide and sequential convex optimization. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4178–4185. IEEE, May 2016.

[84] T. Sheridan. Space teleoperation through time delay: review and prognosis. *IEEE Transactions on Robotics and Automation*, 9(5):592–606, Oct. 1993.

[85] P. Thienphrapa and P. Kazanzides. A distributed I/O low-level controller for highly-dexterous snake robots. In *IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 9–12, Baltimore, MD, Nov. 2008.

[86] P. Thienphrapa and P. Kazanzides. A scalable system for real-time control of dexterous surgical robots. In *IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, pages 16–22, Woburn, MA, Nov. 2009.

[87] P. Thienphrapa and P. Kazanzides. Design of a scalable real-time robot controller and application to a dexterous manipulator. In *IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 2295–2300, Phuket, Thailand, Dec. 2011.

[88] J. P. Thomesse. Fieldbus technology in industrial automation. *Proceedings of the IEEE*, 93(6):1073–1101, June 2005.

[89] Trade Association and others. Firewire$^{\text{TM}}$ reference tutorial. Technical report, 1394 Trade Association, 2010.

[90] M. A. Tsegaye. A comparative study of the Linux and Windows device driver architectures with a focus on IEEE1394 (high speed serial bus) drivers. Master's thesis, Department of Computer Science, Rhodes University, Dec. 2002.

[91] S. S. Vedula, A. Malpani, N. Ahmidi, S. Khudanpur, G. Hager, and C. C. G. Chen. Task-level vs. segment-level quantitative metrics for surgical skill assessment. *Journal of Surgical Education*, 73(3):482–489, 2016.

[92] S. Vozar, Z. Chen, P. Kazanzides, and L. L. Whitcomb. Preliminary study of virtual nonholonomic constraints for time-delayed teleoperation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4244–4250. IEEE, Sept./Oct. 2015.

[93] S. Vozar, S. Leonard, P. Kazanzides, and L. L. Whitcomb. Experimental evaluation of force control for virtual-fixture-assisted teleoperation for on-orbit manipulation of satellite thermal blanket insulation. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4424–4431. IEEE, May 2015.

[94] S. Vozar, S. Leonard, L. L. Whitcomb, and P. Kazanzides. A testbed for evaluating virtual-fixture-assisted teleoperation for on-orbit manipulation of satellite

thermal blanket insulation. In *The 3rd Telerobotics Workshop at IROS: Telerobotics for Real-Life Applications: Opportunities, Challenges, and New Developments*, pages 16–20, Sept 2014.

[95] L. Wang, Z. Chen, P. Chalasani, R. M. Yasin, P. Kazanzides, R. H. Taylor, and N. Simaan. Force-controlled exploration for updating virtual fixture geometry in model-mediated telemanipulation. *Journal of Mechanisms and Robotics*, 9(2):021010, 2017.

[96] Willow Garage. PR2 Manual, 2010.

[97] T. Xia, A. Kapoor, P. Kazanzides, and R. Taylor. A constrained optimization approach to virtual fixtures for multi-robot collaborative teleoperation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 639–644. IEEE, Sept. 2011.

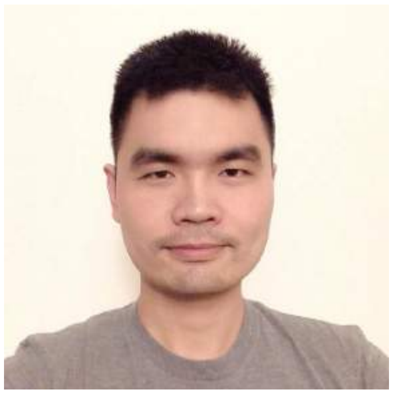
[98] T. Xia, S. Léonard, A. Deguet, L. Whitcomb, and P. Kazanzides. Augmented reality environment with virtual fixtures for robotic telemanipulation in space. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5059–5064, Oct. 2012.

[99] T. Xia, S. Léonard, I. Kandaswamy, A. Blank, L. Whitcomb, and P. Kazanzides. Model-based telerobotic control with virtual fixtures for satellite servicing tasks. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1479–1484, May 2013.

[100] Y. Zhang, B. Orlic, P. Visser, and J. Broenink. Hard real-time networking on firewire. In P. Marquet, N. McGuire, and P. Wurmsdobler, editors, *7th Real-Time Linux Workshop*, pages 1–8, Eindhoven, the Netherlands, 2005. IOP Press.

[101] Y. Zhang, B. Orlic, P. Visser, and J. Broenink. Hard real-time networking on FireWire. In *RT Linux Workshop*, Nov 2005.

# Vita



Zihan Chen received the Bachelors of Science degree in Control Science and Engineering and the Bachelors of Arts degree in English Language and Literature in 2010, and the Masters of Science and Engineering degree in Mechanical Engineering from Johns Hopkins University in 2012. He enrolled in the Computer Science Ph.D. program in 2012. His research focuses on scalable, high-performance control system and semi-autonomous teleoperation.