A Distributed Software Framework for Robotic Surgery

Release 0.00

Christoph Staub[1], Yu Ning[2], Salman Can[1], and Alois Knoll[1]

August 1, 2011

[1]{staub | cans | knoll} at in.tum.de
Technische Universität München, Robotics and Embedded Systems
[2]ning at jhu.edu
Johns Hopkins University, Department of Computer Science

Abstract

The ARAMIS research platform is a telesurgical robotic system for minimally invasive surgery with focus on autonomous functionality. The original software architecture was a hierarchical one, based on the model-view-controller paradigm. To handle the growing number of devices, we introduce a new framework that facilitates the decomposition of system functionality into separate programs, as well as the data sharing between them. This allows us to integrate and test new functionality more quickly. Our work heavily utilizes the *cisst* libraries, developed by the Johns Hopkins University. In particular, we take advantage of cisst's multiprocess networking capabilities. As a proof of concept, we demonstrate the integration of an eye-tracking based endoscope control.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/1338]

Contents

1 Introduction

The active research on medical robotics and telemanipulation systems raises the demand on flexible, distributable and scalable software frameworks. In particular, A framework with quick extensibility can facilitate the work of researchers, since the diversity of research, such as control strategies, image analysis and understanding, or medical workflow recovery, often requires the integration of new hard- and software components into an existing system. This article presents such a framework, through which we transform our software used to date into a newly designed, distributed system, based on open-source components.

The hardware of our research platform is briefly described in section 2, what follows is an overview of the the previous version of the system's software architecture and the needs that yielded a restructuring of the software. The original software was mostly based on the model-view-controller paradigm. This hierarchical approach was inspired by the modular design of the system and maps the individual hardware components to dedicated components of the software. For instance, three classes have been implemented in order to interact with the robots. The model class contains all functionality for on- and off line interaction with the robot (hardware abstraction layer (HAL)). The view class implements an OpenGL 3D model of the robot for visualization and collision detection, which is compliant with the OpenInventor standard, holding a reference to the model to update joint angles according to a timer. Finally, the controller class realizes a Qt user interface, which provides context menus for robot interaction, such as manually alteration of joint angles. As the model does not have any explicit references to neither the view nor the controller, the system can be run independently without any UI. Every manipulator, instrument, or control loop added to the system provides its own thread to guarantee the necessary real-time response time. In order to interchange data between threads, a central class, called `ControlUnit`, provides shared access and interfaces a database to record trajectories. It also takes care of communication with external hardware such as foot pedals, as well as with the PCs that handle signals of the strain gauges sensors and the servo motors. Most importantly, all kinds of execution programs, e.g. two-arm manipulation, three-arm manipulation, automatic endoscope control, knot-tying and machine learning are realized within this class. See Fig. 1 for an overview of the involved components.

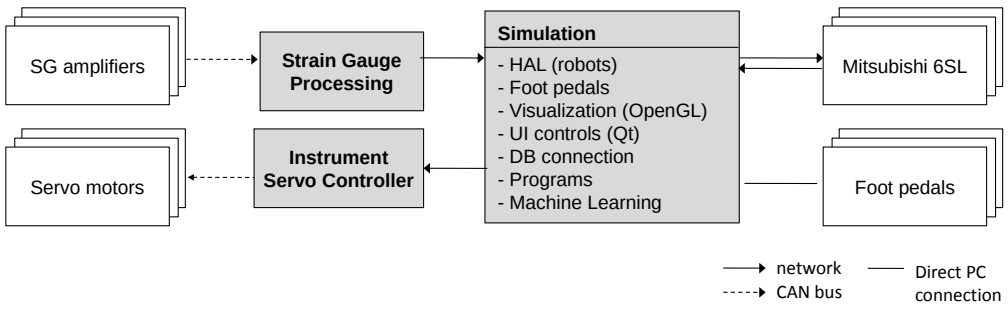
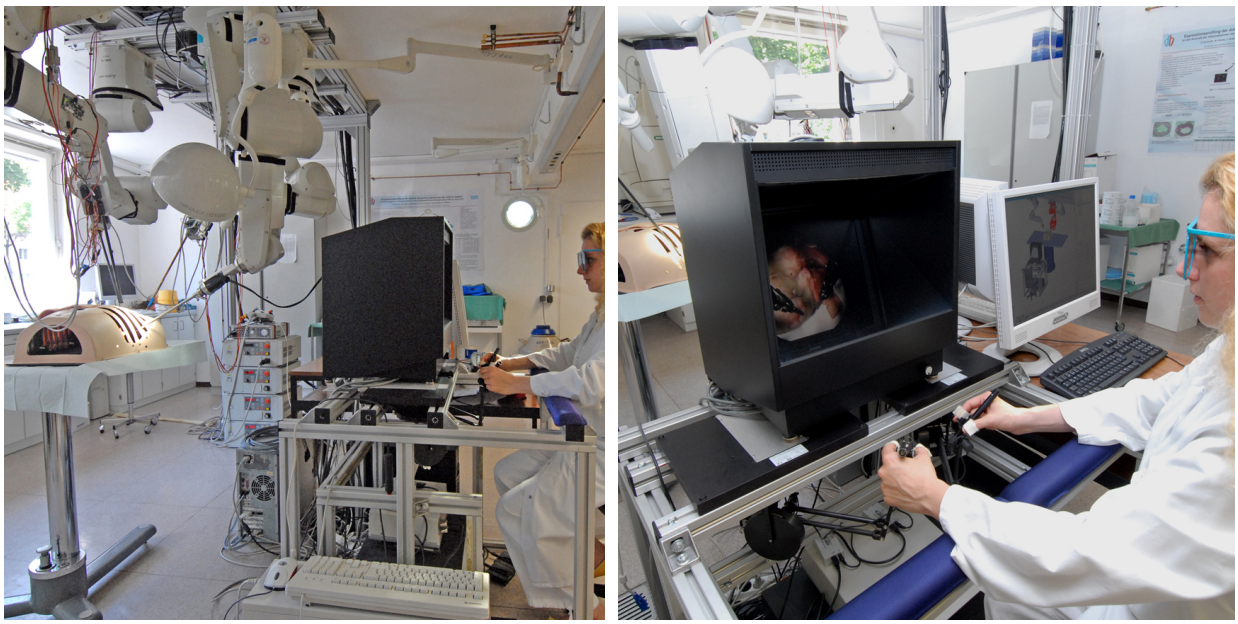


Figure 1: Static communication paths and connections between PCs and hardware in the old EndoPar software architecture

Although interchangeability of several components (e.g., replacement of the robotic hardware) by other instances is guaranteed with this framework, distribution of tasks over the network and data exchange is cumbersome and time consuming to implement. This is mainly due to the central role played by the `ControlUnit`. It manages and processes the entire information flow across the individual threads, according to the currently selected execution program. Consequently, the class is getting more crowded as more hardware is added and more functionality is integrated into the software. Small software components, each

responsible for a certain task, would be much easier to maintain. Outsourcing of functionality to dedicated programs or hardware might also be mandatory in some cases. E.g., our instrument tracking is, due to a lack of available Linux drivers for certain hardware, not part of the main software itself; yet it needs to consume joint readings of the simulation in order to provide tracking results. So far, this means that all necessary connections have to be established via the `ControlUnit` at program startup. If a connection fails, the execution of the main simulation will stop. These drawbacks have led to the development of a new, distributed version of the system's software that allows easy integration of new hard- and software components by decomposing the functionality into separate programs, which share data among each other by means of a central "blackboard". The architecture heavily utilizes the *cisst* libraries' multiprocess networking capabilities and is introduced in section 3.

2 Hardware



(a) Slave System (b) Master Console

Figure 2: The EndoPar system at the German Heart Center

The Endoscopic Partial-Autonomous Robot (EndoPar) system is an experimental robotic surgical system, developed by the Robotics and Embedded Systems research group at the Technical University of Munich. Being a telepresent type of system, it consists of a master console (Fig. 2(b)) and a robotic slave part (Fig. 2(a)). The master console, often referred to as the medical workstation, offers doctors the ability to operate the system via two PHANToM® Premium 1.5 devices from Sensable Inc. A customized handle snap-on, which is similar to handling a pair of tweezers, replaces the original switch in order to replace the digital behavior of the micro-grippers at the distal end of the surgical tools with a continuous input option. The forefinger has to be fixed at a rocker, which is connected to a small DC motor with an integrated position sensor. The force-feedback capabilities of the employed haptic devices are used to display force information that is derived at the instrument tip. Feedback in all translational part is possible, but no torques can be fed back. Forces acting on the instruments are directly measured at the instrument tip by means of sensitive strain gauges sensors. To achieve a high level of immersion of the operator, the master console provides a

high quality stereoscopic view of the scene. Image data is acquired with a stereo endoscope and displayed on a 3D display. The display system utilizes a semitransparent mirror to separate the views of two screens, each displaying the corresponding camera view for one eye. System interaction and switching between the different robot arms are made possible by several foot pedals. The slave part of our system is composed of four ceiling mounted industrial robots with 6 degrees of freedom. Either surgical tools or the camera can be attached via a magnetic clutch with the manipulators. The coupling system includes a hot-plug show that identifies the instrument and bridges all necessary electrical connections. The surgical tools are the EndoWrist™ instruments, originally deployed with the da Vinci® system. All instruments are operated by means of trocar kinematics, which ensures the observance of the fulcrum. For further information about the system we refer the reader to [4].

3 Architecture

The new, distributed architecture of the software comes with the advantage of a highly flexible basic structure that quickly adapts to new scenarios. It is illustrated in Fig. 3.

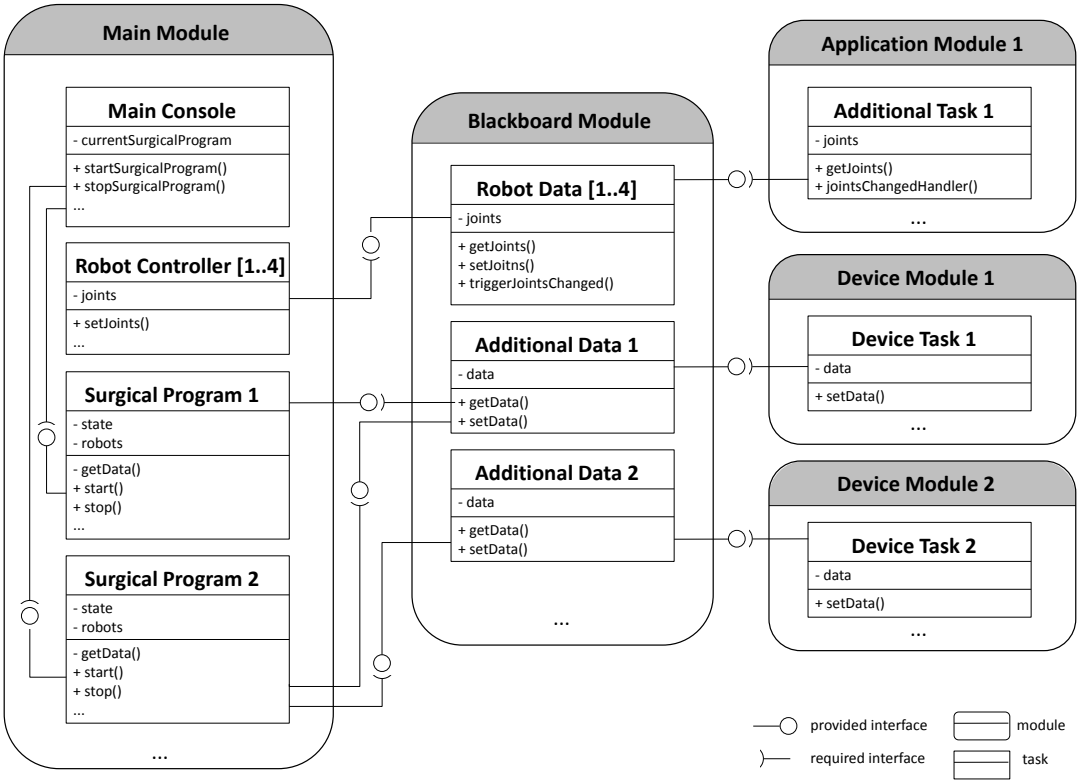


Figure 3: Components and interfaces of the distributed architecture

This framework organizes the robotic software system into *modules*. A module is a software component of the system that can be deployed and run as a separate process on any machine, while sharing data as well as functionality with other modules over the network.

Modules are realized using the *cisstMultiTask* library, which is one of the most important and most utilized components of the open source, cross-platform *cisst* libraries [2]. Specifically, each module is composed

of *cisst tasks* (`mtsTask` objects). Each task creates its own thread[1], together with a series of *provided interfaces* and *required interfaces*. It exports functions to its provided interfaces to make them available to other tasks, and adds to its required interfaces functions (in the form of *cisst* function objects) that it needs from other tasks. In addition to functions, one can also make use of events by having one task add events to its provided interfaces and invoke them as needed, and having another task implement event handlers and add them to its required interfaces. During its initialization, a module establishes run-time bindings between provided interfaces and required interfaces: a provided interface within a task is always connected to a required interface that resides in another task (could be from another module), and vice versa. Connection is based on string matching between process (i.e. module) names, task names, function names, and event names specified by the programmer. Afterwards, every task will be able to make thread-safe calls to functions (including event handlers) provided by its counterparts, even from a different module. *cisstMultiTask* implements intra- and inter-process calls in very distinct ways, but offers almost identical APIs. For inter-process calls, it uses the ZeroC™ Ice™ library [1], thereby hiding the complexity of network communication from tasks. In other words, the programmer would largely write the same code when using *cisstMultiTask* with and without Ice™, the only difference being that, when setting up the bindings, process names must be specified in the former case. Our framework further classifies the modules into three categories: the blackboard module, the main module, and optional modules.

The blackboard module is at the center of the system architecture. It merely serves as a common place for data storage and exchange, where all the other modules can read from and write to, thus resembling a blackboard. Each of its tasks keeps some data in its internal storage, and offers getter and setter functions for those data via its provided interfaces, which can later be connected to required interfaces of another module, thereby enabling that module to access the data. For example, it is usually desirable to create a task for each robot in order to share its joint angles. Each of the robot controller tasks in the main module can then call the setter function to keep the joint values up-to-date, whereas the getter function can be used by a visualization module to update the display of robot postures, or by any other application interested in the joint values. Alternatively, a "joint values changed" event can be added to the blackboard module, which gets invoked every time the setter function is called. That way, with a corresponding event handler in their required interfaces, the visualization module, etc. will be able to receive notification of any change in joint values and take appropriate measures. All the functions provided by the blackboard module should be simple getter and setter functions, written in only a couple of lines of code, with little or no extra computation. This is to ensure fast processing of remote calls at the blackboard, so as to keep the hub of the system responsive enough.

The main module implements key functionality of the system, including direct control of robots (via joint angles or Cartesian position) and advanced application of the robotic system in various scenarios. The latter is realized by means of "surgical programs". Each surgical program requires a particular hardware setup (e.g. which robots are needed, what tool is attached to each robot, what peripheral devices are used, etc.), and adopts a particular way of manipulating the robots (e.g. manual control via PHANToM® devices, gaze control, etc.). The main module should at least contain three kinds of tasks:

- Robot controller tasks should provide functions for basic control of robots. They talk directly to the hardware controllers to perform actual manipulation. They also publish current values of joint angles to the blackboard. Advanced control methods that are based on trocar kinematics (e.g. via PHANToM® devices) can also be implemented here, preferably in derived classes for appropriate hardware types. In short, these classes are analogous to the robot "model" classes in the model-view-controller architecture of the previous software system, with the addition of provided and required interfaces.

[1] When there is no need to create a new thread, `mtsComponent` is used instead of `mtsTask`.

- Surgical program tasks aim to solve of a major problem of the monolithic `ControlUnit` class in the old architecture, where code for different surgical programs interlaces, by creating a seprate class for each program. All these classes derive from an abstract `SurgicalProgram` class. The abstract class specifies common functions like `start()` and `stop()` (for starting and stopping execution of the surgical program). Every subclass needs to implement a state machine that dictates the workflow of the surgical program. The transition between the states is typically invoked by an event, e.g. the robots reaching their target positions, or a foot pedal being pressed. While at different states, the task may call different functions, possibly some combinations thereof, that are offered by the robot controller tasks, to move the robots.

- The main console task should present a user interface for this module, listing the available surgical programs and allowing the user to run any one of them. Of course, it should ensure that only one program is active at a time.

Optional modules include device modules and application modules. A device module enables the integration of a new hardware device (e.g. a foot pedal) into the system. Typically it wraps up the device driver and writes some device-specific data (e.g. whether the foot pedal is pressed) to the blackboard. This is usually done in conjunction with the addition of a new task to the blackboard module, where one or more provided interfaces are created to include getter/setter functions and value changed events. Application modules are software components that supplement the functionality present in the main module. An example is the visualization module, which reads joint values from the blackboard and renders robot movements.

The distributed architecture is advantageous in terms of scalability as well as flexibility. To extend the system with a new hardware or software component, one can always implement it as a new module, which compiles and runs on its own, while being able to share functionality and data with every existing module in the system. More often than not, the new module comes with some shared data as well. Consequently, the blackboard module will need to be augmented with a new task for storing and accessing the data, which is fairly trivial to write. As a further matter, adding a new surgical program to the system is also easier than before, since one can write most of the code in a new class. In either case, new remote calls will likely be involved, and to make them work, one should remember to insert code into the initialization procedure of existing modules for setting up necessary interface connections, which is usually just a few lines of code per module. On the other hand, although the system may consist of many modules, only the blackboard and the main module are required. If an optional module is not needed for the current scenario, one simply does not start it (i.e. not run the module's executable); no changes to the code are necessary.

Our implementation of the new architecture exploits our own robotics libraries [5] in combination with *cisst*. The system currently employs the following modules, distributed over 4 PCs:

- eye tracking
- foot pedals
- main console = simulation
- visualization
- 3D display + augmentation
- instrument tracking
- servo controller for instruments

- strain gauge processing
- position sensor for 7th DoF of PHANToM® (gripper state)
- space mouse camera control
- collision detection

4 Application Example: Gaze Contingent Endoscope Control

To illustrate the functionality of the distributed framework on a real-life example we have chosen an eye-tracking based endoscope control. The scenario is simple enough to be discussed in this context, but includes all major components of the software and exemplifies the multiprocess networking capabilites, which distribute tasks over three PCs running under three different operating systems. For an overview of the scenario and the involved components, see Fig. 4.

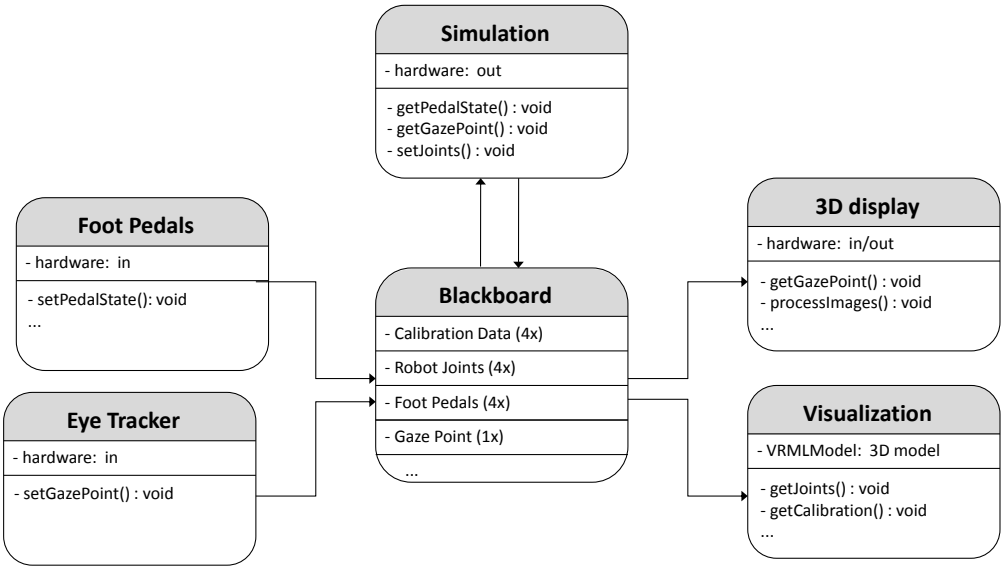


Figure 4: Illustration of the software framework using the eye tracker example

Gaze contingent endoscope control is one of the various options offered as part of the main module. The center of the endoscopic camera image gets automatically aligned with the surgeon's fixation point on the 3D screen, as long as a foot pedal is pressed. Consequently, two hardware components act as input (writer modules):

- The foot pedal module reads the current state of the four pedals (pressed/released).
- The eye tracker module processes the gaze position, obtained by the eye tracker glasses.

The hardware will be interfaced and read out in accordance with the device-specific timings. The obtained values are then published to the central storage instance, the blackboard. The eye tracker [3] is connected via FireWire to a Mac; the foot pedals are connected to the parallel port of the main PC, which is running a standard Linux. All necessary pre-processing steps, e.g., a recursive time-series filtering [6] to smooth

the approximately 400 values/sec obtained by the eye tracker, are performed outside and thus relieves the main module. Besides the already mentioned data, the blackboard holds also calibration data of the surgical instruments, spatial calibration data of the robot bases, and the joint angles of each robot.

The scenario involves two modules that act as readers:

- The 3D display module acquires two video streams from the endoscopic camera. After de-interlacing, correction of brightness and size, the images are displayed at 25fps on the stereo screen. It's running on a Windows machine and also reads the current (smoothed) gaze point to visualize its position on the screen. The visual feedback to the operator improves operability.
- The visualization module shows a 3D environment of the scene, including robot and instrument movements, the operating table, and the master console. To update the joint angles of the models, this module reads the calibration data and the joints from the blackboard at 25Hz.

The main simulation module is usually the only instance that acts as reader and writer simultaneously. In this example, the simulation gets either notified about value changes (in case of the foot pedals), or it periodically reads the eye tracker coordinates as long as the pedal is pressed. The two dimensional gaze vector indicates the direction and distance of the gaze point with respect to the screen center. The simulation then uses a velocity-based control law to generate the movement of the endoscope, parallel to the image plane, with distance-dependent velocity. The endoscope is tracked automatically as long as the length of the vector is greater than a certain threshold. The main module directly talks via UDP to the robot's hardware controller. The robots have a clock cycle of 6.5ms, which means that in every interval at least one set of joint positions needs to arrive at the hardware controller. This timing could not be met by a separate module that reads values from the blackboard and sends them to the robotic hardware, as the calculation of the trocar kinematics already takes about half of the cycle time. Nevertheless, joint values are written to the blackboard for further consumption, e.g., by the visualization module.

References

[1] http://www.zeroc.com. 3

[2] A. Deguet, R. Kumar, R. Taylor, and P. Kazanzides. The cisst libraries for computer assisted intervention systems. *The MIDAS Journal - Systems and Architectures for Computer Assisted Interventions (MICCAI 2008 Workshop)*, 2008. 3

[3] S. Kohlbecher, K. Bartl, S. Bardins, and E. Schneider. Low-latency combined eye and head tracking system for teleoperating a robotic head in real-time. In *Proceedings of the 2010 Symposium on Eye-Tracking Research & Applications*, pages 117–120. ACM, 2010. 4

[4] H. Mayer, D. Burschka, A. Knoll, E.U. Braun, R. Lange, and R. Bauernschmitt. Human-machine skill transfer extended by a scaffolding framework. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2866 –2871, may 2008. 2

[5] Markus Rickert, Michael Geisinger, Simon Barner, and Alois Knoll. Software development workflow in robotics. In *Proceedings of the Workshop on Open Source Software in Robotics, IEEE International Conference on Robotics and Automation*, Kobe, Japan, May 2009. 3

[6] Phillip D. Stroud. A recursive exponential filter for time-sensitive data. Technical Report LAUR-99-5573, Los Alamos National Laboratory, Oct. 1999. 4