WP9 – Validation on platform

Deliverable D9rb.2 – Robotic Application Component Specification

WP9 – Validation on platform : Deliverable D9rb.2 – Robotic Application Component Specification

by F. RUSSOTTO (CEA)

Published: july 2003

Table of Contents

Document Presentation

Project Coordinator

Organization:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14, 46022 Valencia, Spain
Phone:	+34 963877576
Fax:	+34 963877576
Email:	alfons@disca.upv.es

Participant List

Role	Id.	Participant Name	Acronym	Country
CO	1	Universidad Politecnica de Valencia	UPVLC	E
CR	2	Scuola Superiore Santa Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	Commissariat a l'Energie Atomique	CEA	FR
CR	5	Unicontrols	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	Visual Tools S.A.	VT	E

Document version

Release	Date	Reason of change
1_0	28/07/2003	First release

1 Introduction

The objective of this work package is to demonstrate the efficiency of the developed operating-system components (WP4 to WP7). For this purpose three applications are proposed covering the following domains: robotics, multimedia and process control.

This document defines the Robotic Application Component Specification. Robotic applications require high performances from the real-time operating system: high frequency and delay accuracy for digital servo loops, high reactivity to take into account asynchronous events, robust and reliable input/output control, communication and efficient multitasking

This document is not intended to give a full description of the internal structures and mechanisms of the application. Its goal is to provide a detailed description of the interface mechanisms involved when the application interacts with the other components (eg: the Linux Kernel, the RTLinux Kernel and the system components, including the OCERA components).

2 General

2.1 Application overview

The Robotic application that will be developed for the OCERA project is a Servo-control application of a master robot arm (called “haptic device”). It is used in an immersive and interactive theater. The haptic device allows user to manipulate virtual objects located into a virtual 3D scene projected in front of the user through a high dimension screen.

Figure 1 : Interactive and immersive system of virtual manipulation

The haptic device is a six-axis robot arm including six motors, six position sensors and a six-axis force sensor. It is controlled in position and force to give the operator force feedback computed from the interaction with the virtual environment. So that the user can feel contacts, collisions, repulsions, frictions and even gravity when he manipulates objects into the virtual scene.

The system requires high performances from the embedded controller that pilots the haptic device. Indeed, for good force feedback, the haptic device must have a high bandwidth, typically 50 to 100 Hz. Such high bandwidth require a very high sampling frequency regarding the embedded controller; to fix ideas, a minimum of 10 to 20 times the bandwidth is required, e.g.: 1000 Hz sampling rate is needed from the embedded controller.

The Robotic application running on a true robot-arm will be finalized at the end of the project (application V1). For now, the robotic application V0 will be developed in order to run a simulated robot-arm instead of a true one. This will allow to debug and fine tune the application prior to use it on a real device which could be dangerous for user if that phase was bypassed.

2.2 Hardware architecture

The final target platform for the Robotic application V1 (the controller) is an Intel Pentium III @850 MHz on an Intel i440BX chipset as described in document D9rb.1 (Robotic application requirement specification).

However, for tests purpose, the application V0 will be run on a most powerful platform: an Intel Pentium 4M @2.0 GHz on an Intel i845 chipset (Dell Precision M50). So that, the robot-arm simulator introduced for test purpose should not disturb the application behavior, moreover we should be able to run locally in a graphical environment a Client application and an interface for the simulator. This will improve the application robustness.

2.3 Software architecture

2.3.1 Operating system, Linux & RTLinux kernels

The Operating System installed for Application V0 is Linux. It is based on a Debian 3.0R1 distribution.

The Linux kernel has been recompiled to version 2.4.18 (compiler: gcc 2.95.4) with the following patches preliminary applied (prepatched OCERA Linux kernel): 'BigPhysArea', 'LowLatency', 'PreemptKernel' and OCERA pcomp-1.0-1 (PK/RTL compatibility). The rtl-3.2-pre1 patch was also applied to Linux kernel. The following kernel options have been selected: no ACPI, no APM, BigPhysArea, Kernel-Preemption, Low-Latency, IDE DMA[1] and APIC UP; the full configuration file will be made available on CVS.

RTLinux kernel version 3.2-pre1 has been installed with BigPhysArea patch applied. The following kernel options have been selected : dynamic memory support, no user space real-time; the full configuration file will be made available on CVS.

[1] in case of using an old or low-perf. disk drive, this option should be disabled in order to avoid latency issues due to PCI DMA.

2.3.2 OCERA components

The following OCERA components will be used in application V0:

- Doubly Indexed Memory Allocator 0.70-1
- POSIX Barriers 0.1-1
- POSIX Timers 0.2-1
- POSIX Trace 1.0-1
- Fault Tolerance Application Monitor 0.1-1
- Fault Tolerance Controller 0.1-1

2.3.3 Robotic application V0

The overall Robotic application V0 consists mainly of two components : a Servo-control application running on the local machine (mainly in the Linux kernel) and a Client application allowing high level control of the Servo-control application and running remotely on a Linux or Windows machine or locally in the Linux user space.

The Client-application consists mainly of one component described shortly in the next chapter but not detailed since it does not involve any real-time feature. In the next chapters, we will use: “application” to deal with: “Servo-control application”.

The Servo-control application is intended to control in position, speed and force a six DoF robot-arm called Virtuose. As the application is in pre-alpha test version, it will not be connected to a true Virtuose robot-arm but to a simulated one running synchronously in the context of the main Servo-control task.

The Servo-control application is divided into 5 main components which are described in the following chapter:

- Communication protocol (**hap_protocol**)
- Synchronous cycle supervisor (**hap_clock**)
- Servo control (**hap_servo**)
- Input/Output (**hap_io**)
- Virtuose simulator (**hap_simulator**)

Each component dependency is also described as soon as it involves calls or requires services from Linux, RTLinux or OCERA components.

At runtime, the Servo-control application consists of three hard real-time kernel tasks and two soft real-time user tasks (actually 2 threads launched by a main process sleeping until threads termination); see figure below for details.

- Soft real-time tasks (user space) :
    - SERVER and CLIENT threads of the PROTO_INTERFACE process respectively run the server side and the client side of the **hap_protocol** component,
- Hard real-time tasks (kernel space) :
    - PROTO_IN runs the **hap_proto_in** sub-component of **hap_protocol**,
    - CLOCK runs the **hap_clock** component,
    - SERVO runs the **hap_servo** component and its dependencies including the **hap_proto_server** sub-component of **hap_protocol**, the **hap_io** component and the **hap_simulator** component.

The following figure shows the overall Servo-control application architecture (excluding Client application). A graphical user interface should be developed to control the behavior of the Virtuose robot arm simulator (**hap_simulator**); this component will run locally in user space and will communicate with the simulator through RT-FIFOs (component and associated FIFOs are not represented on the figure). Main components of the application are detailed in the next chapter.

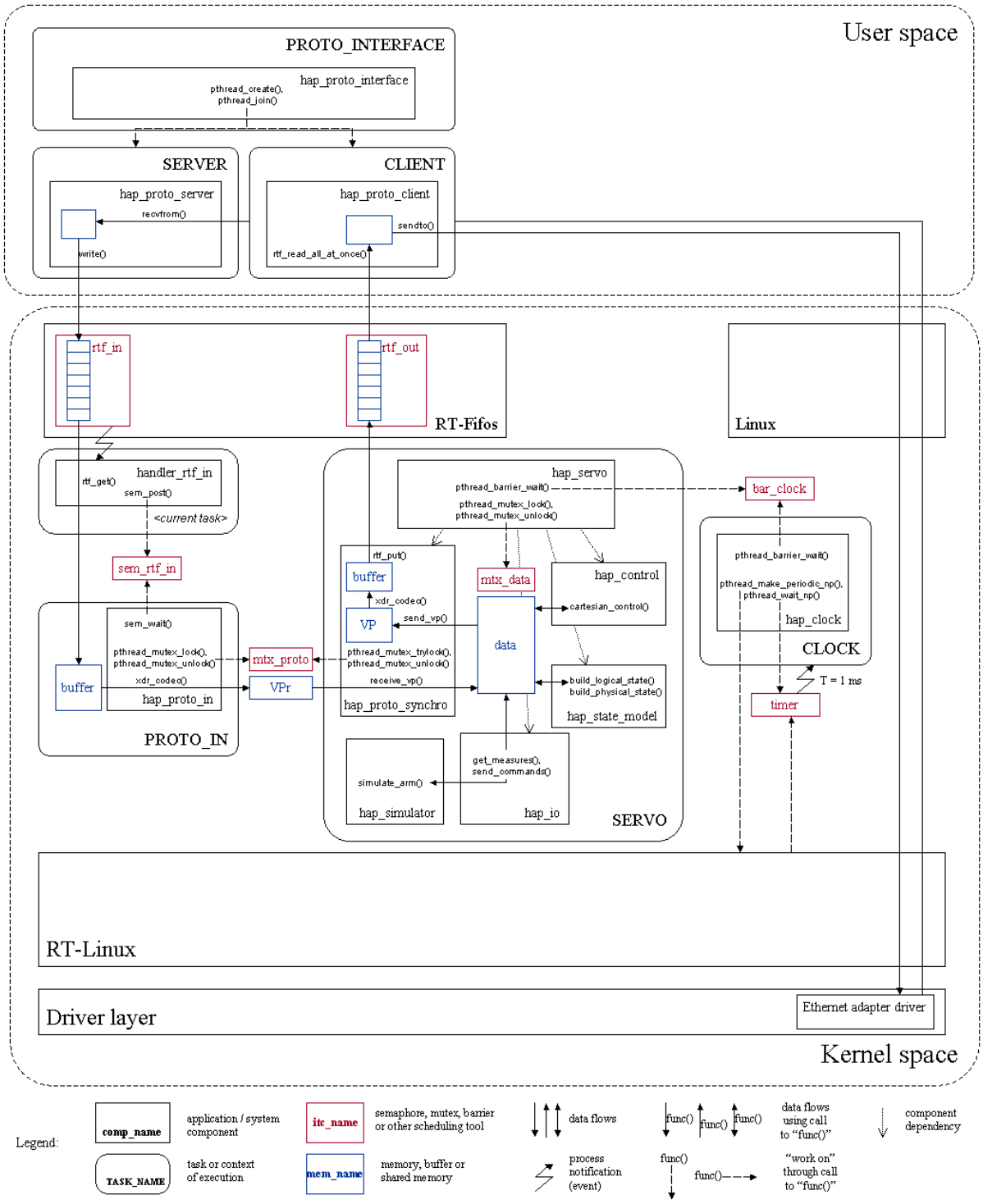

Figure 2 : Servo-control application architecture

3 Main components description

3.1 Client application (virtuose_client)

The Client application consists of an executable linked to a static library called **virtuose_api** that runs in the user space. The **virtuose_api** library provides a complete API set for high level control of the Servo-control application and data transfer at high rate in soft real-time between the Client and the Servo-control applications.

The Client application is intended to be run on a remote computer (though it can be run locally in the machine user space). The Client application is the only task of the Robotic application that can run remotely; all other tasks described in this chapter run on the local machine and are part of the Servo-control application. The **virtuose_api** uses a standard Ethernet connection and the UDP/IP protocol to communicate with the Servo-control application (though no Ethernet adapter is required if the Client application run locally).

The communication API used by **virtuose_api** is the standard Socket one. Data are encapsulated in a specific protocol (named: "haptic protocol") onto UDP/IP. Data transmitted are encoded to then decoded from the Sun® XDR® format to avoid data corruption due to communication between different platforms (**virtuose_api** is available for Linux, Win32 and SGI).

The **virtuose_api** API is intended to be linked to an application running the Physical Engine (as described in document D9rb.1); this will be done in the second phase of the OCERA project for Robotic Application V1. For Robotic application V0, the **virtuose_api** will be linked to a test Client application that will simulate the final Physical Engine Client application behavior (all functionalities of the Servo-control application are not fully available yet and thus do not allow to run a true Physical Engine). The test Client application will consist of a graphical user interface representing (in 3D) a virtual world including a pair of spheres which one can be coupled to the (simulated) Virtuose robot arm. The test Client application will also perform the initialization tasks needed by the Servo-control application to run properly : references definition, units definition, application specific internal variables definition and interactive mode initialization.

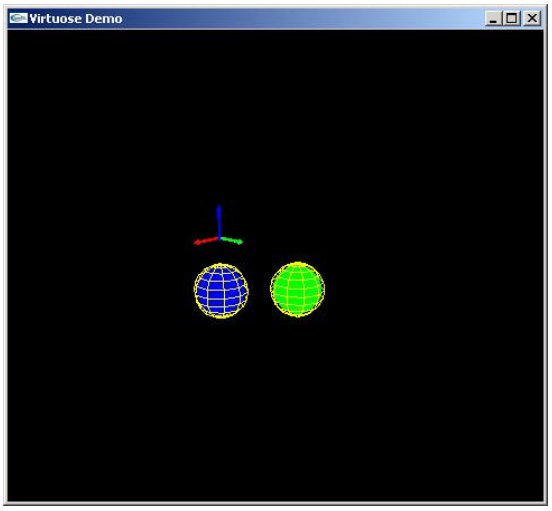

Figure 3 : "Test" Client application V0 snapshot

3.2 Haptic controller (hapticctrl_app)

The Haptic controller component (**hapticctrl_app**) is the top-level component of the Servo-control application dependency tree. It implements the entry point of the kernel application (**init_module()**), the application main initializing function, application specific servo-control functions and the application termination function. The initializing function creates, at the end of initialization phase, the main kernel threads of the application (**CLOCK**, **SERVO** and **PROTO_IN**).

The hapticctrl_app component is not represented on Figure 2 since it is active only at application startup. Figure 2 shows active tasks at runtime only.

The basic tasks carried out by **hapticctrl_app** at application initialization are the following:

- ITCs creation (**rtf_in**, **rtf_out**, **sem_rtf_in**, **mtx_proto**, **mtx_data**, **bar_clock** and other needed RT-FIFOs),
- application components initialization: Communication protocol (**hap_protocol**; this initiates Client connection), Servo (**hap_servo**; this initializes servo-loop parameters), Input/Output (**hap_io**; this does nothing in application V0 but will initialize I/O cards in application V1), State model (**hap_state_model**; this initializes internal variables needed to compute logical and physical states), Control (**hap_control**; this initializes controller filters), Virtuose simulator (**hap_simulator**; this initializes initial state of the simulator) and all required dependencies,
- application specific variables/parameters definition and setting (references, units and internal variables),
- initial (simulated) robot arm positions calibration,
- **CLOCK**, **SERVO** and **PROTO_IN** kernel tasks creation.

The basic tasks carried at by **hapticctrl_app** at application termination are the following:

- Client connection closing (**hap_protocol**),
- **SERVO** and **PROTO_IN** kernel tasks deletion,
  [the following tasks may be performed at module unload (**cleanup_module()**)]
- ITCs destruction,
- dynamic memory deallocation (most of the application components dynamically allocate data to store internal structures at initialization phase).

The API used to start the kernel threads is the standard RTLinux/Free one except for the SERVO task which will be run using the Fault-Tolerance OCERA component (see § 3.5):

- CLOCK: **pthread_create()** then **pthread_make_periodic()** (periodic task; T = 1 ms),
- SERVO: **ft_task_init()** and **ft_task_create()** (one shot mode task),
- PROTO_IN: **pthread_create()** (one shot mode task).

The **hapticctrl_app** component is compiled as a Linux kernel module (except the **hap_proto_interface** which is compiled as a standard user-space executable; see § 3.3); only this application kernel module has to be inserted into the Linux kernel. The Servo-control application will be launched using a shell script; the script will run in the background the **hap_proto_interface** component and then insert the **hapticctrl_app** module into the kernel.

3.3 Communication protocol (hap_protocol)

The goal of the Communication protocol component (**hap_protocol**) is to provide communication facilities between the Client application and the Servo-control application.

The tasks carried out by this component are the following:

- Receive and interpret control frames transmitted from the Client application through Ethernet (UDP/IP) to the controller,
- decode frames from the Sun® XDR® format to the platform internal format,
- store the decoded data into a shared memory for use by the SERVO task (see § 3.5).
- Collect answer-data from the Servo-control component ,
- encode data from the platform internal format to the Sun® XDR® format,
- send the answer frames to the Client interface through Ethernet.

The **hap_protocol** component is divided into 4 sub-components described below. The **hap_proto_server** and **hap_proto_client** components are part of the **hap_proto_interface** sub-component which is compiled as a standard Linux executable.

3.3.1.1 Server (hap_proto_server)

The communication protocol server (**hap_proto_server**) runs as a separate thread in the Linux user space (**SERVER**). Its main goal is to serve UDP requests from the Client application. It is launched at initialization as a standard POSIX thread by the **hap_proto_interface** component main task (**pthread_create()**).

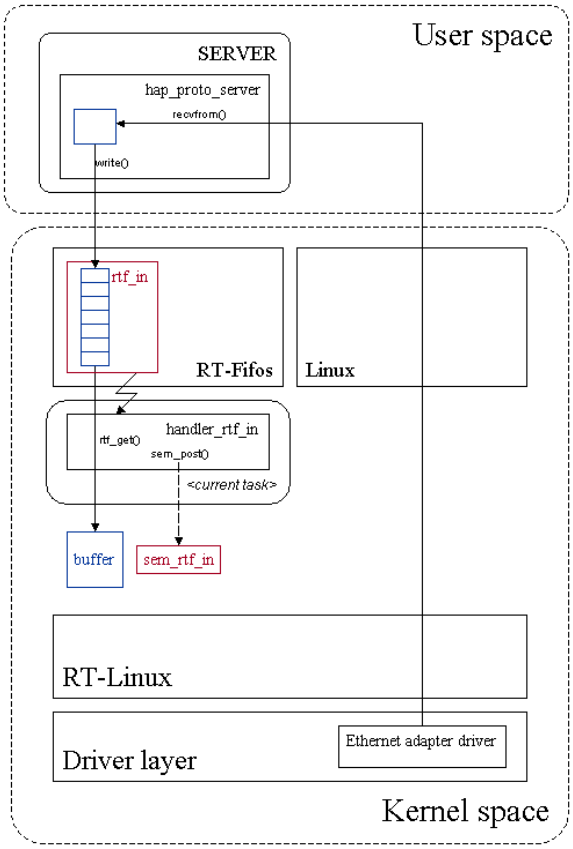

Figure 4 : hap_proto_server component architecture

The basic tasks carried out by **hap_proto_server** are the following:

- Listen to the Client application requests on a socket opened at initialization (**recvfrom()**),
- receive a full length control frame from the client (maximum length : 90 bytes),
- transmit the full frame to the Listener sub-component through the **rtf_in** RT-FIFO (**write()**),
- loop.

As soon as a data is available in the **rtf_in** FIFO, a handler associated with the FIFO transfers the data to a secured buffer in kernel space (**rtf_get()**) and increments a counter by the number of bytes sent. When the counter reaches the expected frame length, the handler posts the **sem_rtf_in** synchronization semaphore (**sem_post()**) that is initially taken. This releases the **PROTO_IN** task (see below).

3.3.1.2 Listener (hap_proto_in)

The communication protocol listener (**hap_proto_in**) runs as a separate real-time thread in the kernel space (**PROTO_IN**). Its main goal is to interpret high-level commands from the control frame, to decode XDR encapsulated data and to perform a CRC on data. The **PROTO_IN** task has the lower priority of the RT tasks of the application.

As the **xdr_codec()** function (that encodes and decodes XDR data) is non reentrant, calls to this function are protected by a mutex semaphore (**mtx_proto**). **mtx_proto** also protects concurrent accesses to the **VPr** shared memory (see below).

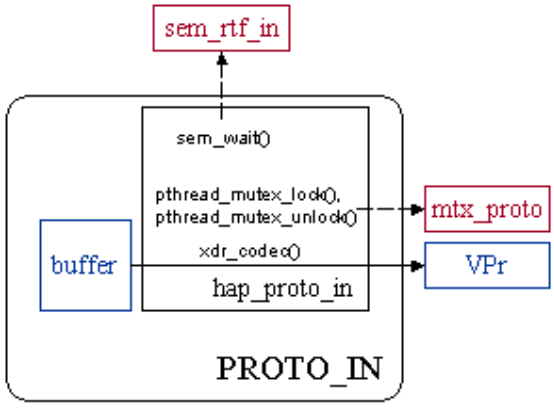

Figure 5 : hap_proto_in component architecture

The basic tasks carried out by **hap_proto_in** are the following:

- Wait for the **sem_rtf_in** synchronization semaphore (**sem_wait()**) to be posted by the **rtf_in** handler,
- take the **mtx_proto** mutex semaphore that grants exclusive access to protocol (**pthread_mutex_lock()**),
- copy the control word and decode the encoded data from the buffer to the **VPr** shared memory (**xdr_codec()**),
- post the **mtx_proto** mutex semaphore (**pthread_mutex_unlock()**),
- take the **sem_rtf_in** synchronization semaphore,
- loop.

3.3.1.3 Synchronizer (hap_proto_synchro)

The communication protocol synchronizer (**hap_proto_synchro**) runs in the **SERVO** task context. It is called by the **hap_servo** component within the synchronous cycle of the **SERVO** task (see § 3.5) and acts as the Client application / Servo-control application synchronizer.

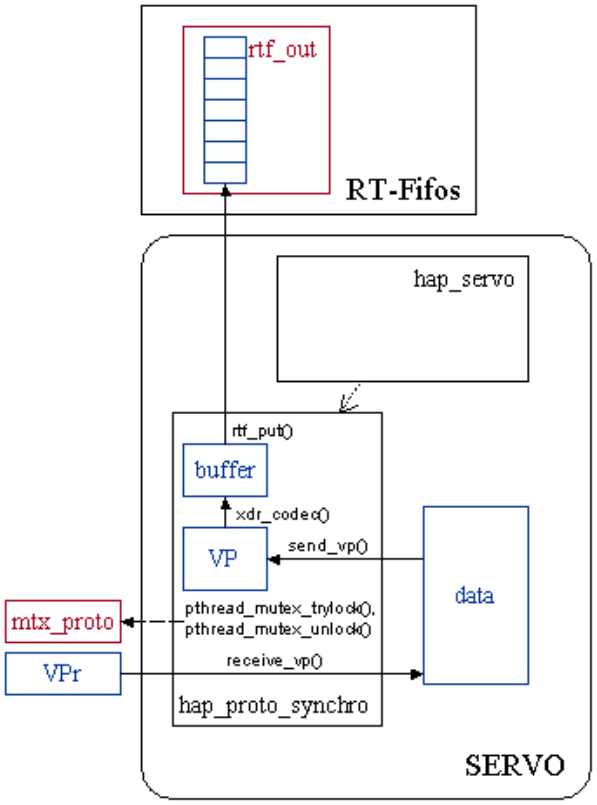

Figure 6 : hap_proto_synchro component architecture

The basic tasks carried out by **hap_proto_synchro** are the following:

- Try to lock the **mtx_proto** mutex semaphore (**pthread_mutex_trylock()**)
- if **mtx_proto** has been successfully locked : copy data from **VPr** to the **data** shared memory (**receive_vp()**),
  else : compute the best estimate of **VPr** and store result into the **data** shared memory (**hap_servo** input data),
- unlock the **mtx_proto** mutex if previously locked (**pthread_mutex_unlock()**),
- copy **hap_servo** output data from the **data** shared memory to the **VP** memory (**send_vp()**),
- try to lock the **mtx_proto** mutex semaphore (**pthread_mutex_trylock()**)
- if **mtx_proto** has been successfully locked :
  - XDR encode data in **VP** to a buffer (**xdr_codec()**),
  - generate answer frame from data (haptic protocol) and send it to the **rtf_out** FIFO (**rtf_put()**),

  else : do nothing,
- unlock the **mtx_proto** mutex if previously locked (**pthread_mutex_unlock()**),
- return from call.

3.3.1.4 Client (hap_proto_client)

The communication protocol client (**hap_proto_client**) runs as a separate thread in the Linux user space. Its main goal is to send data coming from **hap_proto_synchro** to the remote Client application using UDP/IP through an opened socket on the client machine. It is launched at initialization as a standard POSIX thread by the **hap_proto_interface** component main task (**pthread_create**()).

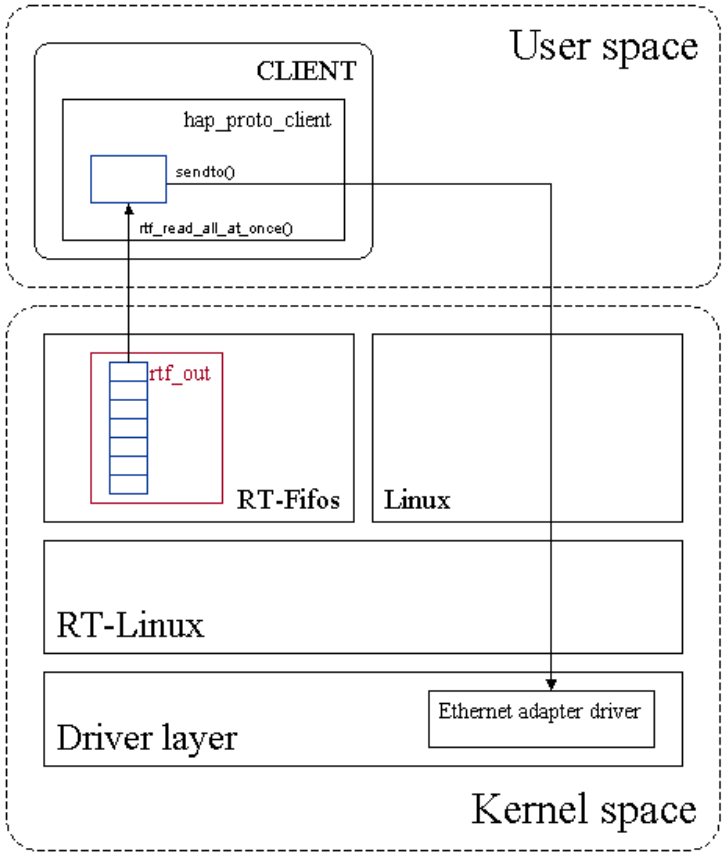

Figure 7 : hap_proto_client component architecture

The basic tasks performed by **hap_proto_client** are the following :

- Collect a full-length answer frame from **hap_proto_synchro** through RT-FIFO **rtf_out** (**rtf_read_all_at_once**(); note that this useful and efficient "RTAI" function will be implemented as close as possible to the original one),
- send the full length answer frame to the Client socket (**sendto**()),
- loop.

3.4 Synchronous cycle supervisor (hap_clock)

The goal of the Synchronous cycle supervisor (**hap_clock**) is to give the start top of the synchronous cycle (e.g.: to the **SERVO** task) and to perform benchmarking on execution times and jitters. Since it has been designed for such purpose, the **hap_clock** component can also perform check on execution time of the **SERVO** task and run an alternate task in case of deadline miss (degraded behavior). However, as the OCERA Fault-Tolerance component can provide this mechanism in a more efficient way, it will be used instead, although, the **hap_clock** component will still be used to check an abnormal situation such as: 5 consecutive deadline miss from either the main **SERVO** task (normal behavior) or the auxiliary one (degraded behavior); see below and § 3.5 for details.

The **hap_clock** component runs as a separate real-time task in the kernel space (**CLOCK**). This task has a higher priority than all other tasks of the application. This is also the only periodic task of the application.

At its initialization, the **CLOCK** task performs a **pthread_make_periodic_np()** (in order to make periodic) with a period of 1 millisecond and then sleep for a while (using **usleep()**) in order for the **SERVO** task to execute its first code instructions; the **SERVO** task then starts by locking on the **bar_clock** barrier by calling in the first place the **pthread_barrier_wait()** function (see § 3.5).

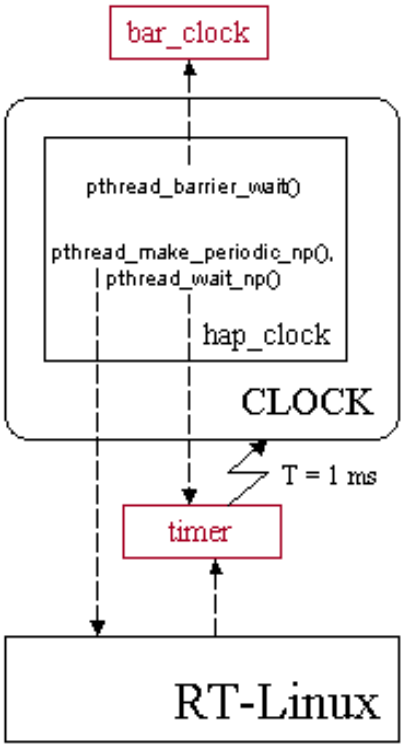

Figure 8 : hap_clock component architecture

An application termination order can be sent to the application from the Linux user space. This feature is implemented using a dedicated RT-FIFO and an associated handler (not represented on figures) that changes state of a global flag (**servo_stop**) as soon as a data is posted in the FIFO. The **servo_stop** flag is set to FALSE at application startup and reset to TRUE by the handler when called. It is tested inside the **CLOCK** task loop (see below).

In order to check whether the **SERVO** task is working or not, a global flag (**servo_working**) is used and tested inside the **CLOCK** task loop. The flag is set to **TRUE** at the beginning of the servo-loop and reset to **FALSE** at the end (see below and § 3.5).

The basic tasks performed by **hap_clock** are the following :

- Wake up the **SERVO** task by unlocking the synchronization barrier **bar_clock** using a **pthread_barrier_wait()** call (the **bar_clock** barrier is initialized with a *count* attribute set to **2**),
- sleep for 3/4 of the synchronous cycle period (750 µs = 3/4 * 1 ms) using **usleep()**,
- check that either the main **SERVO** task or the auxiliary one completed its work by testing the global **servo_working** flag (see § 3.5), if it is TRUE, then increments a local counter (**miss_count**), else reset the counter,
- perform jitters computation and update statistics,
- wait for next period (**pthread_wait_np()**),
- if both **servo_stop** is FALSE and **miss_count** is less than 5, then loop to beginning,
  else : inhibit motor commands, power off (simulated) robot supply, execute the application termination function (that will terminate among other things **SERVO** and **PROTO_IN** tasks; see § 3.2) and exit from **CLOCK** task.

3.5 Servo-control (hap_servo)

The Servo-control component (**hap_servo**) and its dependencies is the main component of the application. It implements the generic servo-control functions that are called within the main Servo-control task (**SERVO**) context.

The **SERVO** task runs in kernel space with a greater priority than the other real-time tasks except the **CLOCK** one. It is a one-shot mode task looping indefinitely after initialization phase and performing the successive sequential tasks needed to control the (simulated) Virtuose robot-arm in position, speed and force (this loop is called in the following: "the servo-loop").

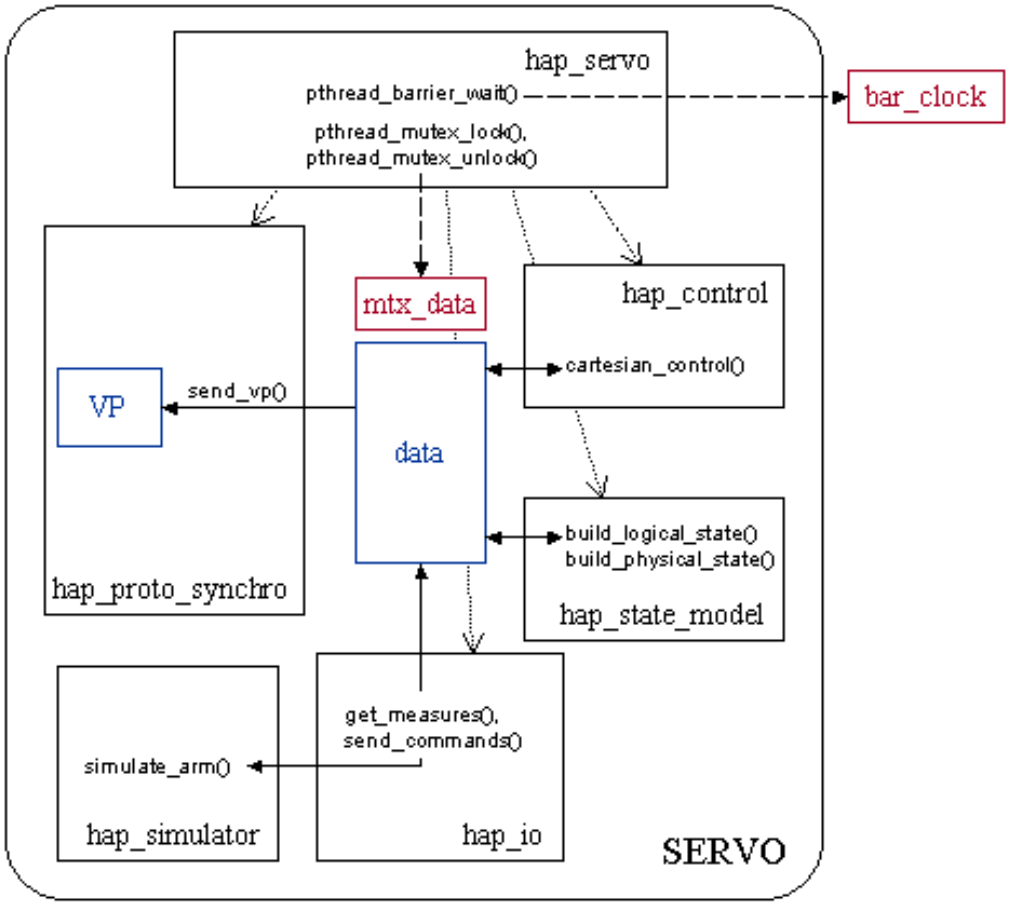

Figure 9 : hap_servo component architecture

The SERVO task will be created using the OCERA Fault-Tolerance API (e.g. : **ft_task_init()** and **ft_task_create()**). The same code will be used for both the main **SERVO** task (normal behavior) and the auxiliary one (degraded behavior); the reason is that the code of **hap_servo** has reached such level of maturity that one can make the assumption that no bug remains in it, so a deadline miss could either occur on hardware failure (e.g.: memory used on the local machine is not ECC compliant and thus could fail) or on some localized/limited strong computation needs.

The basic tasks (e.g.: the servo-loop) carried out by **hap_servo** and its dependencies are the following :

- Wait for **hap_clock** synchronization on **bar_clock** barrier (**pthread_barrier_wait()**),
- set the **servo_working** global flag to TRUE,
- take the **mtx_data** mutex that protects access to the main servo-control internal structure (**data**)[2],
- start acquisition and retrieve position, speed and efforts applied on the (simulated) Virtuose robot-arm by calling the **get_measures()** function (**hap_io** component),
- build the logical and physical states of the robot (**build_logical_state()**, **build_physical_state()**; **hap_state_model** component),
- compute the commands to control the (simulated) Virtuose robot-arm movements and compute the efforts to be applied to the coupled virtual object (**cartesian_control()**; **hap_control** component),
- send commands to the (simulated) Virtuose robot-arm (**send_commands()**; **hap_io** component),
- call the **send_vp()** function of the **hap_proto_synchro** component that will send the efforts to be applied to the coupled virtual object (see § 3.3.1.3),
- post the **mtx-data** mutex to allow other tasks to access the **data** structure,
- reset the **servo_working** flag to FALSE,
- loop.

3.6 Input/Output (hap_io) and Virtuose simulator (hap_simulator)

The Input/Output component (**hap_io**) is intended to implement the functions that control the hardware (Input/Output PC104 cards); this will be done for Robotic application V1 since it will work on true hardware. For application V0, the **hap_io** component will redirect **get_measures()** and **send_commands()**[3] functions (see § 3.5) to and from the Virtuose simulator component (**hap_simulator**) that will be specifically developed for the application V0 purpose. This task is, at now, the only one performed by the **hap_io** component.

The Virtuose simulator component (**hap_simulator**) will be specifically developed to simulate dynamically (but in a simplified way) the behavior of the true Virtuose robot-arm. The component main function (**simulate_arm()**) runs in the context of the SERVO task and is called within the servo-loop (see § 3.5).

A GUI should be specifically developed to interact in (soft) real time with the simulator. It should communicate with it through dedicated RT-FIFOs from the Linux user-space. However, as this development represents a substantial work that is not necessary needed to evaluate Robotic application V0, this component design could be postpone to the next project period. In such case, the GUI would be replaced by an equivalent off-line result analysis. The following figure shows a snap-shot of the GUI now in pre-alpha test version available for Win32 only at now.

---

[2] indeed, this protection mechanism will not be used for application V0 since no other task accesses the **data** structure ; however, it will be used in the future.

[3] the **send_command()** function is inhibited until a power-on request coming from the Virtuose robot-arm ; this will be simulated using a dedicated RT-FIFO, an associated handler and a global flag.

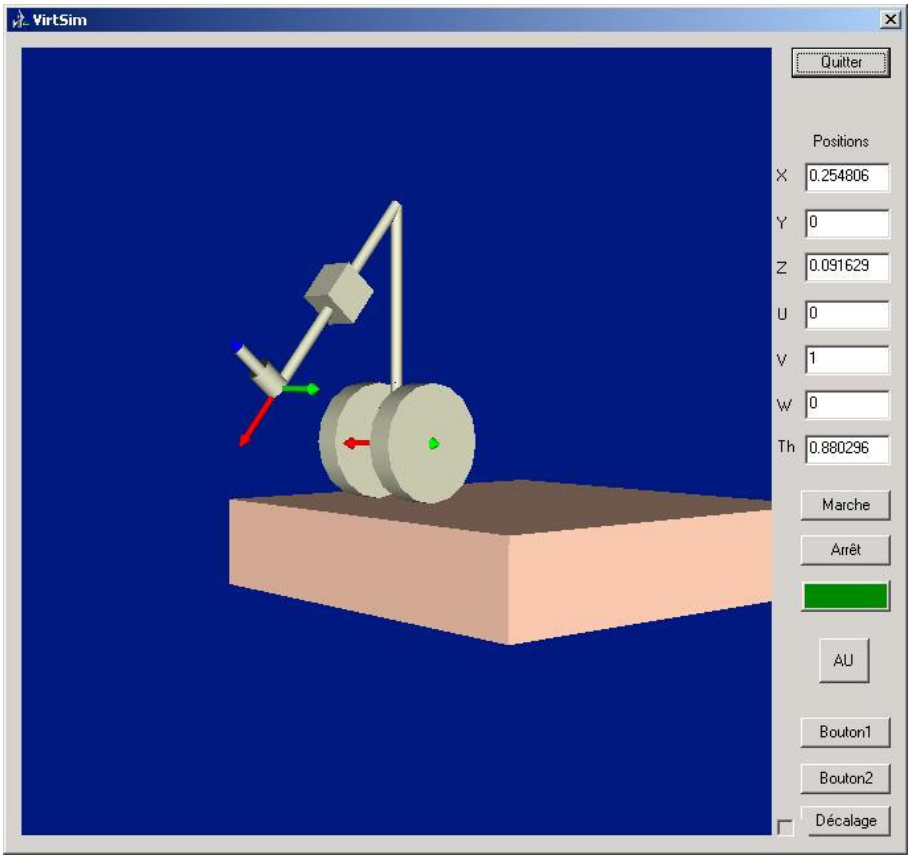

Figure 10 : Virtuose robot-arm simulator GUI

The basic tasks carried out by **hap_simulator** are the following:

- Get efforts applied by user on the (simulated) Virtuose robot-arm handle (this will be done through the GUI if implemented or from a pre-computed sequence loaded at application startup if not),
- compute movement of the (simulated) Virtuose handle resulting from user efforts applied on it and commands applied to the (simulated) Virtuose motors through **hap_io**,
- send computed position to the GUI (if implemented; or log to a buffer otherwise) that will refresh image,
- return the computed position and speed to **hap_io**.

4 Requirements

Document D9rb.1 defined requirements relative to the operating systems and OCERA components in order for the robotic application to run properly. The goal of this chapter is to define requirements relative to the application itself.

4.1 Functional requirements

The main requirement associated to the Robotic application is at first a correct behavior, as described earlier. That is to say: correct application initializing, working and shutdown, following the mechanisms described in the previous chapter.

Correct behavior of the application will be checked through the following phase sequence:

- At kernel module loading, the application should start and initialize properly, then, it should immediately enter the synchronous cycle and stay in this state indefinitely. Motor commands to the (simulated) Virtuose motors should be inhibited until a power-on request coming from the (simulated) Virtuose robot-arm is received.
- At Virtuose robot-arm power-on, the motor commands should be immediately enabled and the servo-loop within the synchronous cycle should work properly allowing bilateral cartesian coupling of the (simulated) Virtuose robot-arm with a virtual object running in the test Client application. Movements and coupling effects (force feedback) should be observable from both the Servo-control application and the Client application.
- The application should stay in the synchronous cycle until an application termination request is received or if exactly 5 consecutive synchronous cycle deadline miss occur. In case of such event, the application should go in a fail-safe state carrying the following tasks out (in this order): inhibit motor commands, power off (simulated) robot supply, close protocol (e.g.: communication), terminate all kernel tasks (except the one active if any) and perform all other needed jobs to terminate properly the application.

4.2 Performance requirements

As the platform used for application V0 is more powerful than the one that will be used for application V1, the obtained performances should be higher than the one specified in document D9rb.1. However, we will check the following figures:

- A full synchronous hard real-time cycle should not exceed 500 µs.
- The SERVO task should start computing at the latest 50 µs after CLOCK task wake-up.
- No missed deadline should be observed within the hard real-time cycle, even while user space is heavily loaded by a Linux task.
- Data transfer rate between the Servo-control application and the Client application should be at least: 1 sample / 3 ms, continuous, with less than 1 sample lost on 10 (long term).

5 Acronyms

ACPI	Advanced Configuration and Power Interface
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
APM	Advanced Power Management
CRC	Circular Redundancy Checksum
DMA	Direct Memory Access
DoF	Degrees of Freedom
FIFO	First In First Out (ITC mechanism)
GUI	Graphical User Interface
IP	Internet Protocol
ITC	Inter-Tasks Communication
MUTEX	MUTual-EXclusion semaphore
OCERA	Open Components for Embedded Real-time Applications
PCI	Peripheral Component Interconnect
POSIX	Portable Operating System Interface
RT	Real-Time
UDP	User Datagram Protocol
XDR	eXternal Data Representation (Sun microsystems)