Surgical Assistant Workstation (SAW) Communication Interfaces for Teleoperation

Min Yang Jung, Gorkem Sevinc, Anton Deguet, Rajesh Kumar, Russell Taylor, and Peter Kazanzides

Abstract— **As robotic systems for complex applications such as surgery evolve, multiple and possibly distributed computing engines are becoming common. This creates a real need for an integrated framework allowing multiple elements of such robotic systems to be developed independently on a framework supporting common functions such as communication. Development of such frameworks is an active area of research, and the Surgical Assistant Workstation (SAW) is our modular software framework that provides integrated support for robotic devices and imaging sensors, as well as analysis, computation and visualization tools. Here we present an overview of inter-process communication (IPC) mechanisms in the SAW, including their performance qualification for teleoperation, and the design of a teleoperated application using these interfaces. Implementations and simple experiments using multiple hardware platforms (Sensable Omni, Novint Falcon devices, and the da Vinci surgical system) showing the modularity of our implementation are also presented.**

I. Introduction

A number of frameworks and packages have been proposed or developed to support the efficient development of distributed robotic surgical systems. Several reviews of such frameworks and packages have already been published [1], [2], [3], [4], [5] and there is a web-site dedicated to cataloguing and reviewing these systems as well as standardizing a reference architecture for robotics [6]. Several commonly used frameworks for robotics research are summarized below and compared to our Surgical Assistant Workstation (SAW). While many of these frameworks adopt a component-based architecture for software reusability, their overall distinct designs reflect the desire to efficiently support a particular project or environment.

Player: Player [7] is a popular set of tools for robotics research including a range of robotic device drivers. Conceptually, it is a hardware abstraction layer for robotic devices that also includes data communication mechanisms among drivers and control programs. Communication *interfaces* are based on a TCP socket-based client/server architecture.

OROCOS: The Open RObot COntrol Software (OROCOS) [8] includes real-time C++ libraries for advanced machine-tool and robot control. Orocos components communicate with each other using interfaces which consist of properties, events, methods, commands and data flow ports and this communication relies on the ACE ORB (TAO), a popular open-source CORBA implementation.

Orca: The Orca project [9], [10] aims to provide building-blocks (components) that can be combined to build arbitrarily complex non-real-time robotic systems. It uses the Internet Communication Engine (ICE) [11] network middleware.

SAW: The Surgical Assistant Workstation (SAW) is a software development framework that enables rapid prototyping of new applications for robot-assisted surgery [12], [13]. SAW consists of a component-based framework and a set of implemented components that provide interfaces to many of the hardware and software modules commonly used for robot-assisted surgery. Hardware interface components support common robotic devices, imagers, and other sensors, including a "wrapper" for the research interface [14] to the da Vinci surgical robot (enabled upon conclusion of a collaborative agreement with the manufacturer). Software components provide functionality such as video processing, 3D user interfaces, and robot motion control. The SAW framework is based on the *cisst* libraries [15] (see Fig. 1), which include a network layer built on top of ICE.

The above frameworks share a component-based design philosophy, although they use different networking solutions. The SAW inter-process communication (IPC) uses ICE, but unlike Orca, the design is not dependent on ICE. It is possible to substitute another middleware package, even a native socket-based implementation, if needed. All Orca components run as separate processes and communicate only via ICE, whereas SAW provides flexiblity for implementation such that tasks may communicate either with inter-thread communication (ITC)–tasks running within a single process– or inter-process communication (IPC)–with tasks being distributed across networks. ITC is performed using a lock-free data exchange mechanism. The programming interfaces (API) are identical in both cases.

Both Orocos and SAW use an Interface Description Language (CORBA IDL in Orocos and ICE Slice in SAW) and allow components to work across networks transparently. Orocos does not yet fully support its component interfaces (i.e., the event interface is not available) and permits only primitive C++ types and `std::vector<double>` to be used across networks. SAW supports a complete set of component interface objects: four command types and two event types. Moreover, all data types provided by the underlying *cisst* libraries can be used across networks. Orocos and SAW both support real-time programming; for Orocos, this is evident in its reliance on a real-time Linux operating system. SAW is truly platform independent, and includes an operating system abstraction library that supports Windows, Linux, and MacOS X as well as real-time Linux variants. This is important for robot-assisted surgery applications (the main application domain for SAW) because there often is

Authors are with the Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA. Peter Kazanzides can be contacted at pkaz@jhu.edu.

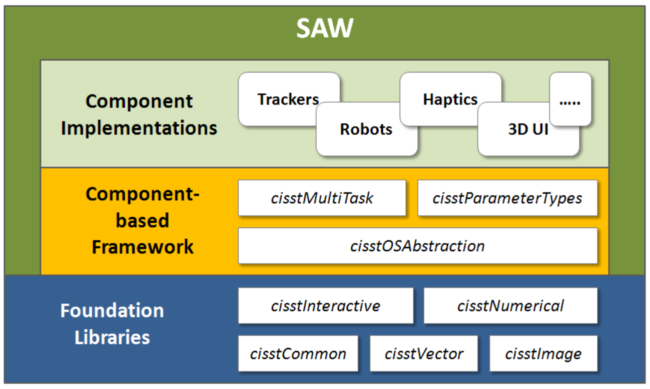


Fig. 1: SAW Components

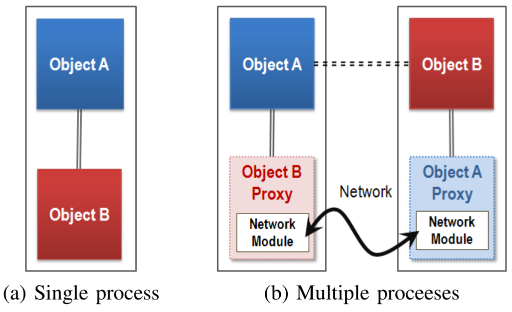


Fig. 2: The concept of the Proxy Pattern

a constraint to use a non-real-time operating system, such as Windows, to enable integration of devices. The SAW network layer enables real-time and non-real-time systems to co-exist within a single distributed application.

The following sections describe the details of the network layer and IPC module.

A. *Design*

1) Proxy Pattern: When two objects–*Object A* and *Object B*–are locally connected to each other (i.e., running within a single process) as in Fig. 2a, the basic concept of the Proxy Pattern is to replace the *local* connection between them by a *conceptual local* connection over a network. Two proxies–the *Object A Proxy* and the *Object B Proxy*–are set up in both processes and locally connect to the corresponding original object in the same process, as in Fig. 2b. Note that a proxy object is always local to its peer original object and the original local connection remains unchanged from the original objects' point of view.

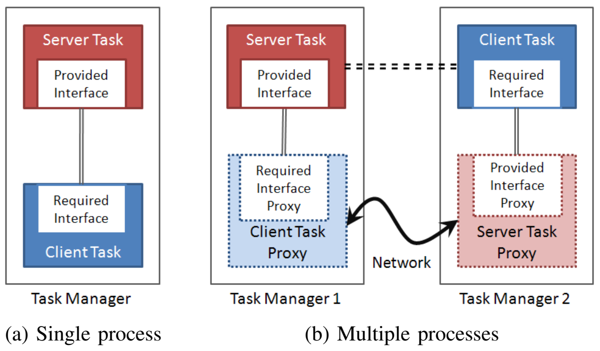


Fig. 3: The Proxy Pattern in the *cisstMultiTask* library

TABLE I: List of Proxies in the *cisstMultiTask* Library

Original Class	Proxy Class

(Base classes for networking)	`mtsProxyBaseCommon`
	`mtsProxyBaseServer`
	`mtsProxyBaseClient`

`mtsTaskManager`	`mtsTaskManagerProxy`
	`mtsTaskManagerProxyServer`
	`mtsTaskManagerProxyClient`

`mtsDevice`	`mtsDeviceProxy`

`mtsDeviceInterface`	`mtsDeviceInterfaceProxy`
	`mtsDeviceInterfaceProxyServer`
	`mtsDeviceInterfaceProxyClient`

`mtsCommandVoid`
`mtsCommandWrite`
`mtsCommandRead`
`mtsCommandQualifiedRead`
	`mtsCommandVoidProxy`
	`mtsCommandWriteProxy`
	`mtsCommandReadProxy`
	`mtsCommandQualifiedReadProxy`

`mtsFunctionVoid`
`mtsFunctionWrite`
`mtsFunctionRead`
`mtsFunctionQualifiedRead`
	`mtsFunctionVoidProxy`
	`mtsFunctionWriteProxy`
	`mtsFunctionReadProxy`
	`mtsFunctionQualifiedReadProxy`

`mtsMulticastCommandWrite`	`mtsMulticastCommandWriteProxy`

The *cisstMultiTask* library defines a *task* or a *device* as a basic component. Tasks include their own execution thread, whereas devices do not; for convenience we shall henceforth use the term task to refer to either. Each task can have any number of *provided interfaces* and *required interfaces*, as in Fig. 3a. Since actual data communication between tasks occurs through interfaces, it is necessary to split this connection using *task proxies* and two types of interface proxies–*provided interface proxy* and *required interface proxy*, as shown in Fig. 3b.

Each interface consists of four types of *commands* (void, write, read, and qualified read) and two types of *events* (void and write). In the multi-threaded case, all commands and events are non-blocking. A client task submits void and write commands to a server task's queue; these commands are later dequeued and executed by the server task. The client task does not wait for a response from the server task. As the names imply, void commands take no parameters and write commands take one parameter (the data to be "written"). Void and write events are similar, except that they originate in the server task and are queued for execution by the client task. Read commands take one parameter and are used by the client task to read buffered data from the server task. Each task contains a circular buffer (called the *State Table*) that provides a thread-safe and lock-free mechanism for client tasks to read data. The qualified read command is conceptually like the read command except that it accepts two parameters; the second parameter allows the client to pass additional information (the qualifier). This latter command is not automatically thread-safe, so programmers must be careful to use the qualifier parameter in a thread-safe manner. For example, it is acceptable to use the qualifier to specify reading of a particular array element.

Thus, the application of the Proxy Pattern to the *cisstMultiTask* library requires proxy objects for all of these objects, as shown in Table I. These proxy objects are completely hidden from an application layer; they are dynamically created and managed internally. We defined three base classes to encapsulate all the implementation details for networking and make adding a new proxy type simple and systematic. With these classes, we could not only significantly increase code reusability, but also manage proxy objects in a more consistent manner. Furthermore, this design allows the library to be independent from a specific network middleware.

2) Networking Middleware: There are many networking middleware packages currently available such as Data Distribution Service (DDS), CORBA, SOAP, Spread, and ICE. Because the network module introduces additional processing for data exchange between tasks, it naturally affects the overall performance of a system. With our own review of these packages and from previous studies on a robotics system with middleware [16], [10], [1], [17], [18], we concluded that ICE best satisfied our design requirements for the following reasons:

Multi-language and cross-platform support: ICE supports C++, Java, Python, etc. Like *cisst*, it can also run under Windows, Linux, and MacOS X.

Interface Description Language (IDL): ICE provides the Specification Language for ICE (SLICE) to define an interface and a data structure for a proxy in an easy, flexible, and extensible way. Since component interfaces and corresponding data structures are key aspects for software reusability [19], SLICE can be a useful tool for designing the network layer.

High network performance: From several reports from ZeroC, ICE is known to perform efficiently in terms of latency and throughput. We performed testing to confirm this, as described in the following sections.

Proxy Pattern: The basic concept of ICE is the Proxy Pattern. Because we adopted it as well in the design of the network layer, the overall structure and implementation of the module can be more consistent and more simplified.

3) Code-level Changes: Our proxy-based IPC module design described above satisfies the requirement for minimal code-level changes because all proxy-related processing is encapsulated inside the *cisstMultiTask* library. To illustrate, we consider a case where two tasks are running in the same process: one is a `serverTask` that offers a provided interface named `providedInterfaceName` and the other is a `clientTask` with a required interface named `requiredInterfaceName`. The library includes a Task Manager class (a Singleton object). Applications register tasks with the Task Manager and then use it to start/stop tasks and establish connections between their interfaces. In this case, the core implementation at the code-level would be the following:

```
taskManager = mtsTaskManager::GetInstance();
taskManager->AddTask(serverTask);
taskManager->AddTask(clientTask);
taskManager->Connect(
    "clientTask", "requiredInterfaceName",
    "serverTask", "providedInterfaceName");
```

The conversion from a multi-threaded (single process) implementation to a networked (multiple process) implementation requires only minor changes to the setup and use of the Task Manager, which now consists of a Global Task Manager (GTM) that manages the local (proxy) Task Managers in each process. The conversion procedure to distribute the tasks consists of three simple steps:

1) Set up the Global Task Manager (GTM): The GTM maintains a list of tasks and their access information. Conceptually, this is similar to the CORBA naming service.

```
taskManager->SetTaskManagerType(
    mtsTaskManager::TASK_MANAGER_SERVER);
```

2) Register Server Task: Set up the local task manager and add the server task. Internally, proxy objects are created dynamically and the task is registered to the GTM.

```
taskManager->SetTaskManagerType(
    mtsTaskManager::TASK_MANAGER_CLIENT);
taskManager->AddTask(serverTask); // as before
```

3) Register Client Task and Connect: Set up the local task manager, add the client task, and connect to the server task. The client task is internally registered to the GTM and the connection between the two tasks is automatically established across the network.

```
taskManager->SetTaskManagerType(
    mtsTaskManager::TASK_MANAGER_CLIENT);
taskManager->AddTask(clientTask); // as before
taskManager->Connect(             // as before
    "clientTask", "requiredInterfaceName",
    "serverTask", "providedInterfaceName");
```

From our experience converting several projects from an ITC-based application to an IPC-based distributed application, this conversion process is simple and clear.

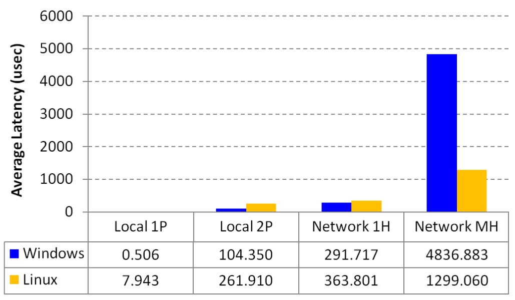
B. Networking Layer Performance

The conversion of an ITC-based application into an IPC-based one requires additional processing such as networking overhead, serialization, and deserialization. This naturally leads to lower overall system performance. To quantify the additional latency, we configured a test-bed and created a performance benchmark application. The specifications of the computers used are:

- **Win1**: Windows XP, Microsoft Visual Studio 2008, Core2 Duo 3.16GHz, 4G RAM, 1 Gbps Ethernet (wired)
- **Win2**: Windows Vista, Microsoft Visual Studio 2008, Core2 Duo 2.5GHz, 4G RAM, 54 Mbps wireless
- **Linux**: Ubuntu 8.0.4, GCC, Pentium IV 3.2 GHz, 1G RAM, 1 Gbps Ethernet (wired)

For IPC-based performance tests, we used the campus wired and wireless networks at Johns Hopkins University. The benchmark application defines a client and server task and uses a qualified read command with arguments of `mtsDouble` type. It obtains the local timing information from the client task, carries it to and from the server task, and calculates the elapsed time (i.e., the execution time of the qualified read command). To investigate how much latency is introduced, we tested four different configurations:

- **Local 1P**: two threads in a single process, single machine, *no networking*.



	Local 1P	Local 2P	Network 1H	Network MH
🟦 Windows	0.506	104.350	291.717	4836.883
🟨 Linux	7.943	261.910	363.801	1299.060

Fig. 4: Networking Overhead: Average Latency

- **Local 2P**: two threads in two processes, single machine, *local* networking (loop-back).
- **Network 1H**: two threads in two processes, two machines, one-hop networking with *wired* Ethernet.
- **Network MH**: two threads in two processes, two machines, multi-hop networking between wired Ethernet and *wireless* network.

The Local configurations were tested on Win1 and Linux, and the Network configurations were tested between Win1 and Win2 (reported as "Windows") and between Win2 and Linux (reported as "Linux"). The qualified read command was executed 10,000 times per test and each test was repeated 5 times.

The results are shown in Fig. 4. As expected, the single-process case (Local 1P) has the lowest latency (less than 10 microseconds). The two process, single machine, case (Local 2P) indicates that the performance penalty due to the ICE middleware is on the order of a few hundred microseconds; this includes the overhead for serialization, deserialization, and dynamic object creation. The Network 1H case shows that the wired Ethernet connection adds about 100-200 microseconds; the total latency of 300-400 microseconds is still acceptable even for fast control loops (on the order of 1 kHz). Finally, the Network MH scenario shows that the "worst case" latency (at least in our limited geographical domain) is a few milliseconds, which is still acceptable for higher-level control tasks, such as teleoperation.

II. Teleoperation Testbeds

We prototyped teleoperation applications using three separate devices. The Sensable Phantom Omni (Fig. 5a) is a haptic device capable of six degrees of freedom (DOF) positional sensing and 3-DOF force feedback. It supports an IEEE-1394a interface and is compatible with Sensable's OpenHaptics toolkit. SAW interfaces to communicate with the OpenHaptics Toolkit layer have been developed, providing the user the ability to read positions, velocities, etc. and send positions without direct interaction with OpenHaptics. The Novint Falcon (Fig. 5b) has 3-DOF position sensing and force feedback and was designed as a gaming controller capable of providing more realistic feedback. It also supports an IEEE-1394a interface and the manufacturer provides a low-level interface library. The Falcon employs a developer's kit similar to Sensable's OpenHaptics, for which SAW interfaces have been created. The da Vinci S Surgical System (Intuitive Surgical, Inc.) is a teleoperated robotic system for minimally invasive surgery. It consists of three components. The surgeon console contains a stereo viewer, configuration controls, and the master handles. A patient side manipulator cart may hold up to four slave surgical instrument manipulators, including one dedicated for holding an endoscopic camera, and a touch screen. A vision cart contains the supporting electronics for the stereo video endoscope used for visualization. It includes a network interface (the research API [14]), enabled only on certain systems by the manufacturer, that streams the system state over an Ethernet connection allowing da Vinci masters to be used for teleoperation of other devices. A set of SAW interfaces is used to interact with this read-only API.

A. Teleoperation methods

Fig. 6 outlines the basic teleoperation information flow. Cartesian displacements of one device in this master/slave setup are used to command the force-feedback on the other device using a simple teleoperation control law. The position offset, scaling, and force-feedback gains of each device are individually configurable. In addition to the position, the devices also communicate their button state allowing disconnection of teleoperation for manual reconfiguration (*clutching*) of local offsets. A `CollaborativeControlForce` task connects and shares the current state (position, button states, desired forces) between the devices.

As a demonstration of our modular implementation, we use a simple control law. For the haptic devices, given Cartesian positions p_m, p_s, d_m, d_s, offsets o_m, o_s, displacements e_m, e_s, gains k_m, k_s, scale sc, and commanded forces f_m, f_s for the master and the slave respectively, we obtain:

$$d_m = p_m + o_m \tag{1}$$

$$d_s = p_s + o_s \tag{2}$$

$$e_m = d_m - sc * d_s \tag{3}$$

$$e_s = d_s - (-sc) * d_m \tag{4}$$

$$f_m = max(k_m * e_m, lim_m) \tag{5}$$

$$f_s = max(k_s * e_s, lim_s) \tag{6}$$

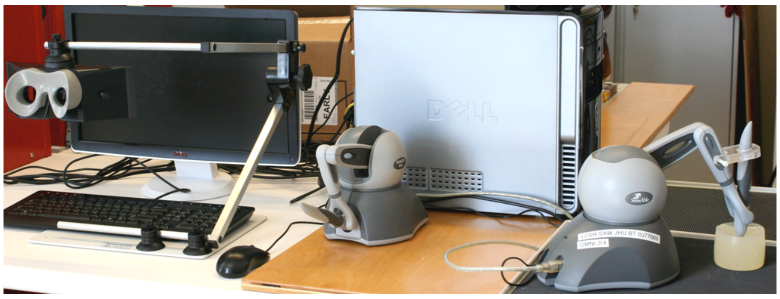
where lim_m, lim_s limit the maximum magnitude of the master and slave forces to be displayed, respectively. In the demonstrations below, we use constant gains.

To limit maximum displayed force, this implementation includes a *ratchet* effect, allowing the devices to be independently reconfigured if the applied force reaches the maximum force limits. The offsets are recomputed, and teleoperation resumes once the forces return below the maximum limits.

B. Experiments

Fig. 7 shows master and slave position tracking for a Sensable Omni master and slave using the above framework. Given the modular nature of our implementation, we can easily introduce local models for the remote device into our architecture to compensate for varying update rates, network latencies, and packet loss. In these experiments, we use simple Euler integration:

$$x(t) = x(t-1) + \delta t * \dot{x}(t-1) \tag{7}$$


(a) The Sensable Omni Master/Slave Setup with a Stereo Viewer

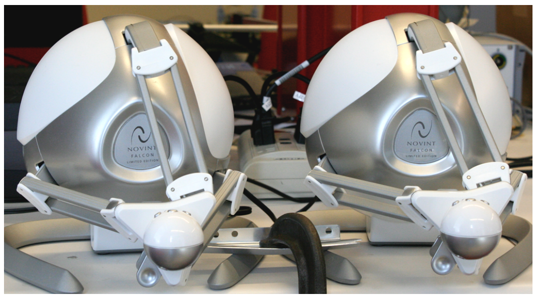

(b) The Novint Falcon

Fig. 5: The Devices Used in Our Experimental Setup

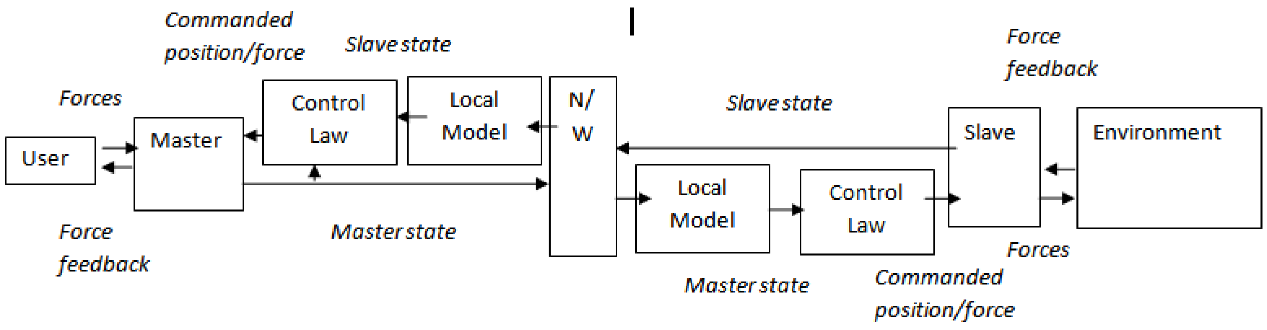


Fig. 6: Block Diagram of Teleoperation Architecture.

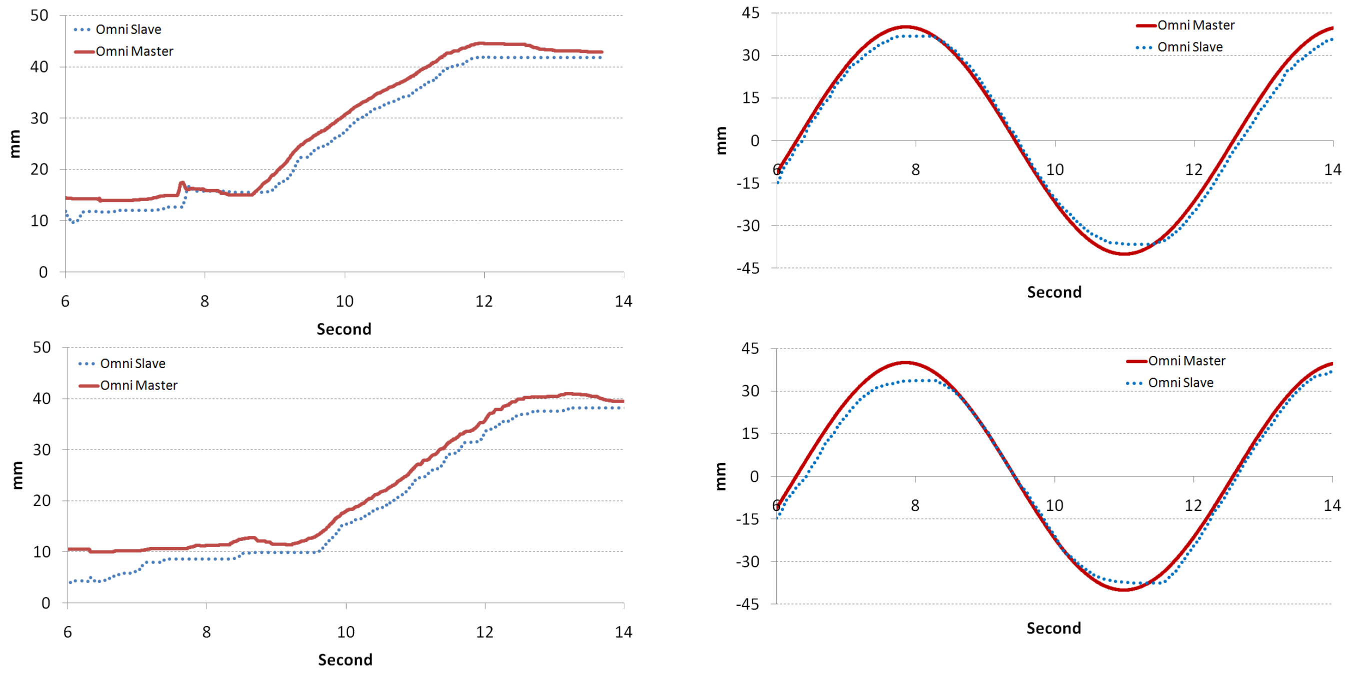


Fig. 7: Position tracking for Omni master and slave manipulators for line movement (left) and sinusoidal input (right).

Other controllers and local models can be integrated by replacing this simple model. We can teleoperate any combination of our haptic devices–two Omnis as master and slave, two Falcons as master and slave, a mixed configuration, or using the da Vinci master's as slaves over the da Vinci research API. The experimental framework allows for configurable task updates; here we used a 1.0ms task period.

To simulate the delay of our network layer, we created a `DelayTask`. This task delays the reading of positions from the two devices, simulating a delay that could potentially be created by a network layer. Using this task and simple Euler integration mentioned above, we ran two teleoperative experiments to visualize the effects of the delay and our efforts to correct for the errors introduced.

We tested the `CollaborativeControlForce` with delay under two circumstances: point-to-point motion and sinusoidal wave teleoperation. In the point to point case, we moved the master device manually and observed the slave device following. In the sinusoidal wave case, we fed the sinusoidal wave to the master device, and let the slave device follow. In both cases, the offset between the positions of the two devices (y-axis in Fig. 7) is negligible. This offset results from the initial repositioning of the devices to align, which is beyond the scope of this paper. However, the difference of the positions over time (x-axis in Fig. 7) significantly shows the delay of one device versus the other. In the sinusoidal wave case, we observed that with the introduction of the `DelayTask`, the slave device has an obvious delay with respect to the master. Subsequently, with the addition of the Euler integration, we observed a decrease in the delay of the slave device compared to the master.

III. Conclusions

This paper presented the SAW component-based framework and described the communication between component interfaces, which is identical for the single process (multi-threaded) and multi-process cases. While the single process case obviously produces the lowest latency, an efficient network layer based on ICE provides latencies less than a few hundred microseconds for optimal network topologies, such as single-hop wired Ethernet, and on the order of a few milliseconds for typical multi-hop wired/wireless networks. The former is suitable for high-bandwidth (kiloHertz) closed-loop control, whereas the latter is an acceptable haptic update rate for teleoperation tasks. The paper further presented a teleoperation application for robotic surgery systems based on the SAW framework, including implementations for common haptic devices and integration with the da Vinci surgical system. Experiments utilizing the da Vinci master manipulators are ongoing. We are also integrating the ability to share synchronized video and state over the network, as well as local models of state to overcome network latency and data loss.

IV. Acknowledgments

This work is supported in part by National Science Foundation EEC 9731748, EEC 0646678, and MRI 0722943.

References

[1] J. G. Greg Broten, Simon Monckton and J. Collier, "Software systems for robotics an applied research perspective," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 11–16, Nov 2006.

[2] A. Kapoor, A. Deguet, and P. Kazanzides, "Software components and frameworks for medical robot control," in *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, May 2006, pp. 3813–3818.

[3] Y. hsin Kuo and B. A. MacDonald, "A distributed real-time software framework for robotic applications," in *Proc. IEEE International Conference on Robotics and Automation ICRA 2005*, Apr. 18–22, 2005, pp. 1964–1969.

[4] T. H. Collett, B. A. MacDonald, and B. P. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proc. of the Australasian Conf. on Robotics and Automation (ACRA)*, Sydney, Australia, 2005.

[5] E. Woo, B. A. MacDonald, and F. Trépanier, "Distributed mobile robot application infrastructure," in *International Conference on Intelligent Robots and Systems (IROS)*, Las Vegas, October 2003, pp. 1475–80.

[6] Robot standards and reference architectures. [Online]. Available: http://www.robot-standards.org

[7] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *In Proceedings of the 11th International Conference on Advanced Robotics*, 2003, pp. 317–323.

[8] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the orocos project," in *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on*, vol. 2, Sept. 2003, pp. 2766–2771 vol.2.

[9] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Towards component-based robotics," in *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, Aug. 2005, pp. 163–168.

[10] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for robotics," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'06) Workshop on Robotic Standardization*, Dec 2006.

[11] M. Henning, "A new approach to object-oriented middleware," *Internet Computing, IEEE*, vol. 8, no. 1, pp. 66–75, Jan-Feb 2004.

[12] The surgical assistant workstation (saw). [Online]. Available: http://www.cisst.org/saw

[13] B. Vagvolgyi, S. DiMaio, A. Deguet, P. Kazanzides, R. Kumar, C. Hasser, and R. Taylor, "The Surgical Assistant Workstation: a software framework for telesurgical robotics research," in *MICCAI Workshop on Systems and Arch. for Computer Assisted Interventions*, Midas Journal, Sep 2008. [Online]. Available: http://hdl.handle.net/10380/1466

[14] S. P. DiMaio and C. J. Hasser, "The *da Vinci* research interface," in *MICCAI Workshop on Systems and Arch. for Computer Assisted Interventions*, Midas Journal, Sep 2008. [Online]. Available: http://hdl.handle.net/10380/1464

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

[16] M. Reggiani, M. Zuppini, and P. Fiorini, "A software framework for process control in the agroindustrial sector," in *Automation Science and Engineering, 2007. CASE 2007. IEEE International Conference on*, Sept. 2007, pp. 164–169.

[17] A. Bzostek, R. Kumar, N. Hata, O. Schorr, R. Kikinis, and R. H. Taylor, "Distributed modular computer-integrated surgical robotic systems: Implementation using modular software and networked systems," in *MICCAI '00: Proceedings of the Third International Conference on Medical Image Computing and Computer-Assisted Intervention*. London, UK: Springer-Verlag, 2000, pp. 969–978.

[18] O. Schorr, N. Hata, A. Bzostek, R. Kumar, C. Burghart, R. Taylor, and R. Kikinis, "Distributed modular computer-integrated surgical robotic systems: Architecture for intelligent object distribution," 10 2000.

[19] G. S. Broten, D. Mackay, S. P. Monckton, and J. Collier, "The robotics experience," *Robotics and Automation Magazine, IEEE*, vol. 16, no. 1, pp. 46–54, March 2009.