HIROSCO - A High-Level Robotic Spacecraft Controller

Martin Stelzer, Bernhard Brunner, Klaus Landzettel, Bernhard-Michael Steinmetz, Jörg Vogel, Gerd Hirzinger *

* DLR German Aerospace Center, Institute of Robotics and Mechatronics, Germany
e-mail: martin.stelzer@dlr.de

Abstract

This paper presents a high-level control architecture for robotic spacecrafts. The design of this architecture focuses on future On-Orbit Servicing missions. Part of it is a component framework that improves software reuse in space applications and enables real-time communication between different components of a satellite which is essential for on-orbit servicing. Further, this architecture supports online reconfiguration of the components, resource management and a distribution of the components across a network. A supervisor monitors and coordinates all attached components. A prototype was successfully tested with a two axis robot and a force-reflecting joystick in a telepresence scenario.

1 Introduction

Nowadays people are used to services such as GPS and live TV coverage and rely on their availability. These services depend on a satellite infrastructure in earth orbit. But vital components of a satellite may fail like every other technical system. Usually, this satellite is lost and has to be shut down, even if most of its components are still operational, because it cannot be reached by humans for repair. The service itself must be moved to another part of the infrastructure or becomes unavailable. The satellite remains as space debris and endangers neighbor satellites, if it cannot hold the pose anymore. The increasing amount of space debris may ultimately lead to collisional cascading [7] that renders the use of satellites in certain orbits impossible for generations [8].

The emerging field of On-Orbit Servicing (OOS) is able to solve the problems stated above. An OOS satellite, called servicer, has a manipulator as payload that can grasp, repair or dispose of a defect satellite, called target. An example of a servicer is depicted in figure 1. Using modern teleoperation and telepresence techniques an operator on ground is able to remotely control the manipulator [13].

This is challenging because for the first time the movement of a satellite payload has a serious impact on the overall satellite pose [9]. Basically, there are two approaches to compensate this impact in order to successfully grasp the target. In the first approach the Attitude & Orbit Control System (AOCS) tries to hold the pose of the satellite using thrusters or reaction wheels while the manipulator is moving towards the target. Two problems arise here. First, the dimensions of the reaction wheels would need to be huge in order to compensate the dynamic movement of the manipulator, which is not feasible. Second, thrusters are not accurate enough to position the end effector of the manipulator at the right place, because the lever arm is several meters long. The second, more promising approach is to completely switch off the active parts of the AOCS and let the manipulator control system (MCS) take over the attitude control by adjusting the movements of the manipulator. Therefore, the MCS must compute the dynamic model of the overall satellite system. This model needs sensor information about the pose of the satellite from the AOCS in real-time. In order to realize this approach, a real-time link between the AOCS and MCS is mandatory. Further real-time links to the MCS, e.g. from a communication control system for telepresence control or from a camera control system for visual servoing, are supposable.

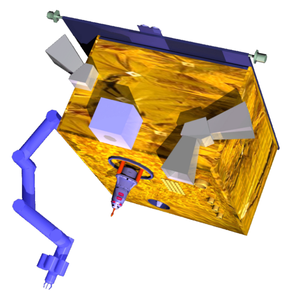


Figure 1: Example of a servicer

However, to the knowledge of the authors there is no satellite control architecture available that is able to support real-time links in a generic way. Furthermore, common satellite controllers follow a monolithic architecture so the code must be adapted to every new mission and code reuse is hindered.

This paper presents the current stage of HIROSCO, a High-Level Robotic Spacecraft Controller, developed by the DLR Center of Robotics and Mechatronics. This paper is organized as follows: Section 2 summarizes different concepts that are related to this work. The design of HIROSCO is explained in section 3 and its architecture is described in section 4. Thereupon, section 5 presents details about the demonstrator. The paper concludes with section 6 and future work is outlined in section 7.

2 Related work

Within the last decade there has been much effort in the field of software frameworks for robotic applications, so-called robot frameworks. Their common goal is to ease the development of complex robotic applications by segmenting the application into smaller parts or layers to break down the complexity.

The Open Robot Control Software (OROCOS) [4] is a general purpose and open source framework to control distributed robotic systems. It uses many aspects of component-based software engineering (CBSE). CBSE basically proposes a segmentation of an application into reusable components. For details about CBSE and its benefits for (robotic) applications see [3] and [14]. Furthermore, OROCOS offers an existing set of components such as a real-time toolkit and a kinematics and dynamics library that can be used to develop a robotic application. These components can be dynamically added to or removed from an application. A component can be extended or created from scratch by a developer to enhance functionality provided by existing components. In 2004, the ORCA [2] project emerged from OROCOS. The goals are similar, but they explicitly focus on CBSE and use a different middleware solution. Beyond the segmentation into components, OROCOS and ORCA propose no further architectural constrains. A design choice of ORCA is not to support real-time links between its components.

Player [5] is an object-oriented approach and the de facto standard in the domain of robot frameworks. The Player architecture decomposes an application into clients and servers. The difference between them is that a client only consumes services while a server additionally provides them. A service in the Player context provides access to any kind of hardware device of a robot or to any kind of high-level algorithm such as image processing. These services can be distributed across a network. There is an existing set of services provided by Player and the developer can create custom services. Player provides a TCP implementation for the communication between clients and servers. This implementation can be extended by the application developer to use other protocols or a middleware solution. However, Player does not support real-time links, either.

The three tier architecture (3T) [1] helps to deal with dynamic environments by segmenting an application into three hierarchical tiers: A deliberation tier for high-level planning to create a set of tasks to reach a goal, a sequencing tier that decomposes this tasks into different skills that must be performed to fulfil the task and a reactive tier that finally executes a skill in real-time. The Coupled Layer Architecture for Robotic Autonomy (CLARAty) [12] uses a similar segmentation approach to increase the autonomy of robotic systems. Based on the ideas of 3T, it couples the deliberation and the sequencing tier within a decision layer. This layer has access to a functional layer, which represents the services a robot can provide. The definition of a service corresponds to that one used in the player context. Method calls on abstract interfaces are used for the interaction between most of these services. Changing a method in such an abstract interface triggers a recompilation of all services that use this interface, whether they explicitly use the altered method or not. Furthermore, most of these services share the same memory region which breaks their encapsulation.

3 Description of the design

The design of HIROSCO follows explicitly the requirements of a control architecture for robotic spacecrafts with a focus on OOS. Nevertheless, the design is applicable to non-OOS spacecrafts or even to non real-time applications that simply need coordination and communication across actually distinctive components. A detailed description of the design goals is given in this section.

All satellites are assembled of a satellite platform, standard components (e.g. AOCS, Thermal Control System or Power Control and Distribution Unit) and mission specific payload components (e.g. a manipulator and stereo cameras for OOS). The design of HIROSCO adopts this layout so that a satellite application can be assembled of reusable software components, called subsystems, using a component framework to interact with each other. Such a subsystem consists of an executable file that implements the subsystem's features, documentation about how to use them and an interface description to formalize them independently from a programming language. Thus, the interface description contains vital information about the underlying hardware, available configurations and parameters and offered services. It encapsulates the subsystem because using those services is the only way of interacting with a subsystem. To be able to support the hardware configuration of many satellites, the design should contain an abstraction layer so that subsystems can be implemented independently from the target platform. So far, this layout is quite similar to the OROCOS/ORCA architecture.

However, during its lifetime a satellite runs in various modes, e.g. telepresence or autonomous mode. Each of these modes requires a different set of subsystems to be operational. Some of these subsystems need to be connected to each other by real-time links. Therefore, the design should support the dynamic configuration and interconnection of subsystems. Particularly the autonomous mode requires a supervisor that is amongst others responsible for logging telecommands and telemetry data, monitoring all existing subsystems, global error handling and managing inter-subsystem communication. Likewise, the telepresence mode requires such an entity, for example in case of a ground link failure while the operator grasps the target. Whether the supervisor should be part of a three tier, a coupled layer or a different approach is yet to be determined.

To simplify the commissioning of subsystems for the different modes of a satellite and their coordination, a finite state machine is mandatory for all subsystems.

Unfortunately, space qualified processors suffer from low performance. In order to run sophisticated algorithms on board, several processors are recommendable. Hence, the design should be able to support subsystems that are distributed across a network. Of course, the performance and quality of the real-time links depend on the characteristics of the underlying network.

4 Description of the architecture

The three basic elements of the architecture are depicted in figure 2 using a UML component diagram. A *subsystem* usually represent a set of hardware devices, the corresponding algorithms and data structures. Each subsystem is connected to a *supervisor* that monitors, controls and coordinates the application. For that reason it exchanges telemetry and telecommand data with the subsystems. Furthermore, subsystems can be connected to each other to transfer real-time data. All connections are established by a *component framework* using well-defined interface descriptions. At the moment C/C++ is the only supported programming language. The following subsections will describe those elements in detail.

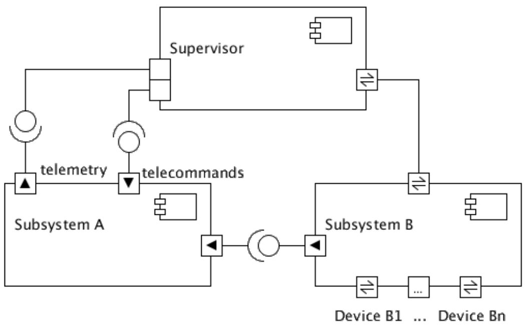


Figure 2: Overview of the architecture

4.1 Component framework

The purpose of the component framework is to provide services most subsystems will require. These services are partitioned into eight libraries.

Every computer can offer a certain amount of resources. But space qualified hardware is expensive so that those resources are very limited on a spacecraft. Exceeding these limits will result in undesirable behavior of the system. Therefore, each subsystem must acquire its resources via the *resource management library*. At the present stage, this library only manages memory. The management of further resources such as file descriptors is subject to future work. To avoid external memory fragmentation on real-time operating systems, this library preallocates large chunks of memory at startup. They are segmented into pools of blocks of different sizes and finally made available to the subsystem. It is not allowed to increase the amount of available memory at runtime. A warning could be generated if a considerable amount of memory (e.g. 75%) is in use. A reference counting smart pointer is in charge of garbage collection. The interface of the resource management is compatible to the allocator interface of the C++ standard library.

The *utilities library* contains an abstraction layer for platform dependent services such as threads and semaphores. Hence, subsystems can be implemented independently from the target operating system. Further general purpose classes, e.g. custom exceptions, are located in this library.

The component framework offers services and corresponding protocols specified by the Telemetry and Telecommand Packet Utilization Standard (PUS) [6]. The interaction of subsystems is based on these services. By now, there is no full compatibility to the standard, but the aim is to provide a generic and compatible implementation of all relevant PUS services. A service is called generic in this case if different subsystems can use it without the need of modification. Currently, services for function call management, event reporting, subsystem testing and housekeeping data reporting are available. Each subsystem may create up to one instance of each service. The configuration of any service a subsystem provides is an essential part of its interface description. Furthermore, real-time links are implemented as custom PUS service. The respective instances of this service perform a handshake to establish a link between two subsystems. During this handshake the protocol layout of the real-time data, byte ordering, sampling time and transfer method (e.g. shared memory or ethernet) are negotiated. Subsystem developers are also able to create their own custom service using this *PUS library*.

In order to be useful for a subsystem a generic service will often require access to the subsystem's data structures and their layout. Such a data structure, that is shared between a subsystem and the component framework, is called parameter. The layout of a parameter, its name and id are part of the interface description. Using the name, a defined parameter can be referenced within the interface description. The combination of layout and id is called parameter id. It is used to reference a parameter within commands of an operator and within interface descriptions of other subsystems. Therefore, both parameter id and name must be unique across an interface description. A subsystem must publish all the parameters it wants to share with the framework and the services must be able to utilize them. For this purpose the *parameter library* provides the required functionality. All available parameters are published in a so-called dictionary that can be accessed by all services and the subsystem. To avoid data corruption, the dictionary provides a locking mechanism if the subsystem or a service needs to access a parameter.

Each subsystem can transmit and receive data to and from a hardware device, using an abstract data flow provided by the *data transfer library*. A data flow represents any means of communication and is configured by an endnode descriptor. This descriptor contains information about the address and the transfer type to be selected

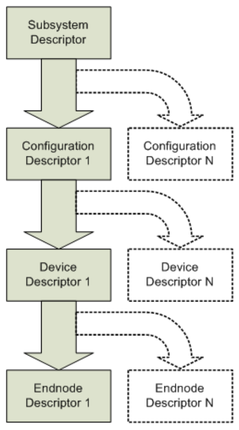


Figure 3: Layout of the subsystem descriptor

for the transfer (e.g shared memory, message queue, IO device, etc.) and is also part of the interface description. Therefore, a change of the location of a hardware device does not require recompilation of a subsystem. Furthermore, issues like de-/serialization and endianess conversions of packets are handled by each data flow and, thus, not relevant to a subsystem. For the communication between all components the transparent inter-process communication (TIPC) [11] is used. Due to this middleware solution, distributed services can be accessed regardless of their physical location within a network.

To read the content of XML files a lightweight *XML parser* is available. This sequential parser ensures that the syntax of a file is written well-formed. However, the semantic is not validated by the parser.

Mandatory to all subsystems is an interface description, called subsystem descriptor, that is written in XML. Its layout (ref. figure 3) is based on the layout of the USB descriptor. The *descriptor library* implements the subsystem descriptor and uses the XML parser to generate an in memory representation of the XML file. This descriptor contains the name and the id of a subsystem that are unique across all subsystems. They are used later on by the supervisor and ground station to identify each subsystem. The definition and initialization of available parameters is also provided by the subsystem descriptor. For example, listing 1 defines four different parameters, while two of them are part of a structured parameter.

Listing 1: Example of parameter definitions

```
<StructRef name="CUR_VALUES" id="0" type="STRUCT">
  <ParamRef name="STATE" id="0" type="INT32"/>
  <ParamRef name="POSITION" id="0" type="DOUBLE"/>
</StructRef>

<ParamRef name="DES_POSITION" id="1" type="DOUBLE"/>
```

Furthermore, the subsystem descriptor contains a list of configurations that can be applied to the subsystem. Subject of these configurations are priority and sampling time of the subsystem and the configuration of the available PUS services. Listing 2 gives an example of a PUS service configuration for the housekeeping data reporting service. It uses the parameters defined in listing 1. This service configuration causes the corresponding service to generate a telemetry packet each 500 cycles and to tag this packet with the id 4711. The content of the packet will be a structured parameter that has been sampled once and a single parameter that has been sampled five times within the given interval.

Listing 2: Example of a PUS service configuration

```
<HKService id="4711" interval="500">
    <SampledOnce>
        <ParamRef name="CUR_VALUES"/>
    </SampledOnce>
    <SampledNREPTimes nrep="5">
        <ParamRef name="DES_POSITION"/>
    </SampledNREPTimes>
</HKService>
```

Moreover, each configuration contains descriptions of available hardware devices. These descriptions include the device settings for this configuration and a list of endnode descriptors that are required by data flows to connect to a specific device. To a certain degree the subsystem descriptor can be changed online and without the need of recompilation of a subsystem because it is initially evaluated while initializing the subsystem at runtime.

The *generic subsystem* represents the active part of the component framework and, thus, distinguishes the framework from a collection of libraries. It simplifies the development of an application specific subsystem because it implements standard behaviors such as state transitions that are equal across subsystems and maintains a list of services that have been started by the generic or the specific subsystem. Furthermore, the generic subsystem implements the main loop of the subsystem. At the beginning of this loop the function management service is started so that commands can be received. Additionally, connections to the services offered by the supervisor are established. After that, the cyclic code is started which processes the current state machine node. Furthermore, all services contained in the list are executed. The sampling time and priority of this loop is adjusted during the commissioning of the subsystem according to the requested configuration. A supposable extension is to move services into their own loop so that different sampling times and priorities can be applied to them.

4.2 Subsystem

Subsystems are the basic components of which an application is assembled. They are connected to their environment via the component framework and can interact with each other using (custom) PUS services. More precisely, these services are the only means of interaction. Therefore, they exactly define the interface of a subsystem.

All a subsystem developer must do in order to create a specific subsystem is to inherit the generic subsystem and implement the finite state machine as depicted in figure 4. This state machine consists of ten separate states. They were designed to ease the commissioning and coordination of subsystems.

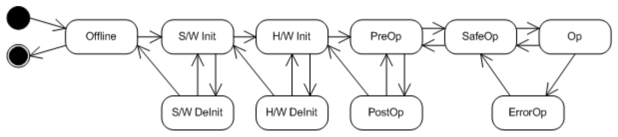


Figure 4: Finite state machine for subsystems

Each subsystem starts or stops in the state "Offline". All services provided by the specific subsystem are started at this time and added to the list maintained by the generic subsystem. Furthermore, the subsystem descriptor is loaded into memory. Apart from those, no further resources are claimed by the subsystem and all hardware devices are switched off. If a subsystem is not required to be active for a longer period of time or if resources are required somewhere else, this state should be selected.

During the subsequent state "Software-Init", all data structures that should be used by services of the component framework must be published as parameters in the dictionary. After this procedure is completed, the generic subsystem will initialize those parameters according to the subsystem descriptor. The next step to commission a subsystem is the "Hardware-Init" state. All hardware devices that belong to this subsystem are activated and initialized in this state.

The configuration of the subsystem takes place in the "Pre-Operational" state. For that purpose, the operator must select a configuration listed in the subsystem descriptor that should be applied. The generic subsystem configures all services that have been added to the service list. The specific subsystem is responsible to establish a connection to its hardware devices and to configure them. Parameters can also be (re-)initialized depending on the selected configuration. This is helpful if controller parameters need to be changed for each configuration. Realtime links are established by the component framework on command of the operator during this state. They will start their operation right after establishing them. Housekeeping data is transmitted to the ground station from now on, too.

After the configuration is completed, the state of a subsystem can be changed to the "Safe-Operational" state. In this state all control algorithms implemented by the subsystem developer are active, but the actuators of the hardware are still disabled. This state can be used to verify the proper function of a subsystem. After the verification is completed, the state machine can switch to the "Operational" state. The actuators are active and the subsystem can now be controlled completely. The "Error-Operational" state secures that the hardware devices can reach a defined state after a severe error has occurred either in the subsystem itself or in a different one. The detection of errors in the sphere of a subsystem is primary the task of the subsystem itself. For example, if the temperature of a hardware device reaches a critical limit the subsystem has to notify the supervisor so it can propagate this error and take further actions. This notification could be executed by a corresponding service, too.

There are two de-initializing states and a "Post-Operational" state which revoke all actions of the corresponding initializing and "Pre-Operational" states. A template of this state machine is available to subsystems developers and can be used as a guideline to implement the state machine of a subsystem. Each state can be subdivided into different phases if required.

By now, the composition of control algorithms and their execution during the operational states is beyond the scope of HIROSCO. But there are already approaches that deal with this topic [10] and which could be integrated into the component framework in future.

Each subsystem runs as a separate process for the reason of encapsulation. Thus, a subsystem cannot influence the data of others by accident.

4.3 Supervisor

Basically the supervisor is a standard subsystem that acts as a superior entity at a higher hierarchical level than the other subsystems. It is subdivided into a platform abstraction package and a supervision package (ref. figure 5).

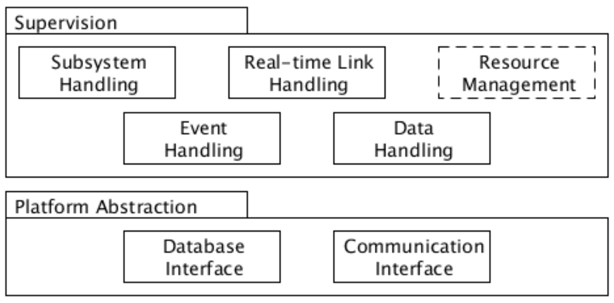


Figure 5: Packages of the supervisor

The platform abstraction package was introduced to encapsulate dependencies that might vary between different missions or satellite platforms. An example for such a dependency is the database. One mission might choose to log all packets to an ASCII file and another one might choose to run a relational database management system. Therefore, the "Database Interface" contains strategies to access different types of databases. The strategy, and thus the database that is currently used, can be changed during runtime. The "Communication Interface" carries out the transfer of packets to and from other subsystems.

The supervision package contains all modules that represent the actual tasks of the supervisor. As the supervisor will receive all telecommands and telemetry data during the lifetime of a satellite, it is in charge of coordinating all other subsystems. The "Real-time Link Handling" ensures that subsystems are not allowed to switch from "Pre-Operational" to "Safe-Operational" as long as a required real-time link is not present. Furthermore, it maintains a graph of all real-time links so it is aware of all existing real-time networks at any time. A real-time network is a net of subsystems that are (transitively) connected to each other. Thus, in case of a failure of a subsystem the supervisor is able to notify all members of a real-time network if necessary.

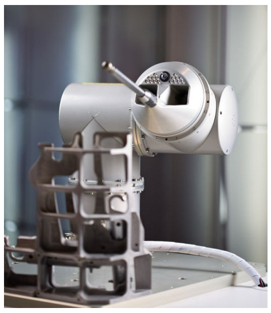
However, the supervisor provides a slightly different set of services than standard subsystems do. Among these, the routing service and the event reporting service are the most important ones at the moment. The routing service is part of the "Data Handling" and is required because by definition subsystems are not able to exchange non real-time data directly. This is due to the fact that the supervisor must give permission for each telecommand to be executed. This decision is based on the current state and mode of the satellite. Furthermore, each telecommand and telemetry data packet must be logged by the supervisor. But not only telecommands must be routed to other subsystems. Also telemetry data must be forwarded to the ground station and, therefore, routed to a communication subystem. The event reporting service deals with asynchronous progress and error notifications. The "Event Handling" may react to an error with predefined recovery plans based on severity and reason of the error. PUS defines three severity levels for error reporting: low, medium and high. For example, if the manipulator failed during a grasp action and had to be shut down to avoid further damage, the MCS would signal an error of high severity. The "Event Handling" must then shut down the real-time network that includes the MCS and hand over the control of the station keeping to the AOCS autonomously. A decisional layer on top of the supervisor could find a recovery plan that suits better than a predefined one or could create a plan which avoids specific errors. However, it does not exists in HIRSOCO by now, but is an aim of future work.

As mentioned before, the resources of a computer system in space are quite limited. Because subsystems run in separate processes with defined memory boundaries, each subsystem can manage this resource locally using the resource management library. The supervisor only has to keep track of how many processes have been started and how much memory they are allowed to use at a maximum. But there are resources that are not local to subsystem scope, e.g. shared memories, ports, files and electrical power. This global "Resource Management" is the task of the supervisor. For example, it must refuse the use of a manipulator if the solar panels do not produce enough power at that time. However, the implementation of this feature is subject to future work.

All subsystems are started by the "Subsystem Handling". Therefore, it can be configured with a list of executable files to load at startup. If a command signals the supervisor to shut down, it will broadcast this signal to all subsystems. They will change back to their "Offline" state before they stop.

5 Practical tests

In order to test, verify and demonstrate the prototype of HIROSCO, a telepresence scenario was chosen. The ROKVISS [13] experimental model serves as manipulator (ref. figure 6(a)). ROKVISS is DLR's most recent space robotics experiment. The flight model of ROKVISS is mounted to the outer surface of the ISS. It has two joints and can be moved along a contour or in free space. Additionally, it can pull a vertically and a horizontally mounted spring. During the practical tests the experimental model can be moved by commands of the operator or by using the DLR force-reflecting joystick (see figure 6(b)). This joystick has two degrees of freedom as well. Both devices are connected to the same real-time computer with a Celeron 2 GHz processor running VxWorks.


(a) ROKVISS experimental model

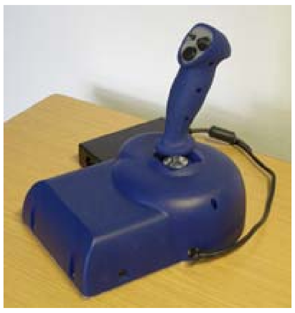

(b) DLR force-feedback joystick

Figure 6: Hardware of the demonstrator

Figure 7 shows the subsystems participating in this scenario. They run at a sampling rate of 2 kHz on the real-time computer. The manipulator subsystem and the joystick subsystem implement an interface to their corresponding hardware using a device driver and algorithms to control the device. Both subsystems can be driven in a stand alone configuration or in a configuration coupled to each other. For the first, a command of the operator triggers the robot to move to a desired position or the joystick to display a desired force. For the latter, the manipulator subsystem provides the current position and the measured force of each joint and requires the desired position for each joint as real-time data. The joystick subsystem offers its current position and requires the current position and measured force from the robot subsystem in real-time. By commands of the operator and after the verification of the supervisor, the two subsystems, or, more precisely, their control algorithms, are connected to each other by

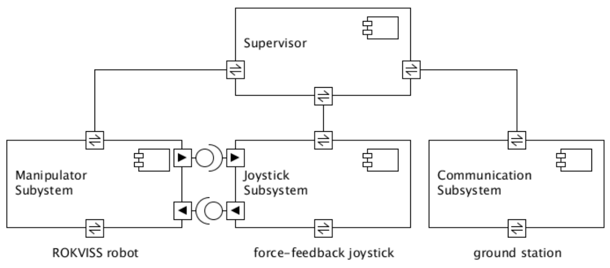


Figure 7: Architecture of the demonstrator

two real-time links, one for each direction, and brought into the "Operational" state. The manipulator subsystem then moves the robot according to the current position of the joystick. On the other hand, the joystick displays the forces encountered by each joint of the manipulator.

The communication subsystem maintains a TCP connection to a simple ground station written in Java that is able to display the housekeeping data of the subsystems and to generate all required telecommands. The position of the robot is displayed using a 3D viewer.

The strategy for error handling implemented in the supervisor is to shut down all subsystems participating in the real-time network in case of errors of high severity. This was tested by unplugging the joystick or the robot from the real-time computer. This results in an immediate shut down of joystick and manipulator subsystem. Errors of medium severity cause the supervisor to change the state of a subsystem to safe-operational. For example, to exceed the torque limit of a joint would cause such an error. Errors of low severity and progress information are simply logged to the console without further reaction.

6 Conclusions

A High-Level Robotic Spacecraft Controller was presented in this paper. It was designed to control satellites for future OOS missions, but is not limited to this domain. Its component framework serves as a platform to interconnect distinctive software components and provides a set of generic services to ease the development of reusable subsystems. The supervisor acts as a superior instance to control and monitor all present subsystems and their real-time links. Despite the fact that the development is still at an early stage, the demonstrator shows that the current prototype of HIROSCO is already able to provide a sufficient amount of the designed features to run a fairly complex telepresence scenario. However, there are still some challenges to master until HIROSCO can be used for controlling a servicer.

7 Future work

Future work will focus more on the supervisor since key abilities such as global resource management and command scheduling are essential for a space mission, but not yet available. Furthermore, some sort of decisional instance is required to handle the largely unknown environment and unforseen circumstances during an OOS mission. Whether to place this instance on top of the supervisor (three-tier approach), within the supervisor (coupled-layer approach) or into a subsystem is yet unclear and needs to be defined and implemented.

However, the component framework is not finished, either. For example, missing PUS services need to be implemented and the existing ones need to be made fully compliant to the standard. Despite the platform independent design, the component framework only supports the real-time operating system VxWorks. In future, Linux and QNX shall be supported, too. To achieve this, different middleware solutions than TIPC might be considered.

Another aim for future work is to create a repository of subsystems that can be used to assemble new applications in a short time. Future integrations in different robotic applications will create the basis for such a repository.

References

[1] R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack, “Experiences with an architecture for intelligent, reactive agents”, Journal of Experimental & Theoretical Artificial Intelligence, 9(2), (1997), pp. 237–256.

[2] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Orebäck, “Orca: A component model and repository”, Software engineering for experimental robotics, , pp. 231–251.

[3] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Orebäck, “Towards component-based robotics”, in 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2005.(IROS 2005), 2005, pp. 163–168.

[4] H. Bruyninckx, “Open robot control software: the OROCOS project”, in IEEE International Conference on Robotics and Automation, 2001. Proceedings 2001 ICRA, volume 3, 2001.

[5] T. Collett, B. MacDonald, and B. Gerkey, “Player 2.0: Toward a practical robot programming framework”, in Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005), Citeseer, 2005.

[6] European Cooperation for Space Standardization Secretariat, “Ground Systems and Operations - Telemetry and Telecommand Packet Utilization (Packet Utilization Standard)”, ECSS-E-70-41A.

[7] D. Kessler, “Collisional cascading: The limits of population growth in low earth orbit”, Advances in Space Research, 11(12), (1991), pp. 63–66.

[8] D. Kessler, “Critical Density of Spacecraft in Low Earth Orbit: Using Fragmentation Data to Evaluate the Stability of the Orbital Debris Environment”, in Proc. 3 rd European Conference on Space Debris, H. Sawaya-Lacoste (ed.), European Space Agency, Noordwijk, 2001.

[9] K. Landzettel, G. Schreiber, B. Brunner, B. Steinmetz, K. Deutrich, and G. Hirzinger, “DLR/ NASDA’s Joint Robotics Experiments on ETS VII”, in J. NASDA (Editor), ETS VII Symposium, Tokyo, Japan, 14.03.2000, 2000, pp. 136–145.

[10] A. Mallet, S. Fleury, and H. Bruyninckx, “A specification of generic robotics software components: future evolutions of genom in the orocos context”, in International Conference on Intelligent Robotics and Systems, 2002.

[11] J. Maloy, “TIPC: providing communication for

Linux clusters”, in Linux Symposium, volume 2, 2004, pp. 347–356.

[12] I. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I. Shu et al., “CLARAty: Challenges and steps toward reusable robotic software”, International Journal of Advanced Robotic Systems, 3(1), (2006), pp. 023–030.

[13] D. Reintsema, K. Landzettel, and G. Hirzinger, DLR’s Advanced Telerobotic Concepts and Experiments for On-Orbit Servicing, volume Volume 31/2007 of *Tracts in Advanced Robotics*, STAR - Springer, 2007, URL http://elib.dlr.de/52944/.

[14] C. Szyperski, Component software: beyond object-oriented programming, Addison-Wesley / ACM Press.