A Surgical Assistant Workstation (SAW) Application for a Teleoperated Surgical Robot System

Release 1.00

Min Yang Jung, Tian Xia, Anton Deguet, Rajesh Kumar, Russell Taylor, Peter Kazanzides

August 19, 2009

Johns Hopkins University, Baltimore, MD

Abstract

The Surgical Assistant Workstation (SAW) modular software framework provides integrated support for robotic devices, imaging sensors, and a visualization pipeline for rapid prototyping of telesurgical research systems. Subsystems of a telesurgical system, such as the master and slave robots and the visualization engine, form a distributed environment that requires an efficient inter-process communication (IPC) mechanism. This paper describes the extension of the component-based framework provided by the *cisst* libraries, on which the SAW depends, from a multi-threaded (local) to a multi-process (networked) environment. The current implementation uses the Internet Communication Engine (ICE) middleware package, but the design does not depend on ICE and can accomodate other middleware choices. A telesurgical robot system based on the da Vinci hardware platform is used as a test case, with a research application that requires coordination between the left and right robot arms (a bimanual knot tying task). Performance measurements on this platform indicate minimal overhead due to the networking.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3079]

Contents

1 Introduction

The Surgical Assistant Workstation (SAW) modular software framework provides integrated support for robotic devices, imaging sensors, and a visualization pipeline for rapid prototyping of telesurgical research systems [7]. It is based on the *cisst* libraries for computer-integrated surgical systems development [2]. The *cisst* libraries can be classified in three categories: 1) foundation libraries, 2) component-based framework, and 3) component implementations (see Figure 1). The SAW is an architectural framework that sits on top of *cisst*; as such, it draws heavily from the component-based framework and from the collection of implemented components, such as robots, collaborative robots (e.g., telesurgical robots), and a 3D user interface toolkit.

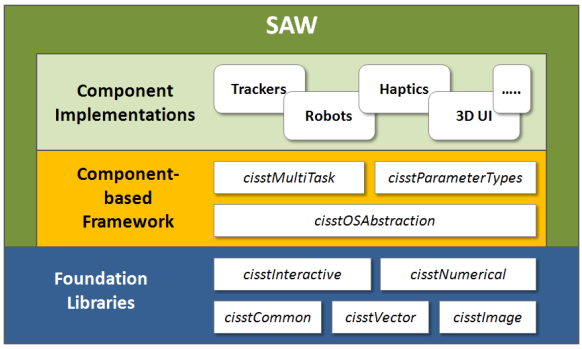


Figure 1: Overview of the *cisst* libraries

In [2], we reported a component-based framework, the *cisstMultiTask* library, where the components consisted of *devices* and *tasks*. A *device* is a passive component that does not have its own thread of control; it is typically used to provide a "wrapper" for a hardware device or an external software package. A *task* is derived from a *device*, but is an active component that contains its own thread of control. Both tasks and devices (henceforth we will use task to refer to either) may contain any number of *provided interfaces* and *required interfaces*. The tasks communicate with each other via these interfaces; specifically, each required interface is connected to a provided interface and information is exchanged via command objects (e.g., using the Command Pattern [3]). A simple example is shown in Figure 2

One major limitation of the framework reported last year was that it was only usable within a single process; that is, it was designed for safe and efficient data exchange in a multi-threaded environment within a single process. Clearly, this is a serious limitation for systems, such as telesurgical robots, that often require multiple computers. As a result, it was not possible to consistently employ the same inter-task communication mechanisms in such systems. Instead, researchers were forced to implement their own inter-process communication (IPC), often resorting to a custom-made socket-based protocol. Although use of a standard protocol, such as OpenIGTLink, would be better, it would still require researchers to employ different communication mechanisms, depending on whether or not the tasks were in the same process.

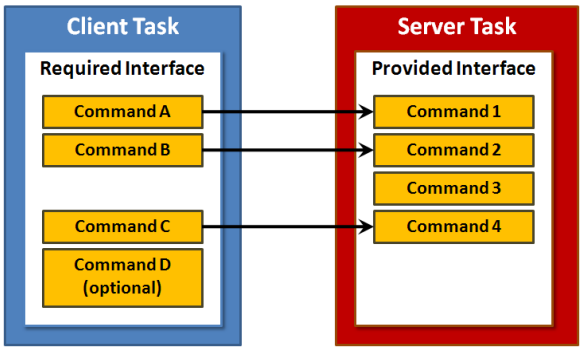


Figure 2: Illustration of component-based framework

Clearly, a toolkit that is designed to facilitate the development of computer assisted intervention systems must be distributed by nature. For code reuse, however, it is desirable to have an implementation that is independent of the topology. In this paper, we report on the development of a network interface for the *cisst* component framework that extends the current inter-task communication mechanism to the multi-process and multi-computer scenarios. The network interface currently uses the Internet Communications Engine (ICE) to provide the low-level functionality, but still preserves the existing inter-task programming model; in fact, other than some additional configuration, researchers will not see a difference in how they create their multi-task applications. It is important to note that the design does not depend on ICE – it is possible to replace ICE by any other package that provides the required features (even standard sockets would do, with a bit more programming effort). There are a plethora of middleware packages that could be considered as alternatives to ICE. Some examples include the Spread Toolkit, Data Distribution Service (DDS), and Common Object Request Broker Architecture (CORBA).

We chose ICE as our network middleware packages for two reasons. The first is that the basic concept of the ICE architecture is the Proxy Pattern [3, 6], which is also the main design concept of the new *cisstMultiTask* library (see Section 3.1). The second is that ICE provides the SLICE–Specification Language for ICE–which is conceptually similar to Common Object Request Broker Architecture (CORBA) Interface Description Language (IDL). SLICE provides flexible and extensible ways to define or modify data structures and interfaces that play a critical role to share information between components [1].

This paper uses a telesurgical robot system, with virtual fixtures, as an illustrative example. This system is described in the following section.

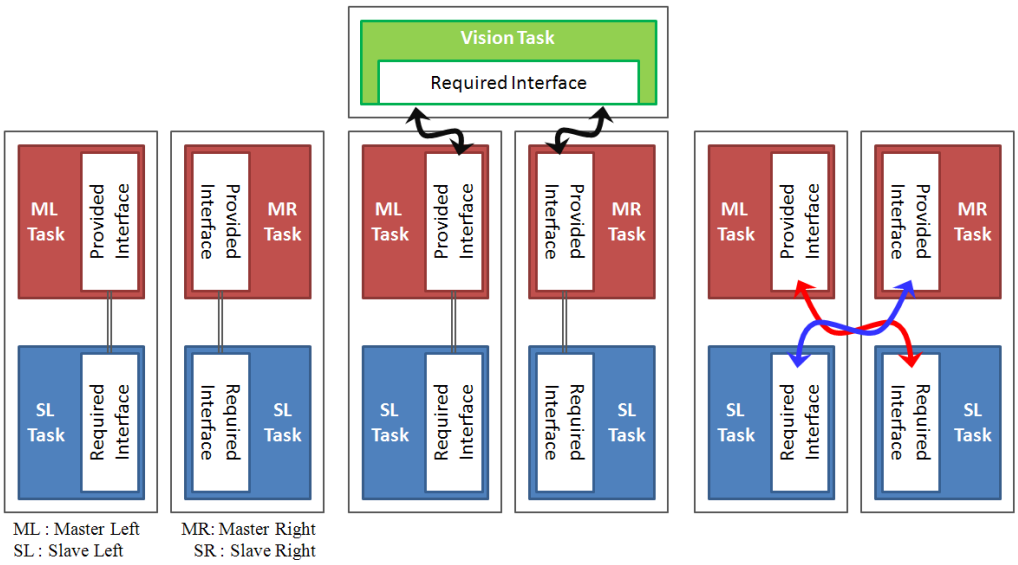
2 Overview of Telesurgical Robot System

The basic telesurgical robot system consists of two master/slave manipulator pairs, a stereo viewing console (all from Intuitive Surgical, Inc.), custom-designed motor controller/amplifier boards and control software based on the SAW application framework. Due to physical hardware limitations, one control workstation cannot host all the required controller/amplifier boards for the two master/slave manipulators pairs. Therefore, each master/slave pair is connected to one control workstation. For this teleoperation configuration, no networked communication is needed (see Figure 3a).

In other configurations where two masters are coordinated, or master/slave reside on different computers, networked communication between multiple computers is required, thus motivating the network extension to the inter-task communication model.

A specific example is provided by the Bimanual Knot Placement project, where we use virtual fixtures to provide guidance in performing the suture knot placement task using both master/slave teleoperated manipulators in the same workspace (see Figure 3b). This is an extension of earlier work that used a simpler experimental setup (Cartesian robots under cooperative control) to demonstrate the concept [5]. In the current system, the surgeon operates the master manipulators, and the slave manipulators follow the masters under certain motion optimizations, which are implemented with virtual constraints such that the correct sliding friction and knot motion trajectory are maintained to place the knot. This project requires coordination of both master manipulators and a stereo vision system to track the position of the knot and both slave tool tips, which must be fed back to the two master manipulators. In this project configuration, each master/slave manipulator pair is controlled by a workstation and both connect to the stereo vision system on another workstation via the network extension.

In the Swapped Slave configuration, the right master controls the left slave and vice versa. In this scenario,



(a) Setup 1 (b) Setup 2 (c) Setup 3

Figure 3: Configurations of the Telesurgical Robot System

a master must use network communication to transmit Cartesian frame information to the opposite slave, which is controlled by a different workstation (see Figure 3c).

In all these configurations, the control task implementation for each master and slave manipulator is independent of the topology. The only differences are during system configuration, where the specific required and provided interfaces are connected. The three configurations shown in Figure 3 have all been implemented and tested, showing the flexibility of the network extension.

3 Network Implementation of Inter-Task Communication

One of the key features of the *cisstMultiTask* library is to allow data exchange between tasks in a thread-safe and efficient way. The initial implementation focused on multithreading within a single process. Nevertheless, the key design concepts–component-based philosophy and self-describing interfaces–allow painless integration via the Proxy Pattern [3, 6] so that *cisstMultiTask* can support IPC.

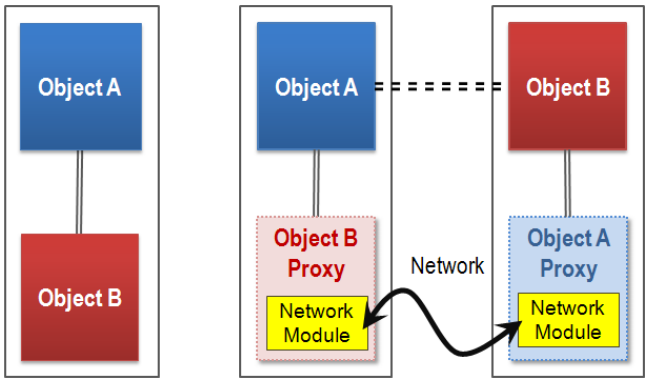
The low-level data communication between proxies is built upon ICE which provides a simple, flexible, light-weight but versatile network middleware. This low-level data communication layer has been abstracted so that the new *cisstMultiTask* library is loosely coupled with ICE, which means the overall design does not depend on ICE. It can be easily replaced with any other package, even native sockets, if they can support the abstraction layer appropriately.

3.1 Design Concept: Proxy Pattern

Using the Proxy Pattern, the *cisstMultiTask* library is able to handle proxy objects as it does the original objects. In Figure 4, object A is locally connected to object B. With the introduction of our proxy-based model, object A is *locally* connected to the object B *proxy* but is *practically* linked to object B across a

network. From object A's point of view, the peer remains the same–object B–in both cases. The major advantage of this design is that proxies can be generated automatically on top of the existing interfaces without any additional code in the interfaces themselves.

Data exchange between the actual objects is mediated by proxy objects that contain the network module–ICE in our case–for data communication. The module manages the low-level data communication layer which provides features for data transfer such as serialization, deserialization, session management, and connection management. This design enables the flexibility to choose alternate network middleware packages.



(a) Single process (b) Multiple proceeses

Figure 4: The concept of the Proxy Pattern in the *cisstMultiTask* library

3.2 Proxies in the *cisstMultiTask* library

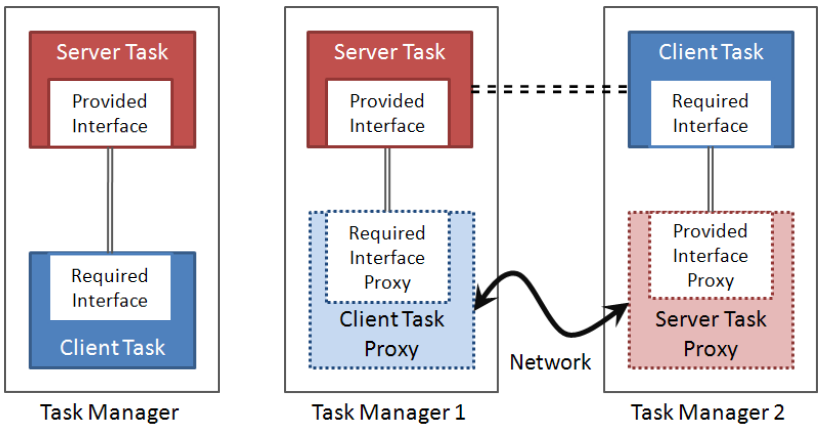
For efficient data exchange in a thread-safe way, the *cisstMultiTask* library uses several components such as *tasks*, *interfaces*, *commands*, and *events*. The implementation of IPC requires these components to be split across a network, which means several proxy classes should be defined as well. In order to increase code reusability and to manage proxy classes in a consistent way, we defined three base classes–`mtsProxyCommonBase`, `mtsProxyBaseClient`, and `mtsProxyBaseServer`–and then created the other derived classes. Table 1 shows the complete list of proxy classes defined in the *cisstMultiTask* library.

Note that these proxy classes are entirely hidden from the application layer and are created and managed internally. This design allows programmers to use any class or object derived from the *cisstMultiTask* base types over the network, thereby enabling network extension with minimal code changes (see Section 3.3).

3.3 Code-level Changes

Use of the Proxy Pattern allows *cisstMultiTask* library users to extend their single process application to a multi-process system that works across a network in a very simple and easy way. If there are two tasks–a server task and a client task–that run in the same process (see Figure 5), the core part of the application configuration code would be the following:

```
taskManager->AddTask(clientTask);
taskManager->AddTask(serverTask);
taskManager->Connect("clientTask", "requiredInterfaceName",
                     "serverTask", "providedInterfaceName");
```



(a) Single process (b) Multiple processes

Figure 5: The application of the Proxy Pattern in the *cisstMultiTask* library

If, however, these two tasks are distributed over a network, the configuration code would be as follows:

Server Process – Simply add the server task and make sure the task manager publishes it:

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

Client Process – Add the client task and connect to the server task via the global task manager:

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

To summarize, there is very little code overhead for application programmers because all proxy objects such as task proxies, interface proxies, command proxies, and event proxies are dynamically created and run internally. The only methods to be implemented are serialization and deserialization if the commands and events use data type that are not supported by the *cisst* package (serialization methods for vectors, matrices, frames, strings and common robot data types are already provided). For new data types, the implementation of serialization should be trivial because the library provides examples and auxiliary methods.

4 Performance Analysis for Telesurgical Robot System

Using the *cisstMultiTask* network extension, the conversion of a single process application to a multi-process application requires additional processing such as serialization, networking overhead (physical data transmission over a network), dynamic object creation, and deserialization. This naturally leads to the lowered performance of the overall system. Thus, it is important to perform experiments to understand how significantly these factors affect the overall performance of the system.

The core idea for testing is not to rely on synchronization between the client and server to measure the network overhead. Instead, we use a data object which goes from server to client and back to the server (we also have a benchmark which uses client to server and back to client). As our datatype carries a timestamp,

Class Type	Original Class	Proxy Class
Base	-	mtsProxyBaseCommon mtsProxyBaseServer mtsProxyBaseClient
Derived	mtsTaskManager	mtsTaskManagerProxy mtsTaskManagerProxyServer mtsTaskManagerProxyClient
Derived	mtsDevice	mtsDeviceProxy
Derived	mtsDeviceInterface	mtsDeviceInterfaceProxy mtsDeviceInterfaceProxyServer mtsDeviceInterfaceProxyClient
Derived	mtsCommandVoid mtsCommandWrite mtsCommandRead mtsCommandQualifiedRead	mtsCommandVoidProxy mtsCommandWriteProxy mtsCommandReadProxy mtsCommandQualifiedReadProxy
Derived	mtsMulticastCommandWrite	mtsMulticastCommandWriteProxy

Table 1: List of Proxy Objects in the *cisstMultiTask* Library

we can compare the time it took with and without the middleware. By subtracting, we get an estimate of the network overhead, including serialization, dynamic creation and de-serialization.

For the communication from server to client and back to server (SCS), we first use a Read command. Since the Read command uses a circular buffer for thread safety, the client receives an object which is no older than one period plus the transmission time. The client then uses a Write command to send back to the server. This command is queued and executed at the beginning of the next period. The total time elapsed should be lower than 2 periods (assuming both client and server use the same period and are sufficiently synchronized).

For the communication from client to server and back to client (CSC), we first use a Write command. As in the previous example, the command is queued. When the command is executed, it sends the data back using an event. The event is queued on the client side. Again, the total time elapsed should be lower than 2 periods.

The goal of this experiment is not to compute the total loop time but to estimate the overhead introduced by the middleware and the required serialization, dynamic creation and de-serialization. For our experiments, we relied on the *cisst* data type `prmPositionCartesianGet` which contains a rotation matrix, position vector (all doubles), a timestamp, a valid flag and a couple more fields for a total of 120 bytes.

Table 2 shows our experiment results using two machines (Ubuntu 8.0.4 with RTAI installed, kernel 2.6.24-16-rtai i686 GNU/Linux, Pentium 4 3.2GHz, 1G RAM).

For both SCS and CSC tests, Linux with RTAI performed better than Linux without RTAI, as expected. The most notable thing here is that the average elapsed time is small (less than 1 msec), even in the worst case (SCS testing, 0.9229 msec under Linux with RTAI disabled). This implies that the overhead due to the introduction of networking is minimal.

The total overhead can be separated into three parts: *preparation and recovery overhead* (serialization/dynamic creation/deserialization), *transmission overhead* (networking that includes middleware layer processing), and *execution overhead* (pointer casting/dereferencing). The preparation and recovery overhead were measured using an independent test program that repeats a testing loop 10000 times that serializes a variable, dynamically creates it, and deserializes it. The results–0.004239 msec and 0.0043557 msec for Linux with

OS	Loop	Category	Avg (ms)	Std (ms)	Min (ms)	Max (ms)
Linux (RTAI)	SCS	Local	0.992913	1.72807	0	4.00046
		Network	1.19414	1.83933	0	20.0023
		Difference	**0.201227**			
	CSC	Local	0.996914	1.7304	0	4.00046
		Network	1.22542	0.523384	0.303618	19.1435
		Difference	**0.228506**			
Serialization+Dynamic creation+Deserialization			0.004239 ms per sample			
Linux	SCS	Local	4.00446	0.178861	4.00045	16.0018
		Network	4.92736	1.72445	4.00045	36.0041
		Difference	**0.9229**			
	CSC	Local	4.25248	0.993137	4.00045	20.0023
		Network	4.88347	1.82513	3.04369	32.8634
		Difference	**0.63099**			
Serialization+Dynamic creation+Deserialization			0.0043557 ms per sample			

Table 2: The overhead measurement results

and without RTAI, respectively–show that this overhead is negligible. Similarly, the execution overhead is also very small. Thus, the transmission overhead accounts for the largest part of the total overhead, which is still relatively small. This low overhead for ICE has already been extensively tested and examined [4].

To summarize, the experiment results show that users can extend a single-process application to a multi-process system with just a small sacrifice in performance.

5 Conclusions

We created a component-based framework for the development of computer-assisted intervention (CAI) systems, focusing initially on achieving high-performance multi-tasking by implementing all tasks as threads within a single process. The component model consists of command objects in provided interfaces that are bound to corresponding objects in required interfaces. Conceptually, the task with the provided interface can be considered the server for the task with the required interface (the client), although a task can have both types of interfaces (e.g., acting as both a client and server in a hierarchical structure). This component model has proven useful for the creation of several CAI systems, but the limitation to a single process has been problematic. A particularly compelling example is a telesurgical robot system which, due to hardware constraints, consists of several robotic arms connected to different computer workstations. Thus, we extended our existing component model to a network configuration by creating proxy objects for the key elements of our component-based framework – specifically, for the task manager, tasks (and devices), interfaces, and commands. We used the Internet Communication Engine (ICE) to provide these proxies, though the design allows other middleware packages to be used. We tested the software by implementing it on the telesurgical robot system described above. Performance results indicate minimal overhead (less than 1 msec) due to the network implementation.

The SAW framework and underlying *cisst* libraries are being released under an open source license. Currently, most relevant portions of the *cisst* libraries (i.e., the foundation libraries and component framework illustrated in Fig. 1) are available at trac.lcsr.jhu.edu/cisst. Current work is focused on increasing the set of

implemented components that enable creation of different CAI systems within the SAW framework.

Acknowledgments

The *cisst* libraries have been created by many individuals; primary contributors include Ankur Kapoor, Ofri Sadowsky, Balazs Vagvolgyi, and Daniel Li. We acknowledge the contributions of Simon DiMaio, Chris Hasser and colleagues at Intuitive Surgical, Inc. towards the SAW framework. This work is supported in part by National Science Foundation EEC 9731748, EEC 0646678, and MRI 0722943.

References

[1] G. S. Broten, D. Mackay, S. P. Monckton, and J. Collier. The robotics experience. *Robotics and Automation Magazine, IEEE*, 16(1):46–54, March 2009. 1

[2] A. Deguet, R. Kumar, R. Taylor, and P. Kazanzides. The *cisst* libraries for computer assisted intervention systems. In *MICCAI Workshop on Systems and Arch. for Computer Assisted Interventions*, Midas Journal: http://hdl.handle.net/10380/1465, Sep 2008. 1, 1

[3] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. 1, 1, 3

[4] M. Henning. Choosing middleware: Why performance and scalability do (and do not) matter, Feb 2009. http://www.zeroc.com/articles/IcePerformanceWhitePaper.pdf. 4

[5] A. Kapoor and R. H. Taylor. A constrained optimization approach to virtual fixtures for multi-handed tasks. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 3401–3406, May 2008. 2

[6] M. Shapiro. Structure and encapsulation in distributed systems : The proxy principle. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 198–204, 1986. 1, 3

[7] 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: http://hdl.handle.net/10380/1466, Sep 2008. 1