A Component-based Architecture for Flexible Integration of Robotic Systems

Min Yang Jung, Anton Deguet, and Peter Kazanzides

***Abstract*— While a robot control framework generally focuses on real-time performance and efficient data exchange between cooperating tasks or processes, an application such as robot-assisted surgery often demands information from, and integration with, a number of other devices. Thus, the software framework for the *integrated system* may have different requirements and priorities than a framework for real-time robot control. This paper reports on a component-based architecture that seamlessly bridges the gap between real-time robot control and a distributed, integrated system. The starting point is the *cisst* library, which provides a component-based framework for lock-free and efficient data exchange between multiple threads within a single process, which is suitable for real-time robot control. This paper describes the extension of the *cisst* library to support distributed systems, while keeping the same programming model as the single-process, multi-threaded scenario. Thus, application software does not need to know whether the component providing services is within the same process, in a different process, or on a different computer. In comparison, most standard middleware packages support components that fall within the last two categories (different processes on the same computer or different computers). This does not allow them to take advantage of the higher performance that can be achieved using standard lock-free data structures that do not rely on the operating system or on middleware services. Thus, the novelty of this approach is that the same component-based architecture and associated programming model extends from a multi-threaded scenario (which provides the best real-time performance) to a standard multi-process distributed system.**

I. Introduction

As robots become more prevalent in society, there is a need for integration with other devices. This is especially true in the area of medicine, where several robot systems have already been introduced into the operating room, but still function primarily in a "stand-alone" configuration with proprietary communication protocols and limited interaction with other medical devices.

We believe that future medical robots will need to integrate with a range of external sensors and imaging devices and utilize both pre-operative and intra-operative information. To address these needs and enable rapid prototyping of new applications for robot-assisted surgery, we are developing the Surgical Assistant Workstation (SAW) software framework[1], [2]. 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 [3] 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.

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

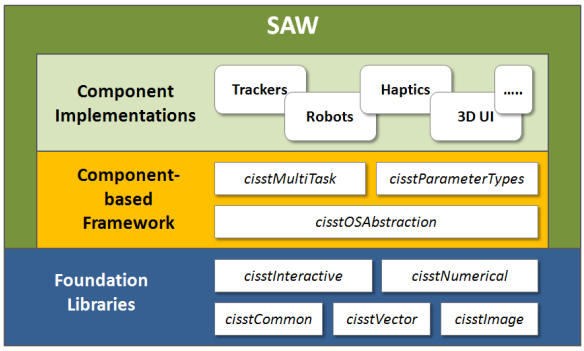


Fig. 1: SAW Components

The SAW framework is based on the *cisst* libraries [4] (see Fig. 1). This paper presents the underlying component-based framework in *cisst* and introduces an extension of this framework from multi-threaded (single process) systems to multi-process and multi-host systems via a network layer. The network layer is currently implemented using the Internet Communications Engine (ICE), but we include an abstraction layer that enables other network middleware to be used.

The key point of our architecture is that it is designed not only for real-time robot control, but also for the integration of robots into larger, more complex systems. Thus, the architecture satisfies real-time performance requirements while also enabling the use of conventional operating systems and devices. Furthermore, the same component-based programming model is used regardless of whether the programmer is implementing hard-real-time robot control with a real-time operating system (possibly with multiple threads in a single process), or data processing (e.g., capturing and processing video streams) with a conventional operating system such as Microsoft Windows, or both.

II. Related Work

A number of architectures, frameworks, and packages to support efficient development of robot control systems have been proposed and developed. Although many of them have adopted a component-based architecture with the goal of achieving software reusability, their overall designs differ, often due to the desire to efficiently support a particular project or environment. Several reviews of such systems for robotics research have already been published [5], [6], [7], [8],

[9], [10], and there is a web-site dedicated to cataloguing and reviewing these systems as well as standardizing a reference architecture for robotics [11].

Several commonly used frameworks for recent robotics research include Player, OROCOS, Orca and ROS.

***Player*:** Player [12] is a popular set of tools for mobile robot 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) [13] was started in 2001 to develop open source robot control software. It 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*:** Branching from the OROCOS project, the Orca Project [14], [15] aims to provide building-blocks (components) that can be combined together to build arbitrarily complex non-real-time robotic systems. It uses the Internet Communication Engine (ICE) [16] as network middleware.

***ROS*:** Willow Garage's Robot Operating System (ROS) [17] is a more recent entry to the field, but has undergone rapid development. It is an open-source package that provides operating system types of services, such as hardware abstraction, low-level device control, and communication between processes, as well as numerous tools to facilitate development. The intent is to create a common platform upon which researchers can build, and then share, high-level robotic algorithms in areas such as navigation, localization, planning, and manipulation.

Although the above frameworks share a component-based design philosophy, each has its own characteristics and architecture. The next sections summarize the *cisst* component framework and introduce the network layer; these are followed by a discussion of the differences between *cisst* and the other software packages.

III. The *cisst* Component Framework

The *cisst* package [18] is a collection of open-source C++ software libraries that are designed to ease the development of computer-assisted intervention systems. It consists of two layers, *foundation libraries* and *component framework*, as in Fig. 2. According to [19], "a component framework

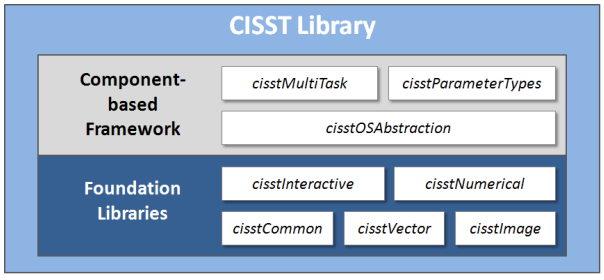


Fig. 2: The *cisst* Library

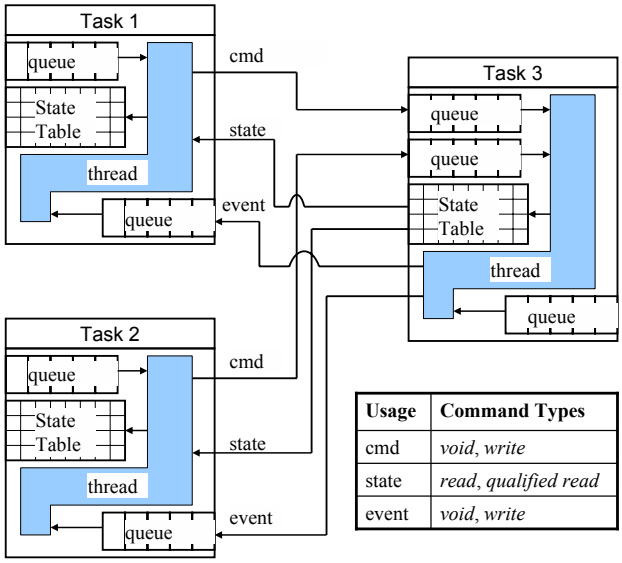


Fig. 3: Communication between tasks

is a skeleton of a component implementation that can be specialized by a component developer to produce custom components." In the *cisst* package, the component framework is primarily provided by the *cisstMultiTask* library; specifically, this library defines a base component class, `mtsComponent`, and a number of derived component frameworks that facilitate creation of components with active execution models, such as `mtsTaskPeriodic`, `mtsTaskContinuous`, `mtsTaskFromCallback`, and `mtsTaskFromSignal`. All of these component frameworks contain a list of *provided interfaces* and *required interfaces*. The "task" components contain a thread and therefore also contain elements, such as a *state table* and a set of input queues, that support efficient, lock-free and thread-safe data exchange, as shown in Fig. 3 [20]. The state table is a circular (ring) buffer that stores a time history of the "important" data in a task; this data can then be read by other components. The state table thus has a single writer and potentially many readers, and thread-safety can be assured without the use of a mutex (lock). The input queues are used to receive commands or events from other components. A separate queue is created for each connected component, so that each queue has only a single writer and a single reader, thus also guaranteeing lock-free thread-safety.

The system defines a component manager that is used to control components (e.g., start, stop) and to create connections between their provided and required interfaces. For example, the required interface of component A can be connected to the provided interface of component B, as shown in Fig. 4. In this context, it is convenient to refer to component A as the *client* and component B as the *server*, though it should be noted that every component can have provided and required interfaces and therefore act as both client and server.

Each provided interface contains a number of command objects that encapsulate the available *services*, which generally are implemented by class member functions of the server component. Command objects are strongly typed, and can

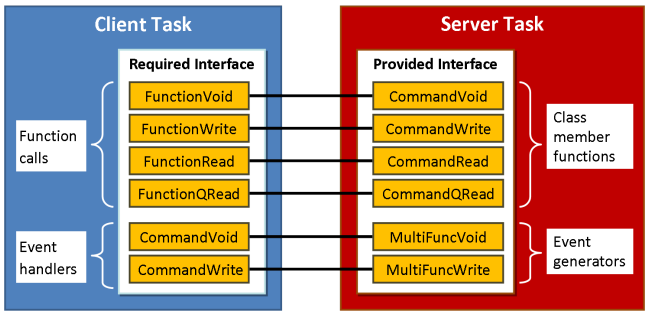


Fig. 4: Client-server components showing command and event objects

be one of four types: void, write, read, and qualified read (qualified read has two parameters: one input, the *qualifier*, that specifies the data to be read, and one output for the returned data). Note, however, that the command object also encapsulates the data exchange mechanism. For example, in most cases executing a void or write command object (in the client thread) does not result in an immediate call to the member function in the server, but rather causes the request to be queued for later execution by the server task, within its own thread, as shown in Fig. 3. This ability to encapsulate the data exchange mechanism in the command object also enables the "behind the scenes" implementation of a network interface, as described in the subsequent sections. The provided interface may also generate events, either with a payload (write event) or without (void event). The client task can use command objects to specify the *event handlers* for these events.

The client's required interface contains a list of command pointers that are set to point to the corresponding command objects in the server's provided interface. For convenience, the *cisstMultiTask* library defines a set of function classes (`mtsFunctionVoid`, `mtsFunctionWrite`, `mtsFunctionRead`, `mtsFunctionQualifiedRead`) that encapsulate the command pointer and define an overloaded function call (parentheses) operator. This enables a more intuitive syntax for invoking a command object; i.e., it looks like a regular function call. The binding of function objects to command objects is done by comparing string names. When the required interface is connected to a provided interface, the system goes through the list of function objects (in the required interface) and looks for a command object (in the provided interface) that has the same name. This string comparison is only performed during the initial connection; afterwards, the client task just executes the command object (via the bound function object).

IV. Design

When extending the current data exchange mechanism of the *cisstMultiTask* library to support inter-process communication (IPC), we set up design requirements as follows:

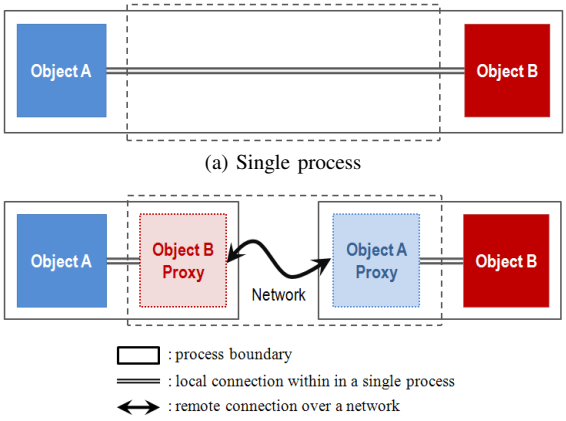
- Existing code should compile with little or no code-level modification (backward compatibility)
- Users should require minimal code-level changes to distribute their components over a network
- Maintain the current programming model so that application software does not need know the system configuration
- Support all data exchange mechanisms transparently

Two approaches that we take to meet these requirements are the Proxy Pattern[21] and the abstraction of the network interface layer. In the following section, we describe details of the two approaches and show how we chose networking middleware and how minimal code-level changes are required to distribute components across networks.

A. Proxy Pattern

In the *cisstMultiTask* library, data communication between components occurs through a pair of connected interfaces (provided interface and required interface), so the starting point to distribute components over a network is to split this connection. However, the design goals require that the current data exchange mechanism should be preserved and be completely supported. To satisfy these two conflicting goals, we adopted the Proxy Pattern.

1) Basic Concept: When two objects–*Object A* and *Object B*–are locally connected to each other (i.e., running within a single process) as in Fig. 5a, the basic concept of the Proxy Pattern is to replace the *local* connection between them by a *logical 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. 5b. Note that a proxy object is always local to its original peer object and the original local connection remains unchanged from the original objects' point of view.



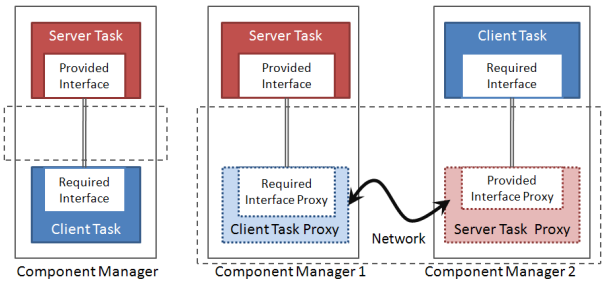
(a) Single process

(b) Multiple processes

Fig. 5: The concept of the Proxy Pattern

2) Application to the cisstMultiTask *library:* If this proxy concept is applied to two tasks connected to each other as in Fig. 6a, the local connection can be logically extended across a network using *task proxies* and two types of interface proxies (*provided interface proxy* and *required interface proxy*). This is shown in Fig. 6b.

This logical extension of a local connection needs further proxy objects to be introduced because the fundamental data exchange mechanism in *cisst* is based on the Command



(a) Single process (b) Multiple processes

Fig. 6: The Proxy Pattern in the *cisstMultiTask* library

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

Original Class	Proxy Class

(*Proxy Base Classes*)	`mtsProxyBaseCommon`
	`mtsProxyBaseServer`
	`mtsProxyBaseClient`

`mtsManagerGlobal`	`mtsManagerProxy`
	`mtsManagerProxyServer`
	`mtsManagerProxyClient`

`mtsComponent`	`mtsComponentProxy`

`mtsComponentInterface`	`mtsComponentInterfaceProxy`
	`mtsComponentInterfaceProxyServer`
	`mtsComponentInterfaceProxyClient`

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

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

`mtsMulticastCommandWrite`	`mtsMulticastCommandWriteProxy`

Pattern. Thus, it is necessary to create proxy objects for the different command types. Table I shows a complete list of internal objects that are pertinent to the logical extension of a connection across a network. All of these proxy objects are completely hidden from the application layer (i.e., library user) and they are all dynamically created and managed internally.

An example of configuration with two processes is shown in Fig. 7. Process `P1` has two components, `C1` and `C2`. `C1` has two required interfaces, `r1` and `r2`, and `C2` has one required interface, `r1`, and one provided interface, `p1`. Process `P2` has two components, `C2` and `C3`. `C2` has two provided interfaces, `p1` and `p2`, and one required interface, `r1`. `C3` has one required interface, `r1`. These components and interfaces are connected as follows:

```
Connect(P1, C1, r1, P2, C2, p1)
Connect(P1, C1, r2, P2, C2, p2)
Connect(P1, C2, r1, P2, C2, p2)
Connect(P2, C3, r1, P2, C2, p2)
```

The first three connections are remote and involve creation of proxies, whereas the last one is local and does not need any proxy. To establish these connections, the component proxy `P2C2onP1` is created in `P1` and two component proxies, `P1C1onP2` and `P1C2onP2`, are generated in `P2`. Furthermore, interface proxies are created for each component proxy as shown in the figure and command/event/function proxies are also created for each interface proxy as needed (not shown in this figure). Note, however, that from each component's point of view, all of these connections are *logically* local connections.

B. Abstraction of Network Interface Layer

We defined three base classes to encapsulate all the implementation details for networking and make adding a new proxy type simple and systematic. When a new network proxy object is to be added, we can just make it inherit from either `mtsProxyBaseServer` or `mtsProxyBaseClient`, depending on its role in networking. As an example, proxies of `mtsManagerGlobal`–which mediates data communication between component managers–and proxies of `mtsComponentInterface`–which handles data exchange between interfaces–are derived from those two base classes and thus implemented in a very similar way without any additional code for networking. With these base classes, we could not only significantly increase code reusability, but also manage proxy objects in a more consistent manner. Furthermore, this design allows the *cisstMultiTask* library to be independent from a specific network middleware. Currently, the library uses ICE but is not dependent on it. If we redefine or modify the proxy base classes, it is possible to substitute another middleware package–even a native socket-based implementation.

C. Networking Middleware Selection

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 overhead for data exchange between tasks, it naturally affects the overall performance of a system. From our own review and previous studies on a robotics system with middleware [22], [15], [6], we concluded that ICE best satisfied our design requirements for the following reasons:

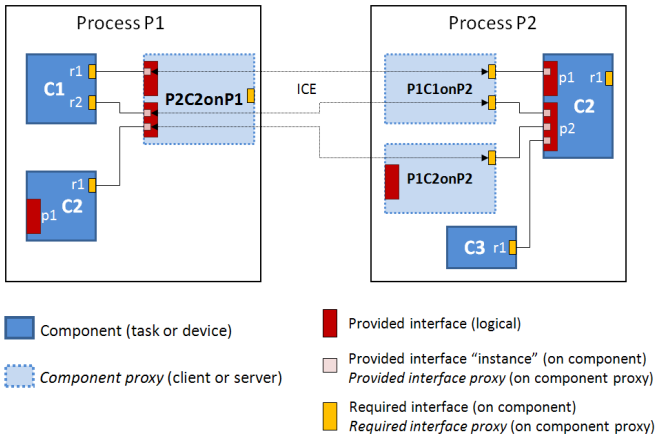


Fig. 7: A Sample Configuration

- **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 [23], SLICE can be a useful tool for designing the network layer.
- **High network performance**: As reported by ZeroC, ICE is known to perform efficiently in terms of latency and throughput [24].
- **Proxy Pattern**: The key concept of ICE is the Proxy Pattern, which is also the basic approach that we take. Because of this common pattern usage, the overall structure and implementation of the networking modules can be more consistent and simplified.

D. Code Changes

The proxy-based design requires minimal code changes to convert a standalone application to a network version by encapsulating proxies inside the *cisstMultiTask* library and hiding them from users. To illustrate, consider a case where two components are running in the same process. A `controlTask` controls a robot and offers a provided interface named `prvIntName` and a `UITask` has a required interface named `reqIntName`. The library includes a *Local Component Manager (LCM)* which is implemented as a Singleton object. An application registers its components to the LCM and uses it to manage them (e.g. create/start/stop/kill) and establish connections between them. In this case, the core implementation at the code-level would be the following:

```
manager = mtsComponentManager::GetInstance();
manager->AddComponent(controlTask);
manager->AddComponent(UITask);
manager->Connect(
    "UITask", "reqIntName",
    "controlTask", "prvIntName");
```

The conversion procedure to distribute the components–i.e., conversion from a standalone implementation (multi-threaded in a single process) to a networked implementation (multiple processes)–consists of three simple steps:

1) Execute the *Global Component Manager (GCM)*: The GCM is a centralized component that manages all LCMs in a system and provides several useful features such as cataloguing a list of LCMs connected or inspecting contents of components. Conceptually, the basic functionality that GCM does is similar to the CORBA naming service but it can provide additional features such as real-time data visualization or centralized data collection across the network. The *cisstMultiTask* library provides two different versions of the GCM–console-based application and GUI application–as a standalone utility.

```
$ GlobalComponentManager // execute GCM
```

2) Define Server Component: Set up the LCM and add server components to it. In the case of a networked LCM, two arguments are required: the IP address of the GCM and the name of the LCM. Then, proxy objects are internally created and the LCM is registered to the GCM.

```
manager = mtsComponentManager::GetInstance(
  "192.168.0.101", "ServerProcess");
manager->AddComponent(controlTask); // as before
```

3) Define Client Component and Connect: Set up the LCM and add client components to it. After creating a LCM instance as above, the LCM requests connection between two components across a network. If connection is established successfully, other proxy objects are created and logically connect the two components across the network.

```
manager = mtsComponentManager::GetInstance(
  "192.168.0.101", "ClientProcess");
manager->AddComponent(UITask); // as before
manager->Connect(
  "ClientProcess", "UITask", "reqIntName",
  "ServerProcess", "controlTask", "prvIntName");
```

From our experience converting several projects from standalone applications to distributed applications, the conversion process has proven to be simple and easy enough to be done in several minutes.

V. Comparison with Other Packages

Robot control packages have their own characteristics, architecture, and major target environments. The *cisst* library does too, but is a relatively general-purpose and flexible framework compared to the other robot control packages mentioned earlier. In this section, we compare the *cisst* library with other packages–Player, Orocos, Ora, ROS–and review major differences in terms of system characteristics.

Networking Layer: All packages reviewed support a distributed system via either external network middleware or internal networking layer. The *cisst* library (ICE), Orca (ICE), and Orocos (CORBA ACE TAO) fall into the first category and Player and ROS fall in the second category.

Interface Description Language (IDL): Since all packages adopt a component-based software engineering philosophy, they need an *interface* that defines data communication between distributed components. This interface description is tightly coupled with the networking layer and there are two basic approaches. The *cisst* library (SLICE), Orca (SLICE), and Orocos (CORBA IDL) use the external package's IDL while Player (configuration file) and ROS (message definition (`.msg`) file) use their own mechanism. The *cisst* library uses the IDL internally (e.g., to create the proxy classes), but relies on standard type libraries (e.g., `cisstParameterTypes`) to define necessary services such as serialization and deserialization. Note that *cisst* supports a complete set of component interface objects–four commands, two events–across networks, whereas Orocos does not yet fully support its component interfaces (the event interface is not available) and permits only primitive C++ types and `std::vector<double>` to be used across networks.

Operating System Support: The *cisst* library is truly platform independent and includes an operating system abstraction library (`cisstOSAbstraction`) that supports

Windows, Linux, Mac OS X, real-time Linux variants (e.g., RTAI), and QNX. The other packages are targeting Unix-based platforms and have partial or no support for Windows (Orocos has just recently provided Win32 support). Cross-platform support is important for robot-assisted surgery applications (the main application domain for *cisst*) because there often is a constraint to use a particular operating system, such as Windows, to enable integration of hardware devices supported only on that platform.

Real-time Support: The *cisst* library and Orocos support hard real-time while Player, Orca, and ROS do not focus on real-time performance. In case of ROS, however, it can be extended for real-time robot control, as in the Willow Garage PR2 robot, and has been integrated with the Orocos Real-Time Toolkit (RTT).

System Configuration: While Player, Orca, and ROS are based on inter-process communication (IPC), the *cisst* library and Orocos have support for inter-thread communication (ITC) within a single process as well as IPC. In case of the *cisst* library, the programming interfaces (API) and internal data exchange mechanism are identical in both cases.

These comparisons are summarized in Table II.

TABLE II: Comparison of Robotic Software Packages

	OS Support		RTOS [1]	Configuration[2]		
	Windows	Linux		MT	MP	MH
cisst	✓	✓	✓	✓	✓	✓
Player	△	✓			✓	✓
OROCOS	✓	✓	✓	✓	✓	✓
Orca	△	✓	△		✓	✓
ROS	△	✓	△		✓	✓

[1] means support for *hard real-time* (not just *runnable* on RTOS)
[2] MT: multi-thread, MP: multi-process, MH: multi-host
✓: fully supported, △: partial support

VI. Conclusions

We presented the *cisst* library focusing on its component-based framework and network layer. The library supports various platforms and the architecture of the library enables integration of real-time and non-real-time components using the same component-based programming model. Currently, the SAW software framework, and underlying *cisst* library, are used to develop several robot-assisted surgical applications and we anticipate that it will be used for more diverse domains that can benefit from its flexible architecture and characteristics.

VII. Acknowledgments

Russell Taylor, Director of the CISST ERC, provided the impetus for creating the *cisst* libraries and the SAW framework and many individuals have contributed to both. Ankur Kapoor did the initial development of the component framework and operating system abstraction library. This work is supported in part by National Science Foundation EEC 9731748, EEC 0646678, and MRI 0722943.

References

[1] The Surgical Assistant Workstation (SAW). [Online]. Available: http://www.cisst.org/saw

[2] 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

[3] 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

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

[5] T. Niemueller, "Developing a behavior engine for the Fawkes robot-control software and its adaptation to the humanoid platform Nao," Master's Thesis, RWTH Aachen University, Knowledge-Based Systems Group, April 2009.

[6] G. Broten, S. Monckton, J. Giesbrecht, and J. Collier, "Software systems for robotics: An applied research perspective," *Intl. Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 11–16, Nov 2006.

[7] A. Kapoor, A. Deguet, and P. Kazanzides, "Software components and frameworks for medical robot control," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2006, pp. 3813–3818.

[8] Y. hsin Kuo and B. MacDonald, "A distributed real-time software framework for robotic applications," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, Apr 2005, pp. 1964 – 1969.

[9] 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.

[10] E. Woo, B. A. MacDonald, and F. Trépanier, "Distributed mobile robot application infrastructure," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, Las Vegas, Oct 2003, pp. 1475–80.

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

[12] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage project: Tools for multi-robot and distributed sensor systems," in *Proc. Intl. Conf. on Advanced Robotics (ICAR)*, 2003, pp. 317–323.

[13] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the Orocos project," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, vol. 2, Sep 2003, pp. 2766–2771 vol.2.

[14] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Towards component-based robotics," in *Intl. Conf. on Intelligent Robots and Systems (IROS)*, Aug 2005, pp. 163–168.

[15] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for robotics," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), Workshop on Robotic Standardization*, Dec 2006.

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

[17] 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*, 2009.

[18] The cisst libraries. [Online]. Available: http://www.cisst.org/cisst

[19] D. Brugali and P. Scandurra, "Component-based robotic engineering (Part I)," *IEEE Robotics & Automation Magazine*, pp. 84–96, Dec 2009.

[20] P. Kazanzides, A. Deguet, and A. Kapoor, "An architecture for safe and efficient multi-threaded robot software," in *IEEE Conf. on Technologies for Practical Robot Applications (TePRA)*, Nov 2008, pp. 89–93.

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

[22] M. Reggiani, M. Zuppini, and P. Fiorini, "A software framework for process control in the agroindustrial sector," in *IEEE Intl. Conf. on Automation Science and Engineering (CASE)*, Sep 2007, pp. 164–169.

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

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