ERAS Documentation

Release 0

IMS

August 29, 2015

Contents

High level design docs:

CHAPTER 1

Mission Operations Control Center - High Level Design Documentation

Author Mario Tambos

1.1 Change Record

2015.06.16 - Document created.

1.2 Introduction

1.2.1 How to use this document

The objectives of the present document are:

- provide the reader with an overview of how the Mission Operations Control Center is organized,
- facilitate communication between team members by stipulating a common vocabulary,
- aid development and maintenance by specifying and limiting the concerns of each of the Mission Operations Control Center's components.

This document should be the entry point of any new collaborator to the ERAS project looking to work with the *MOCC*. From here, such collaborator can choose to continue reading the *MOCC*'s implementation, infrastructure or management documents, depending on whether the reader is interested in, respectively, developing software, aiding with the underlying infrastructure or helping with the human resources side of the *MOCC*. Also relevant for the reader would be the documents specific to the subsystem he/she wants to contribute to.

1.2.2 Purpose

The Mission Operations Control Center (MOCC) is in charge of planning the missions performed in European MaRs Analogue Station for Advanced Technologies Integration (ERAS), and following said plan.

In this context, a **plan** is understood to be a sequence of activities, together with the expected outcomes of these activities. By **following a plan** is meant the execution of the plan, the control of its progress as well as the countermeasures needed to correct eventual deviations from the individual activity's expected outcomes.

1.2.3 Reference Documents

- [1] ++ C3 Prototype document v.4
- [2] ++ Software Engineering Practices Guidelines for the ERAS Project
- [3] ++ Dachstein 2012 Mission Report
- [4] ++ Morocco MARS2013 Mission Report
- [5] ++ V-ERAS Project Description (2014 Release)
- [6] ++ V-ERAS-14 Mission Report

1.2.4 Glossary

AI Artificial Intelligence

ERAS European Mars Analog Station

EVA Extra-Vehicular Activity

GUI Graphic User Interface

IMS Italian Mars Society

MOCC Mission Operations Control Center

TBC To Be confirmed

TBD To Be Defined

UI User Interface

1.2.5 Overview

The *first section* of this document describes at the highest level the organization of the *MOCC*, its communication channels, subsystems and their general responsibilities, as well as the assumptions made during the design process. The *second section* describes the documents associated with the present one. Finally, *Section 3.* deals with miscellaneous factors that need to be addressed or acknowledged when implementing or operating the *MOCC*.

1.3 1. The MOCC System and its Subsystems

1.3.1 High level architecture

The *MOCC* is divided in five subsystems with clearly separated responsibilities, as shown in *Figure 1*.

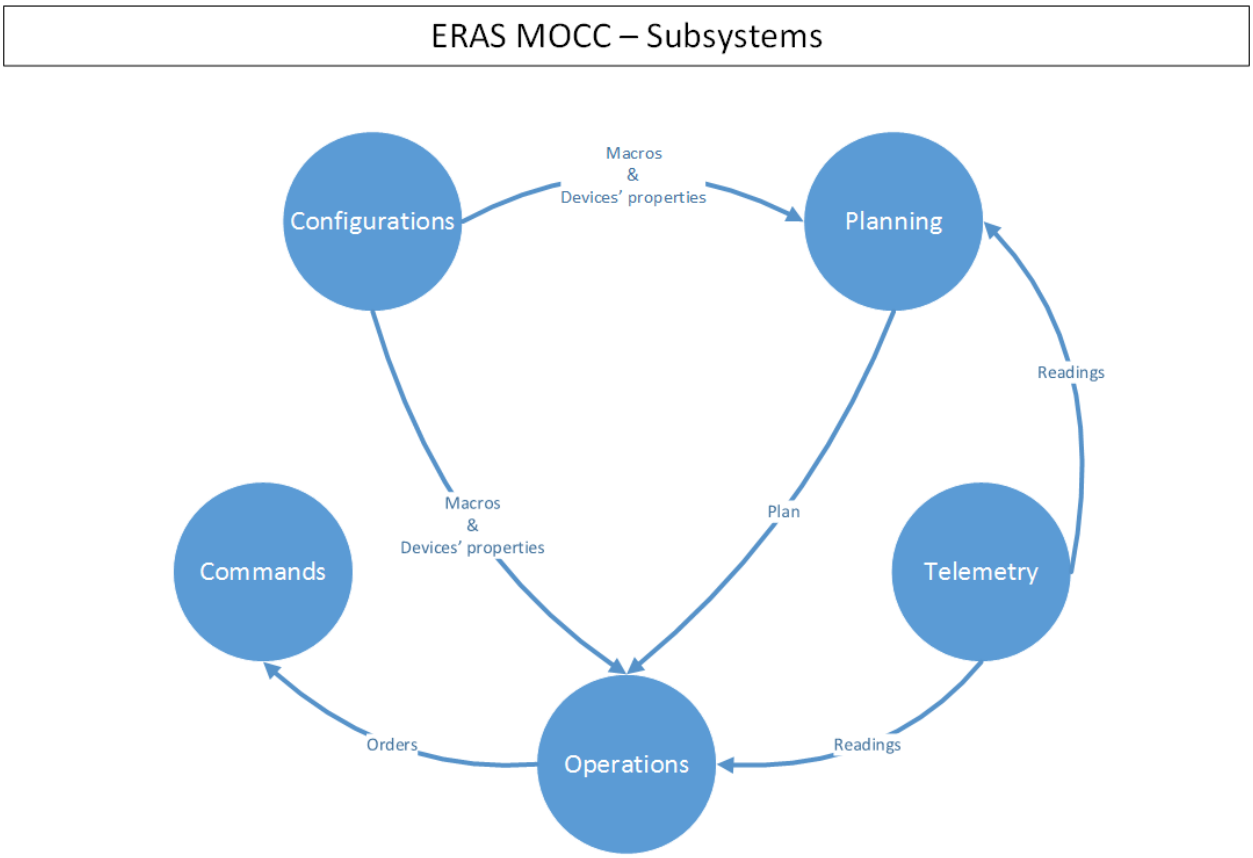


Fig. 1.1: Figure 1. The MOCC architecture

The design's building blocks are **systems**, **subsystems**, **components** and **communication channels**.

The *MOCC* is considered to be a system in itself, which is divided in subsystems. Any external services the MOCC may interface with are also considered systems.

A component is a software artifact that performs certain function. Each subsystem is built of components; no component belongs directly to the *MOCC* or to more than one subsystem, and there is no part of any subsystem that is not a component.

Finally, communication channels represent all the media and supporting infrastructure needed to allow the exchange of information between the systems, subsystem and components.

The main philosophies behind the design are those of extensibility, separation of concerns, and fault-tolerance: it should be easy to add a new component, it should also be easy to determine where the new component belongs; and, finally, any subsystem should remain functional in the event of failure of all the other subsystems.

1.3.2 Entities

The entities in the The *MOCC* are classified according to two criteria. The first is whether they are controlled by the *MOCC*, or if they are outside the *MOCC*'s control. Controlled entities are called **internal**, whereas non-controlled entities are called **external**.

The second criterion discriminates between so called *basic* and *composite* entities. There are four types of basic entities:

- Actors: crew and electromechanical devices able to perform actions.
- Observers: crew and electromechanical devices able to provide information.
- Analysts: *AI* agents able to deduce information from observations.
- Users: crew operating the *MOCC*.

On the other hand, composite entities encompass any and all entities that fulfill functions of several basic entities (e.g. actor and observer), for instance:

- Astronauts
- Rovers
- Satellites
- Etc.

1.3.3 Interfaces and Communication Channels

There are four kinds of interfaces to be considered:

- **External** interfaces, between internal and external entities, or vice-versa.
- **Subsystem** interfaces, between subsystems.
- **Component** interfaces, between components of a subsystem.
- **User** interfaces, between components and users.

1.3.4 Subsystems

The core of the *MOCC*'s tasks is performed by two subsystems: the **Planning** and the **Operations** subsystems, with the other three subsystems fulfilling support roles.

The Planning Subsystem

One of the two core subsystems of the :MOCC:, the Planning subsystem is in charge of defining the scope and expected results of ERAS's missions. The plans this subsystem creates are composed of a series of **steps**, performed by actor-entities, and **parameters**. Each step along the way has an **expected outcome**, which can be checked during the plan's execution. The plans can be composed of serial steps, parallel steps, or any combination thereof. By parameters is understood any piece of information relevant to the plan's execution, beside the steps themselves, for instance, the plan's start date, the plan's location, or actors involved.

Plans can be one-offs or periodical. In the case of one-off plans a fixed start date is set;M in the case of periodical plans a period is set, e.g., once a day, once every two weeks, etc.

The Planning subsystem comprises two types of components – software and data components. The software components are the **user interfaces** used to build the plans, together with a set of *AI* **assistants**. These assistants should help

the user with the plan building, by analyzing feasibility, evaluating constraints, calculating duration and resources, etc. The data components are **plan templates**, i.e., pre-built plans with free parameters (e.g. no specific resources assigned, or start date set); and **plan instantiations**, which are plans with all their parameters set.

Figure 2 shows the internal structure of this subsystem. In few words, the person in charge of building ERAS 's plans will build plan templates using the user interfaces, with the help of *AI* assistants. When a plan is needed, this person will then create a plan instantiation, using the same *UI* and assistants, by filling in the template's missing parameters. In order to build the templates, the user shall use the information provided by the Configurations Subsystem about the device's and crew member's capabilities, as well as the macros. To allow this, the planning *UI* s must pull this information from the Configurations Subsystem.

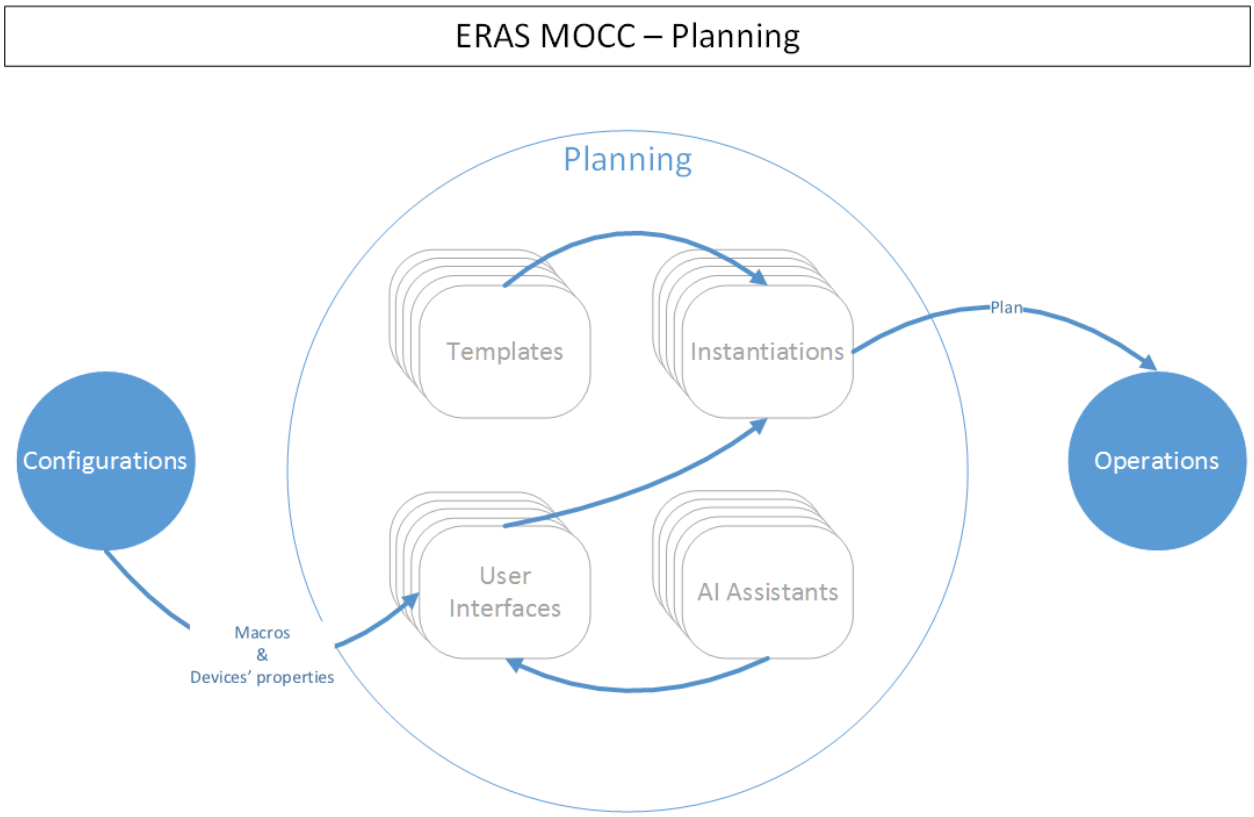


Fig. 1.2: Figure 2. The Planning Subsystem

The Operations Subsystem

This system is the second key component of the *MOCC*. Its tasks are to **execute** a mission's plan, **control** its progress and **correct** any eventual deviations from the plan. The Operations Subsystem's internal structure is shown in *Figure 3*, whereas the interactions between this subsystem's components and the other subsystems in the *MOCC* are shown in *Figure 3.1*, *Figure 3.2*, *Figure 3.3* and *Figure 3.4*.

Executing a plan involves sending commands to devices and crew. Control a plan's execution involves collecting, analyzing and presenting telemetry from devices and crew to the Operations Subsystem users. Finally, correcting deviations from a plan involves:

- Detecting the deviation, based on the telemetry and the plan's step's expected outcomes.
- Devising corrective measures.
- Sending the appropriate corrective commands to devices and/or crew.

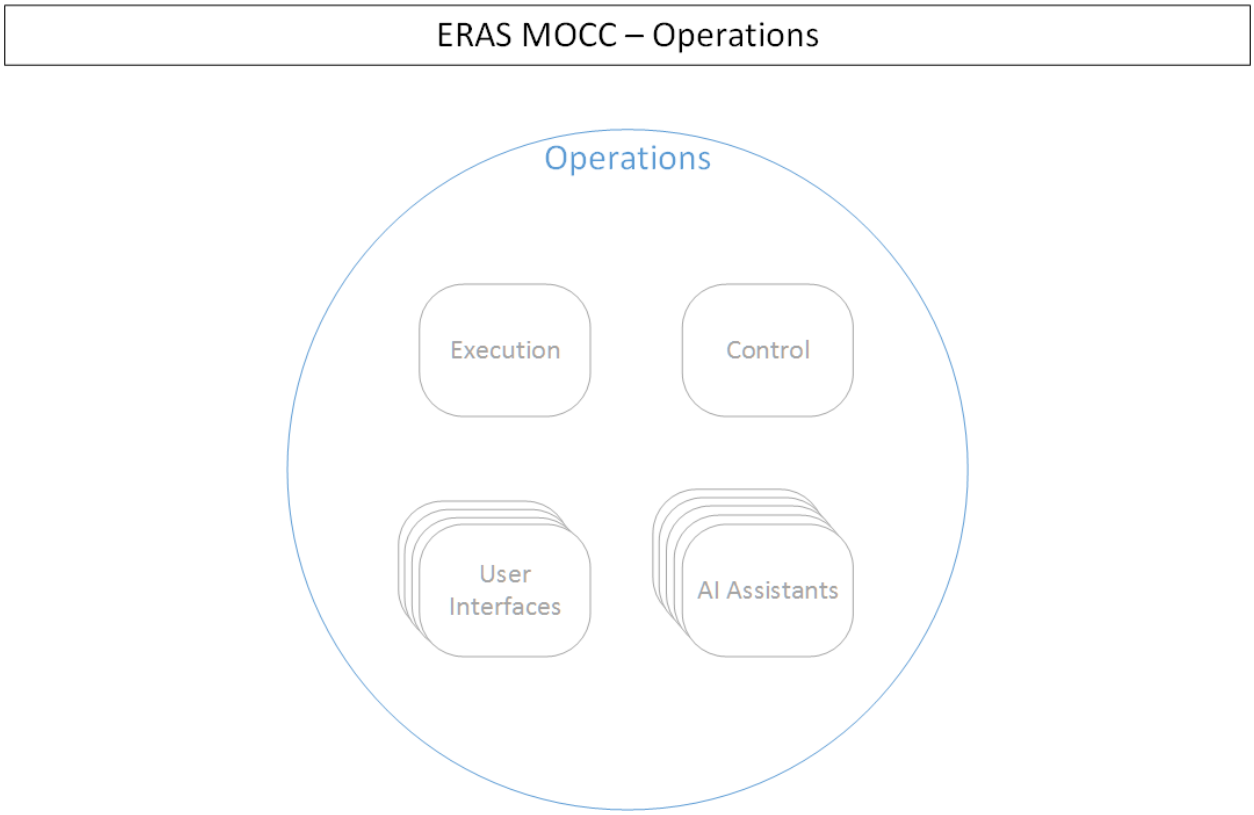


Fig. 1.3: Figure 3. The Operations Subsystem

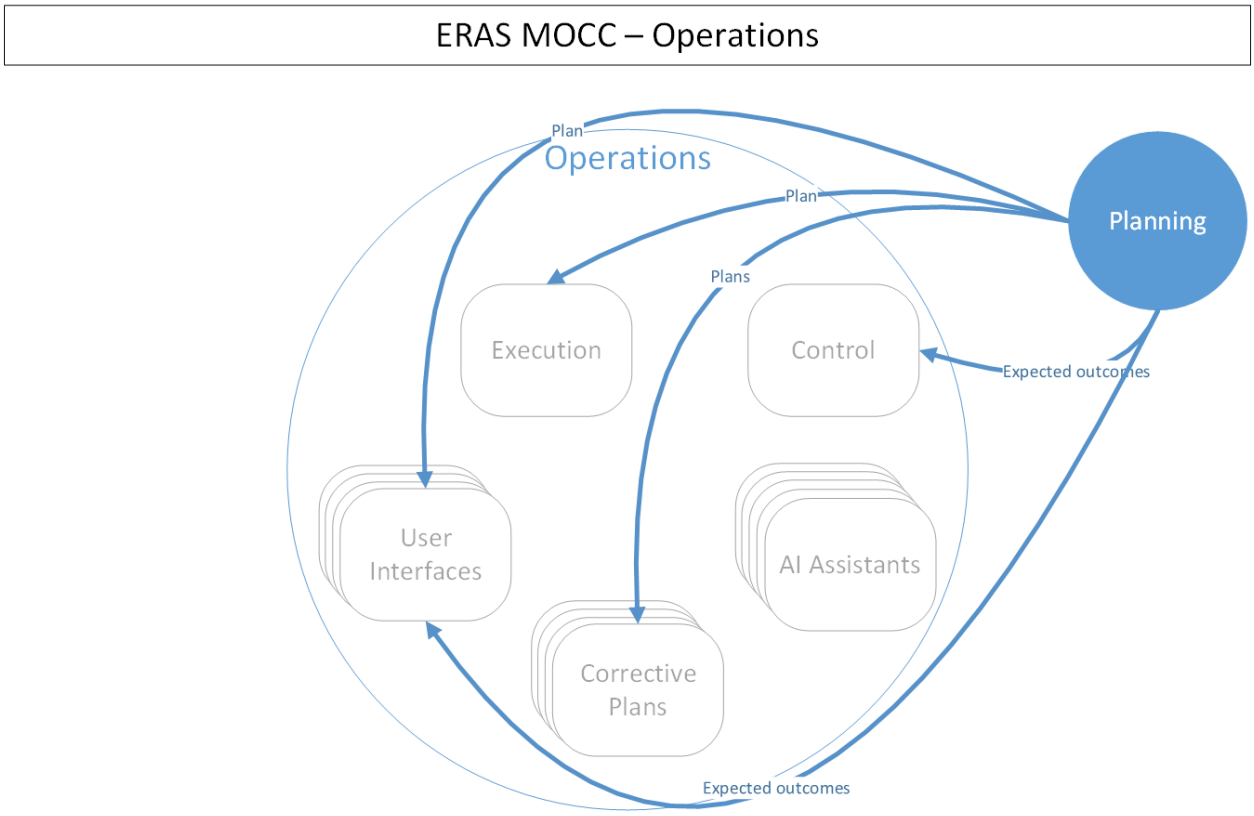


Fig. 1.4: Figure 3.1. Operations interaction with Planning

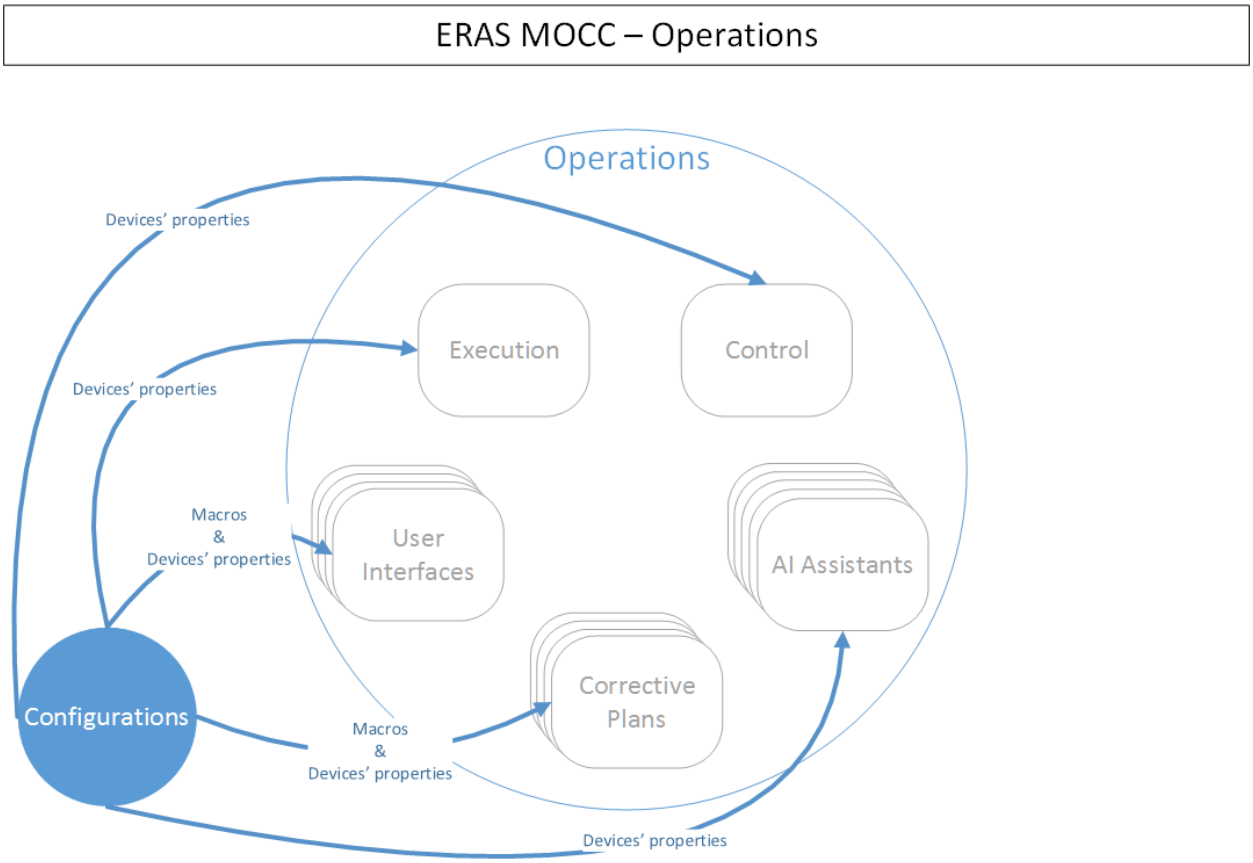


Fig. 1.5: Figure 3.2. Operations interaction with Configurations

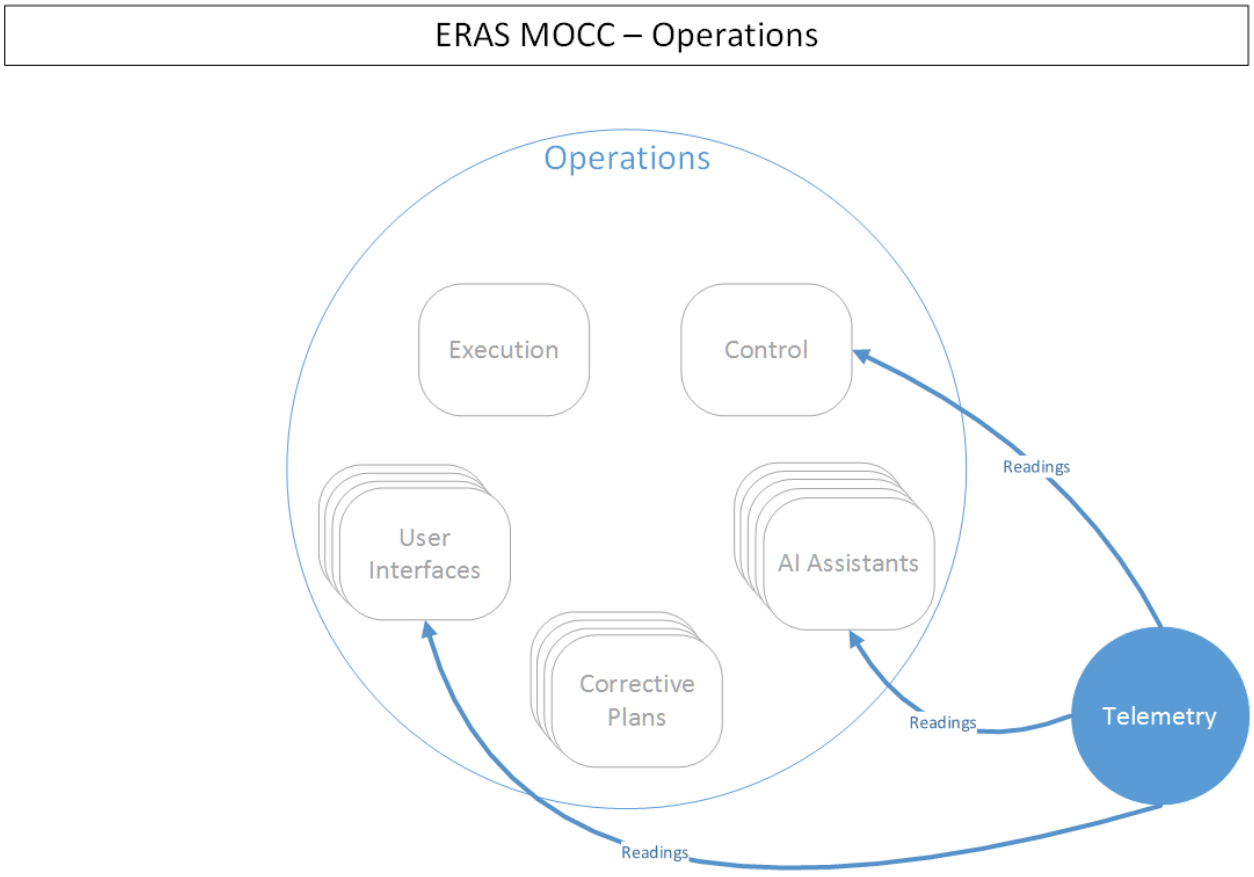


Fig. 1.6: Figure 3.3. Operations interaction with Telemetry

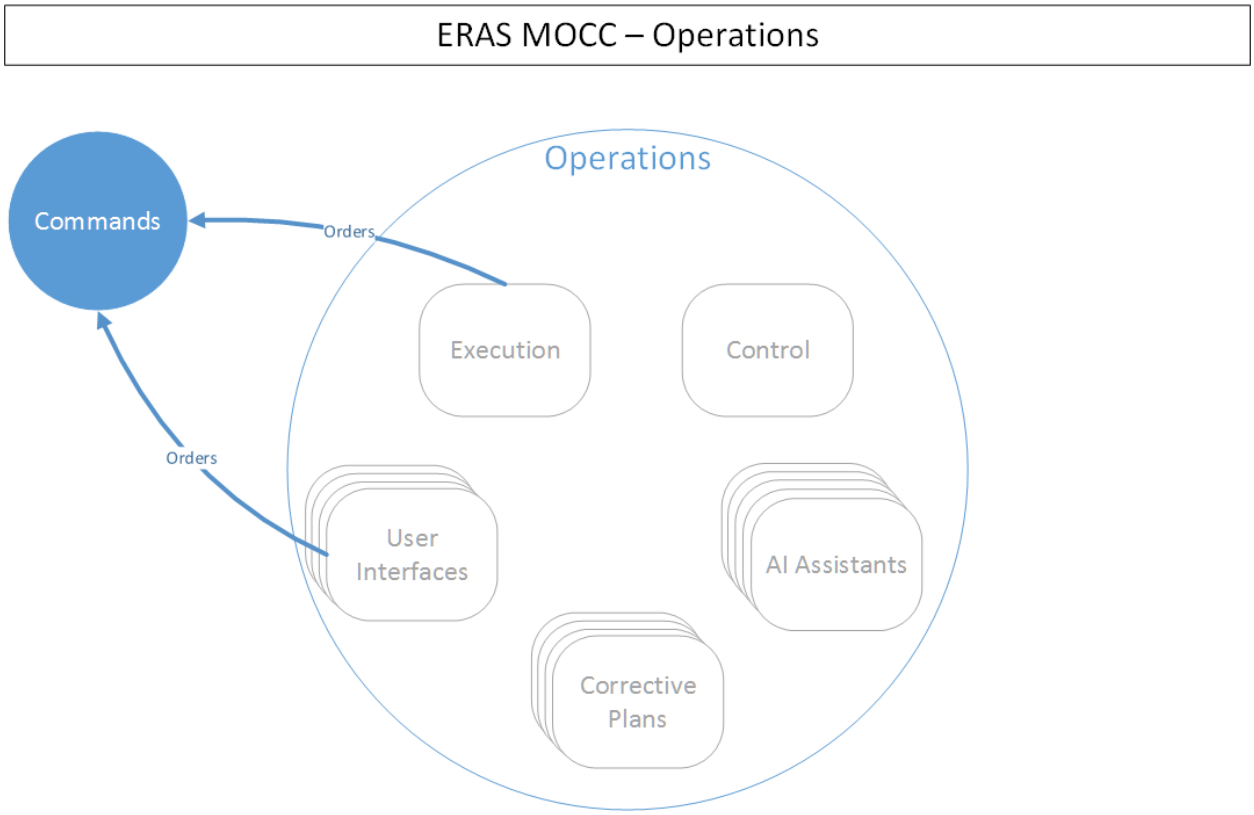


Fig. 1.7: Figure 3.4. Operations interaction with Commands

The **Execution** component is in charge of executing the mission plans. This involves sending all automated commands to devices and crew members at the correct time. For commands that cannot be automatically sent, the Execution component should send a cue to one of the *UI* in order for a user to manually send the command. Part of the Execution component's task is to pull from the Planning Subsystem information about the next plans to execute. Moreover the Execution component needs to pull from the Configurations Subsystem information about the TANGO device addresses and other interface requirements. Finally, the automated commands are sent though the Commands Subsystem.

The **Control** component's task is to check during the plan whether the expected outcomes from the plan in execution match the telemetry readings obtained. If a deviation occurs, the Control component should send an alarm to one of the *UI*, in order to allow the users to perform the necessary corrections. To carry its tasks, the Control component needs to pull from the Configurations Subsystem information about the TANGO device addresses and other interface requirements. The Control component must obtain the device's telemetry readings from the Telemetry Subsystem, whereas the plan steps' expected outcomes come from the Planning Subsystem.

The Operations Subsystem's **user interfaces** have four tasks:

- Show the progress of a plan execution.
- Present the user with the telemetry, and their analysis, collected during a plan's execution.
- Provide the means to send commands outside the plan to the devices and crew.
- Allow the user to build, review and execute corrective measures, in case of a plan deviation.

These *UI* s get the information about the devices from the Configurations Subsystem. They also interface with the Telemetry Subsystem to obtain telemetry readings, as well as with the Commands Subsystem allow manually sending commands. Finally, the *UI* s have to interface with the Planning Subsystem to show the plans and the plans' execution.

The *AI* **assistants** in the Operations Subsystem are of two types:

- **Telemetry Assistants** perform analysis and prediction on subsets of all the telemetry collected. This is used to present summarized information to users, detect problems, project outcomes, etc.
- **Corrective Assistants** help the users build and execute corrective measures, in case of a plan deviation.

The assistants need to interface with the Configurations Subsystem, to obtain devices' addresses and other interface requirements, and with the Telemetry Subsystem, to obtain the devices' readings.

The building of corrective measures mentioned previously can be done by using a plans from a previously built repository of **corrective plans**. This repository should contain plans for correcting common deviations. One such scenario would be as follows:

1. A mission's plan contain a step to drill rock.
2. One of the expected results of the step is that the drill bit's temperature will rise up to 60°C.
3. During operation, the drill bit's temperature rises to 80°C.
4. The user detects the deviation from the expected result (with help from a *UI*, the Control component and possibly a *AI* assistant).
5. To correct the deviation, the user selects a corrective plan for overheating drill bits from the repository, fills the needed parameters, and executes it.

These corrective plans should be built based on both the devices' and crew members' properties, as well as on the original plan.

The Configurations Subsystem

This subsystem is in charge of storing and making available information about the devices and crew that can potentially be involved in a mission, as well as **macros**. A macro is understood to be a series of steps performed by an actor.

The steps in the macro have no expected outcome, however, the whole macro does. Macros differ from plans in that macros have neither a start date. The function of the macros is twofold:

- Help the user build plans and corrective plans by providing a repository of common steps.
- Encapsulate the details of the devices.

The Configurations Subsystem's internal structure is shown in *Figure 4*.

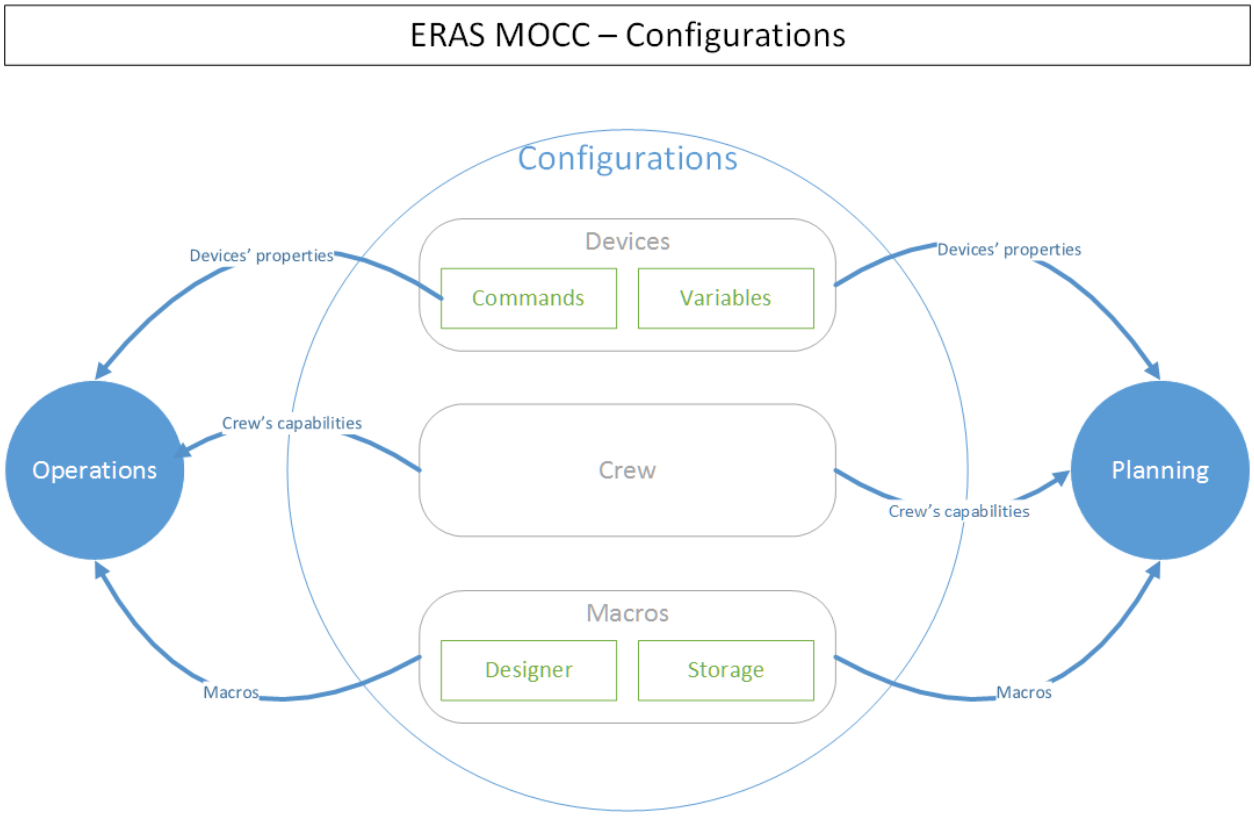


Fig. 1.8: Figure 4. The Configurations Subsystem

Therefore a macro is a higher level interface to the devices. For instance, consider a situation where a user needs to setup an RF chain to receive satellite telemetry. Without macros, the plan should include steps for setting up every device in the chain, with the consequent need to know of the details of those devices. With macros, the user could request an RF setup-macro and use that instead, without having to have knowledge of how the devices in the chain need to be setup.

The Configurations Subsystem comprises three components:

- The **Devices** component stores and provides information about the commands a device can receive and how those commands must be sent: channel the command must be sent through, input parameter's types and restrictions, etc. This component also handles information about the telemetry a device is able to provide, in the form of a device's **variables**. This comprises how to obtain these variables, and what format they have.
- The **Crew** component stores and provides information about the crew member's mission capabilities, e.g., crew member *A* can perform geological analysis, crew member *B* can perform mechanical repairs, crew member *C* can perform first aid, etc.
- Finally, the **Macros** component handles the design, storage and availability of macros. This component contains two subcomponents:

- A **Designer**, which is a *UI* that allows user to build macros.
- A **Storage** subcomponent, which saves and makes available the macros in existence.

The Telemetry and Commands Subsystems

These subsystems are in charge of obtaining data from and delivering commands to devices, respectively. Both have a similar internal structure, shown in *Figure 5* and *Figure 6*.

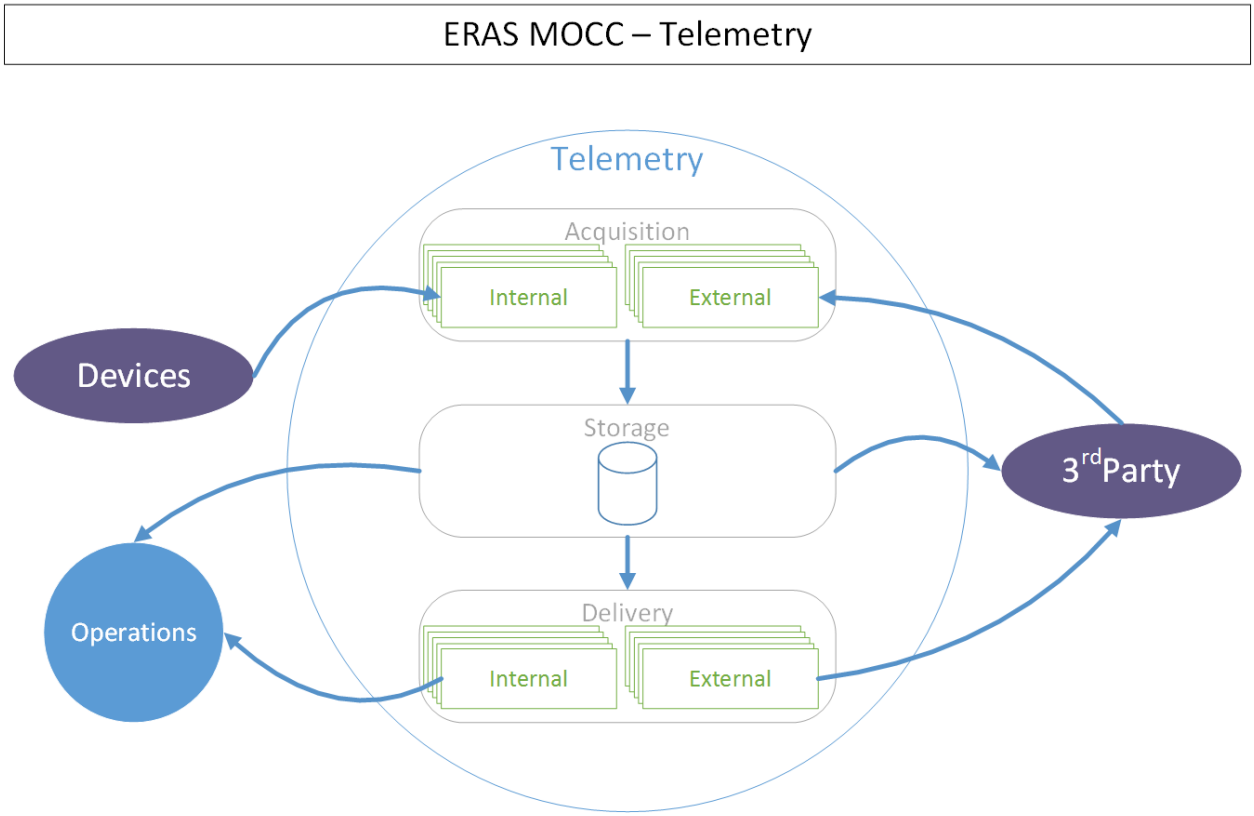


Fig. 1.9: Figure 5. The Telemetry Subsystem

Both subsystems have three components:

- **Acquisition**, which obtains data from devices, in the case of the Telemetry subsystem, and commands from other subsystems, in the case of the Commands subsystem.
- **Storage**, which keeps a historical record of the telemetry read and commands sent.
- **Delivery**, which sends commands to devices and crew, in the case of the Commands subsystem, and sends data to other subsystems, in the case of the Telemetry subsystem.

These subsystems differentiate between two kind of clients. **Internal** clients are other subsystem in the *MOCC*, whereas **external** clients are those outside the *MOCC*.

1.4 2. Documents

The high level documentation of the *MOCC* System comprises 10 documents. Five of those cover the whole system:

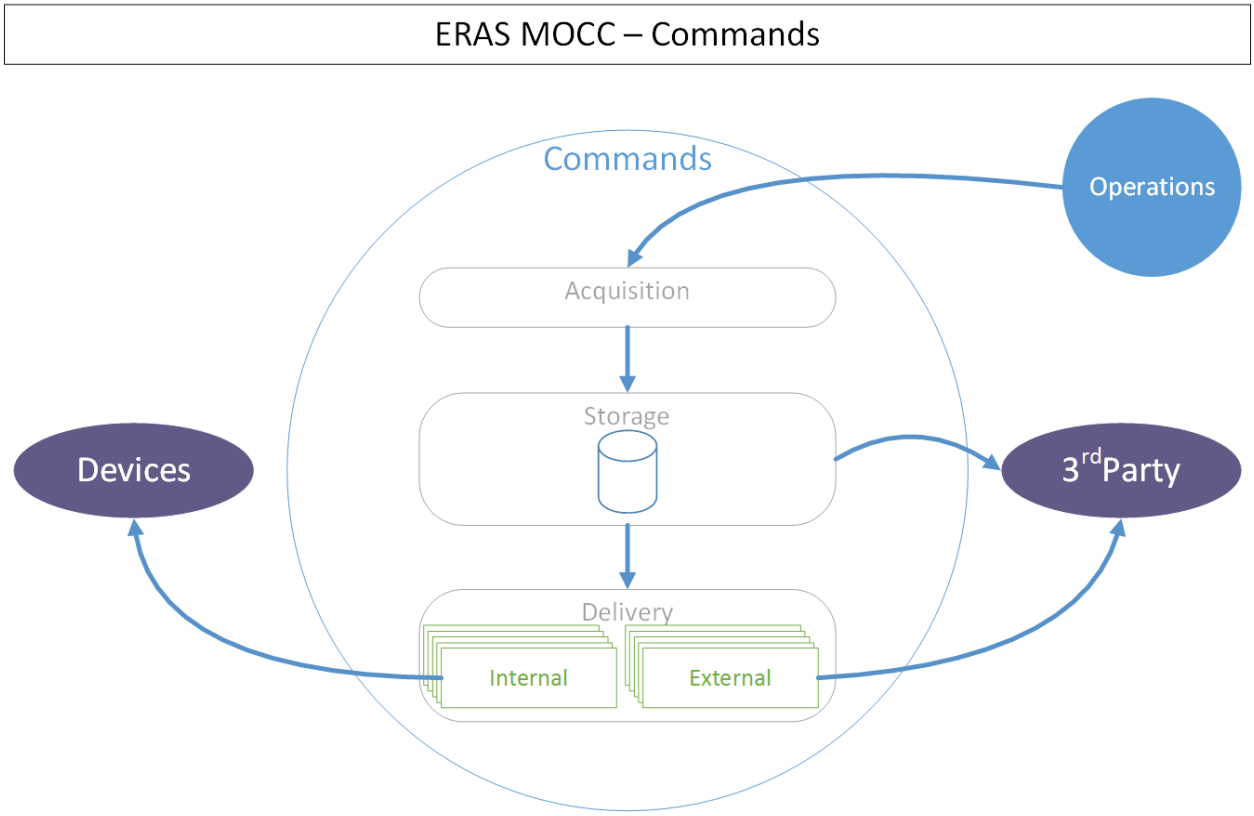


Fig. 1.10: Figure 6. The Commands Subsystem

1. The present document, which gives an overview of all aspects of the whole system.
2. The *MOCC* Implementation Document.
3. The *MOCC* Infrastructure Document.
4. The *MOCC* Management Document.

System-wide restrictions on software, infrastructure or human resources should go in these documents.

The other five deal with each subsystem:

1. The Planning Subsystem Design Document.
2. The Operations Subsystem Design Document.
3. The Configurations Subsystem Design Document.
4. The Telemetry Subsystem Design Document.
5. The Commands Subsystem Design Document.

These last five documents expand on each aspect of the general documents, explaining vague points and refining the granularity of the system-level design. Subsystem-specific restrictions on software, infrastructure or human resources should go in their corresponding subsystem document.

1.5 3. Special Considerations

TBD

CHAPTER 2

Mission Operations Control Center - Implementation Documentation

Author Mario Tambos

2.1 Change Record

2015.06.22 - Document created.

2.2 Introduction

The present document means to laid the foundations for building all the *MOCC*'s software. In doing so this document will specify how the subsystems and components defined in [1] are to be implemented.

2.2.1 How to use this document

This document is meant primarily for software developers. ERAS's contributors wanting to write code for the *MOCC* should first read [1]. The contributor should then read this document, to familiarize him/her-self with the general implementation rules of the *MOCC*, and finally the document corresponding to the subsystem he/she wants to contribute for, if available.

If a detail is left out of this document, the reader should check the subsystem documents. Failing that, those details are left to interpretation.

2.2.2 Reference Documents

- [1] ++ Mission Operations Control Center - High Level Design Documentation
- [2] ++ C3 Prototype document v.4
- [3] ++ Software Engineering Practices Guidelines for the ERAS Project
- [4] ++ Dachstein 2012 Mission Report
- [5] ++ Morocco MARS2013 Mission Report
- [6] ++ V-ERAS Project Description (2014 Release)
- [7] ++ V-ERAS-14 Mission Report
- [8] ++ TANGO Controls
- [9] ++ EUROPA framework
- [10] ++ PANIC - The Package for Alarms and Notification of Incidents from Controls
- [11] ++ ERAS' Tango Devices' JSON Interface Definition

2.2.3 Glossary

AI Artificial Intelligence

ERAS European Mars Analog Station

EVA Extra-Vehicular Activity

GUI Graphic User Interface

IMS Italian Mars Society

MARS Mission Asset and Resource Simulation

MOCC Mission Operations Control Center

NDDL New Domain Description Language

PANIC Package for Alarms and Notification of Incidents from Controls

TBC To Be confirmed

TBD To Be Defined

2.2.4 Overview

Section 1 deals with the choice of communication channels. Sections *2*, *3*, *4* and *5* explain implementation restrictions and considerations of the Planning, Operations, Configurations, Telemetry and Commands subsystems, respectively. *Section 6.* covers the validation and verification procedures. Finally, *Section 7.* deals with miscellaneous factors that need to be addressed or acknowledged when implementing the *MOCC*.

2.2.5 General considerations

As with any other part of the ERAS project, the guidelines in [3] must be followed. If the need for non-Python code arises, the coding style recommended for the language should be used instead of PEP8.

2.3 1. Communication channels

As explained in [3], the communication between the *MOCC*'s components is done via TANGO (see *[8]*). All communication should be done though the TANGO bus. In no scenario should a component communicate outside the bus, with the only exceptions being components that act as a proxy to external services (databases, hardware sensors, external data providers, etc.); in these cases the communication between the proxy and the external service should fulfill the external service's requirements.

In the case of databases, if the data store in the database can be accessed through a TANGO device server, direct access to the database should be avoided.

Unless otherwise specified, all components should be implemented as TANGO device servers.

2.4 2. Planning Subsystem

The framework chosen to implement planning related tasks is EUROPA (see [4]). Among other things, this means that templates, instantiations as well as macros should be created using EUROPA's *NDDL* and stored using EUROPA's Plan Database, and that the *AI* assistants and *UI* s should be developed using EUROPA's API.

The **Mission Asset and Resource Simulation** (MARS) (see *[9]*, Section 7.5) should be also taken into account, specially when developing EVA-related planning agents. In cases where EUROPA and MARS conflict, EUROPA takes precedence.

The components in this Subsystem should present an abstraction layer over EUROPA's and MARS's APIs. Direct access from other Subsystems to said APIs should be avoided.

2.5 3. Operations Subsystem

2.5.1 3.1. Execution Component

The Execution component of this subsystem should use EUROPA's API to execute the plans defined in the Planning Subsystem. However, no component from the Operations Subsystem should get the plans using EUROPA's API directly, but through the Planning Subsystem's components instead. This is done to maintain a clear separation between planning and execution/control, and to facilitate an eventual replacement of EUROPA, if necesary in the future.

2.5.2 3.2. Control Component

The Control component should be able to understand *NDDL*, since the plan step's expected outcomes will be defined in that language. This component should get the telemetry readings through the Telemetry Subsystem's Internal Delivery component(s), and not through accessing the telemetry storage nor the sensors directly. Since the underlying framework is TANGO, this should involve making a request to the TANGO device server in charge of managing the sensor of interest.

2.5.3 3.3. AI assistants

At the time of this writing, two types of *AI* assistants are foreseen:

- **Anomaly Detectors**: these assistants analyze telemetry from all sensors involved in a plan instance, in search for deviations that could compromise the plan. The method of analysis is sensor-dependant, but all anomaly detectors should use a uniform interface to provide their analysis. This means defining an appropriate "Alarm" data structure. The use of *PANIC* (see *[10]*) is advised whenever possible.
- **Corrective Measures Assistants**: these assistants help correct deviations in the plans. They should make use of the tools offered by EUROPA. No other requirements apply at the time.

2.5.4 3.5. User Interfaces

The *UI* s in the Operations Subsystem should be organized hierarchically in a tree, as shown in *Figure 1*. Each node in the tree should be a *UI* widget, which performs the following functions:

1. Gathers data from a data source.
2. Shows a user-defined representation of the data gathered.
3. Provides user-defined aggregations of the data gathered, which can function as a data source.
4. Shows a user-defined representation of the aggregations defined.

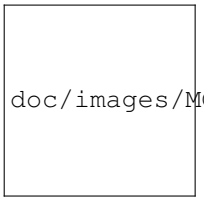


Fig. 2.1: Figure 1. The Operations UIs hierarchy

The aggregations are functions over one or all data sources queried by a node. So, if a *UI*-node **Z** gathers data from data sources *A*, *B* and *C*, possible aggregation functions could be, for instance:

- Maximum over the last 30 seconds of *A*.
- Average over the last hour of *B*.
- Instantaneous minimum of *A* and *C*.
- Average over the last 45 minutes of the cross-entropy of *B* and *C*.

In the tree, the leaves should gather data exclusively from telemetry sources and *AI* assistants, whereas inner nodes should gather data *preferentially* from lower tree nodes and *AI* assistants. The idea is to build abstraction layers to facilitate control and decision making by the Operation Subsystem's users by giving enough information while avoiding cluttering. This way, the users in charge of physical devices can see a detailed view of their devices, users in

charge of groups of devices can see the status of the whole group, the the person in charge of the whole mission can see the global status.

2.6 4. Configurations Subsystem

2.6.1 4.1. Devices

Based on the services provided by TANGO, the Devices component should extend TANGO's capabilities by offering an interface to find devices based on certain criteria, e.g.:

- Devices able to go outside.
- Devices able to load cargo.
- Devices able to measure radiation.
- Etc.

Beyond the information about a device already provided by TANGO (type and name of variables, signature of commands), in the case of devices that offer JSON-encoded variables and/or commands (see *[11]*), this component should also make available the schemas of said variables and commands.

```
NOTE: This last part could be implemented by requiring all ERAS' TANGO
devices to implement a commands that provides the schema information, as
suggested in [11].
```

2.6.2 4.2. Crew

Similarly to the Devices component, this component should offer information about the crew members physical well as mental characteristics, e.g.:

- Personal information: name, nationality, place of origin, etc.
- Biomedical statistics: age, weight, medical conditions, etc.
- Areas of expertise: mechanical engineering, geology, medicine, etc.
- Psychological characteristics: leadership, patience, stress tolerance, etc.

2.6.3 4.3. Macros

Combining TANGO and EUROPA, this component should offer a repository for macros. A user should be able to retrieve the list of macros that involve a certain device or crew member. A user should also be able to obtain the list of devices and crew members involved in a certain macro.

To facilitate the process of macro building, a *UI* should be developed.

2.7 5. Telemetry and Commands Subsystems

2.7.1 5.1. Telemetry Subsystem

The Telemetry Subsystem consists of three components – Acquisition, Storage and Delivery – in charge of reading telemetry from sensors, storing it in a database and making it available to whoever needs it. In particular, the delivery

can be in real-time, in the case when the Subsystem delivers telemetry as it is obtained; or historic, in the case when previously obtained and stored telemetry is needed.

There are two types of subcomponents in each the Acquisition and Delivery components. The first is the acquisition from and delivery to internal services, i.e., services that are part of ERAS, which occurs using the TANGO bus.

The second is the acquisition from and delivery to *external* services, i.e., databases, hardware, third-party services, etc., which occurs using whatever media the external services in question require. However, when the information from the external service is requested by another Subsystem, the request is processed through the TANGO bus, by developing a proxy (a TANGO device server) for the external service needed.

Finally, the Storage component should provide an interface for saving telemetry to a database, as well as retrieving it. Storage requests for this component should not be directly made from outside the Telemetry Subsystem. One possibility to achieve this is to implement it as a software library. However, retrieval request should be accepted from outside sources, though the TANGO bus.

2.7.2 5.2. Commands Subsystem

This Subsystem is very similar to the Telemetry Subsystem. The difference lies in principle in the information flow; whereas the Telemetry Subsystems mainly gathers information from external services (sensors) and delivers it to internal services (chiefly the Operations Subsystem), the Commands Subsystem gathers information only from the Operations Subsystem, and delivers it to internal as well as external services (actuators, crew members, etc.).

2.7.3 5.3. Implementation

Figure 2 shows a diagram of the internal structure of a TANGO device server that is able to gather and provide telemetry, as well as accepting and delivering commands. Any devices involved in operations should follow this structure.

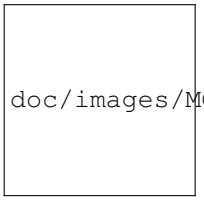


Fig. 2.2: Figure 2. Telemetry and Commands? Modules Implementation

All commands received should be both stored in the Commands Database and delivered to the appropriate external recipient (if needed).

Telemetry should be gathered at the instant the request is received, if the request is for instant telemetry, or retrieved from the Telemetry Database or TANGO attribute history buffer (whatever option is most appropriate).

2.8 6. Validation and Verification Procedures

TBD

2.9 7. Special Considerations

To avoid work duplication, a spreadsheet matching components to actual software modules should be written and linked to this document. The spreadsheet should also indicate whether a component is completely implemented, and if not, what is missing.

Mission Operations Control Center - Infrastructure Documentation

Author Mario Tambos

3.1 Change Record

2015.06.22 - Document created.

3.2 Introduction

3.2.1 How to use this document

3.2.2 Purpose

3.2.3 Reference Documents

- [1] ++ Mission Operations Control Center - High Level Design Documentation
- [2] ++ C3 Prototype document v.4
- [3] ++ Software Engineering Practices Guidelines for the ERAS Project

- [4] ++ Dachstein 2012 Mission Report
- [5] ++ Morocco MARS2013 Mission Report
- [6] ++ V-ERAS Project Description (2014 Release)
- [7] ++ V-ERAS-14 Mission Report

3.2.4 Glossary

AI Artificial Intelligence

ERAS European Mars Analog Station

EVA Extra-Vehicular Activity

GUI Graphic User Interface

IMS Italian Mars Society

MOCC Mission Operations Control Center

TBC To Be confirmed

TBD To Be Defined

3.2.5 Overview

3.3 1. The MOCC System and its Subsystems

3.3.1 High level architecture

3.3.2 Interfaces and Communication Channels

3.3.3 Subsystems

3.4 2. Documents

3.5 3. Management Structure

3.6 4. Infrastructure

3.7 5. Special Considerations

Mission Operations Control Center - Management Documentation

Author Mario Tambos

4.1 Change Record

2015.06.22 - Document created.

4.2 Introduction

4.2.1 How to use this document

4.2.2 Purpose

4.2.3 Reference Documents

- [1] ++ Mission Operations Control Center - High Level Design Documentation
- [2] ++ C3 Prototype document v.4

- [3] ++ Software Engineering Practices Guidelines for the ERAS Project
- [4] ++ Dachstein 2012 Mission Report
- [5] ++ Morocco MARS2013 Mission Report
- [6] ++ V-ERAS Project Description (2014 Release)
- [7] ++ V-ERAS-14 Mission Report

4.2.4 Glossary

AI Artificial Intelligence

ERAS European Mars Analog Station

EVA Extra-Vehicular Activity

GUI Graphic User Interface

IMS Italian Mars Society

MOCC Mission Operations Control Center

TBC To Be confirmed

TBD To Be Defined

4.2.5 Overview

4.3 1. The MOCC System and its Subsystems

4.3.1 High level architecture

4.3.2 Entities

4.3.3 Interfaces and Communication Channels

4.3.4 Subsystems

4.4 2. Documents

4.5 3. Management Structure

4.6 4. Infrastructure

4.7 5. Special Considerations

CHAPTER 5

Tango Servers

5.1 Body Tracker

5.1.1 Software Architecture Document for the Body Tracker application

Author Vito Gentile

Change Record

10th May, 2015 - Document created.

18th Jun, 2015 - Update GUI and added CLI.

Introduction

Purpose

The main goal of this module is to provide full body tracking of the astronauts, in order to interact with the ERAS virtual station. Using data provided by this module, the avatar of an astronaut can move inside the virtual ERAS station environment, by reproducing body movements of a real user inside a Motivity treadmill.

This module is based on the 3D skeleton tracking technique described at *this link*: <http://research.microsoft.com/apps/pubs/default.aspx?id=145347>. The Microsoft Kinect device generates a depth map in real time, where each pixel represents the distance between the Kinect sensor and the closest object in the scene at that pixel location. Based on this map, the Microsoft API implements the aforementioned skeletal tracking algorithm to accurately track different human body parts in 3D. This technique allows to estimate the position of 20 skeletal joints, which can be used on the Blender game engine to properly animate a 3D avatar model.

Note: This module supports and has been tested with the Xbox 360 version of Microsoft Kinect.

This module supports up to four Kinects simultaneously connected to a single machine.

An algorithm to estimate user steps using skeletal joint data is included in this module, in order to estimate and reproduce navigation paths that an astronaut defines by walking inside a Motivity treadmill.

Previous versions of the skeletal tracking module were implemented in C++ using OpenNI/NITE framework, and then C# using the Microsoft Kinect SDK. The new version of this module is based on PyKinect [2], which allows to use the Microsoft API with Python.

The following flow chart gives a pictorial view of the working steps of the body tracker application.

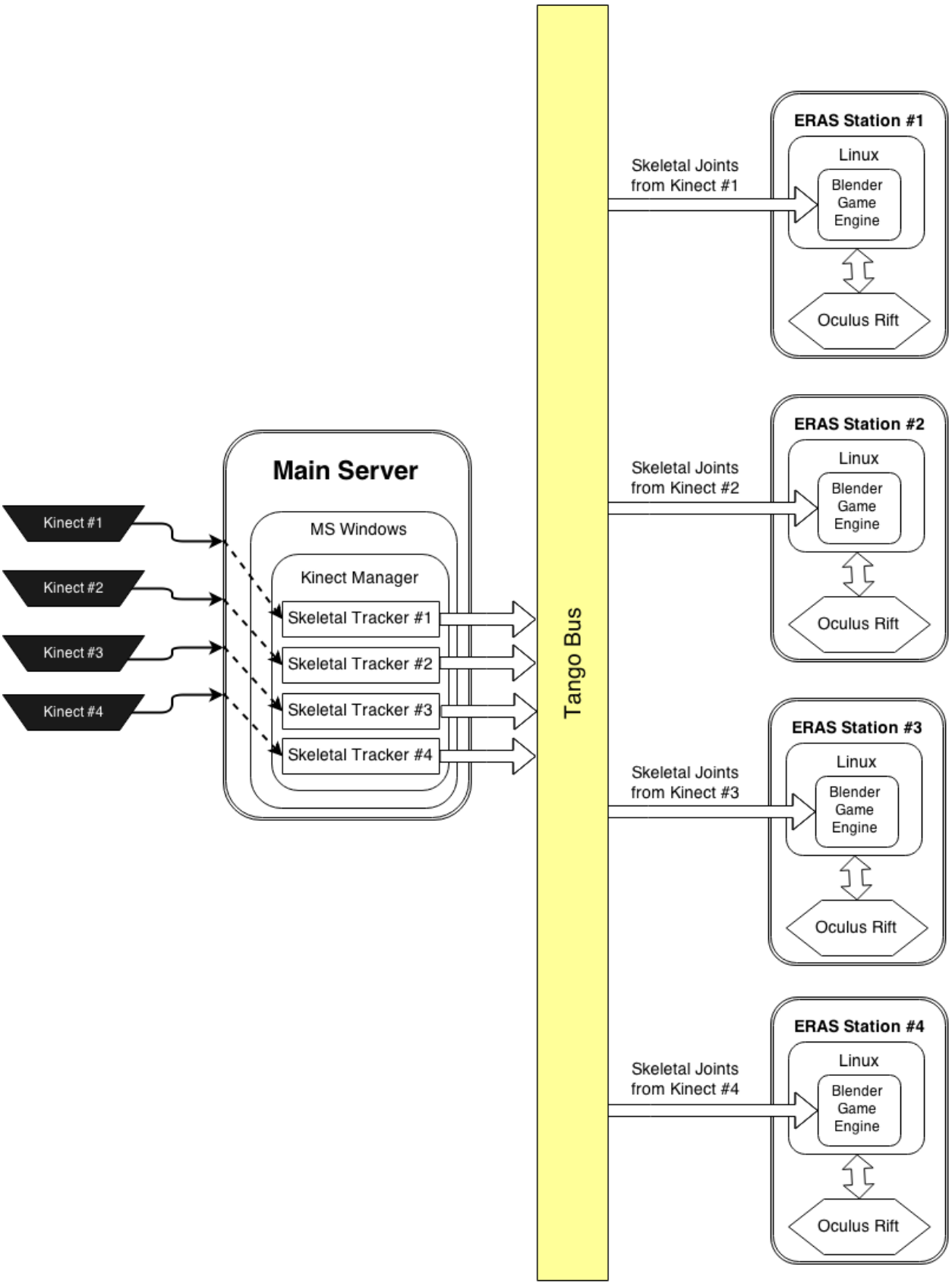


Fig. 5.1: Figure 1. System architecture. One main server with MS Windows is used to manage up to 4 Kinects. Data from these devices are sent to the Tango bus, that makes them available for any other ERAS software module.

Scope

TBC

Applicable Documents

TBD

Reference Documents

- [1] – *Real-Time Human Pose Recognition in Parts from a Single Depth Image*: <http://research.microsoft.com/apps/pubs/default.aspx?id=145347>
- [2] – *PyKinect*: <https://github.com/Microsoft/PTVS/wiki/PyKinect>
- [3] – *Kinect for Windows Sensor Components and Specifications*: <https://msdn.microsoft.com/en-us/library/jj131033.aspx?f=255&MSPPError=-2147217396>

Glossary

IMS Italian Mars Society

ERAS European MaRs Analogue Station for Advanced Technologies Integration

V-ERAS Virtual ERAS

VR Virtual Reality

TBD To be defined

TBC To be confirmed

GUI Graphical User Interface

For better understanding this document, a clear distinction between different kinds of users that interact with the system is needed.

Astronaut An user that interacts with the system using different devices and tools, such as Kinect, Oculus Rift, Motivity and so on, in order to explore the virtual Martian environment

Developer An user that interacts with the system in order to change its behavior. A developer can use programming languages to edit the source code of any ERAS software module

System manager An user that interacts with the system in order to install and configure it, or to assist and/or monitor an astronaut during his/her interactions with the system

Overview

TBD

Make an overview in which you describe the rest of this document and which chapter is primarily of interest for which reader.

Architectural Requirements

Non-functional requirements

Previous version of skeletal tracking module have been based on open source solutions. However, using the Microsoft API provided with the *Kinect SDK v1.8*: <https://www.microsoft.com/en-us/download/details.aspx?id=40278> has shown better performance, so it has been decided to use this software solution.

In order to exploit Microsoft API power, a server with Microsoft Windows 7 is needed. It means that a license for using this operating system is mandatory.

The development process is based on the use of Microsoft Visual Studio 2012 IDE. The Express edition can be used, and it is free (so there is no need for a license).

The application should be written in Python, using PyKinect for interfacing with Microsoft API. It requires CPython 2.7 installed.

Communication among this and other modules is based on the availability of a Tango bus.

Use Case View (functional requirements)

This module should track skeletal joints from an astronaut, and make these data available on the Tango bus.

An algorithm to estimate user step using skeletal joint data should be developed and included in this module, in order to estimate and reproduce navigation paths that an astronaut defines by walking inside a Motivity treadmill.

This module shouble be able to track hand gestures too.

An usable *GUI* should be provided, to allow system managers and maintainers to manage multiple Kinects.

Interface Requirements

This section describes how the software interfaces with other software products or users for input or output.

User Interfaces

GUI (Graphical User Interface) A graphical user interface is provided to system managers, in order to manage multiple Kinects connected to the ERAS system. This GUI can be executed under Windows on a desktop PC, on the same machine that manages skeletal data (the "main server" in Figure 1).

The interface is similar to the one shown in the following pictures:

Available Kinects are those with labels colored in black, while gray labels are used to visually identify the unactive (or unplugged) devices. When a Kinect is available, a system manager can decide to assign it to a Tango server, by selecting the radio button next to the server name.

When multiple Kinects are available, it is possible to figure out "which Kinect is which", in the sense that by observing depth images and comparing them with the scenes in front of each Kinects, you can mentally bind the letters used in the GUI to the physical device.

A sliding cursor is also available on the left of each images, to adjust the tilt angle.

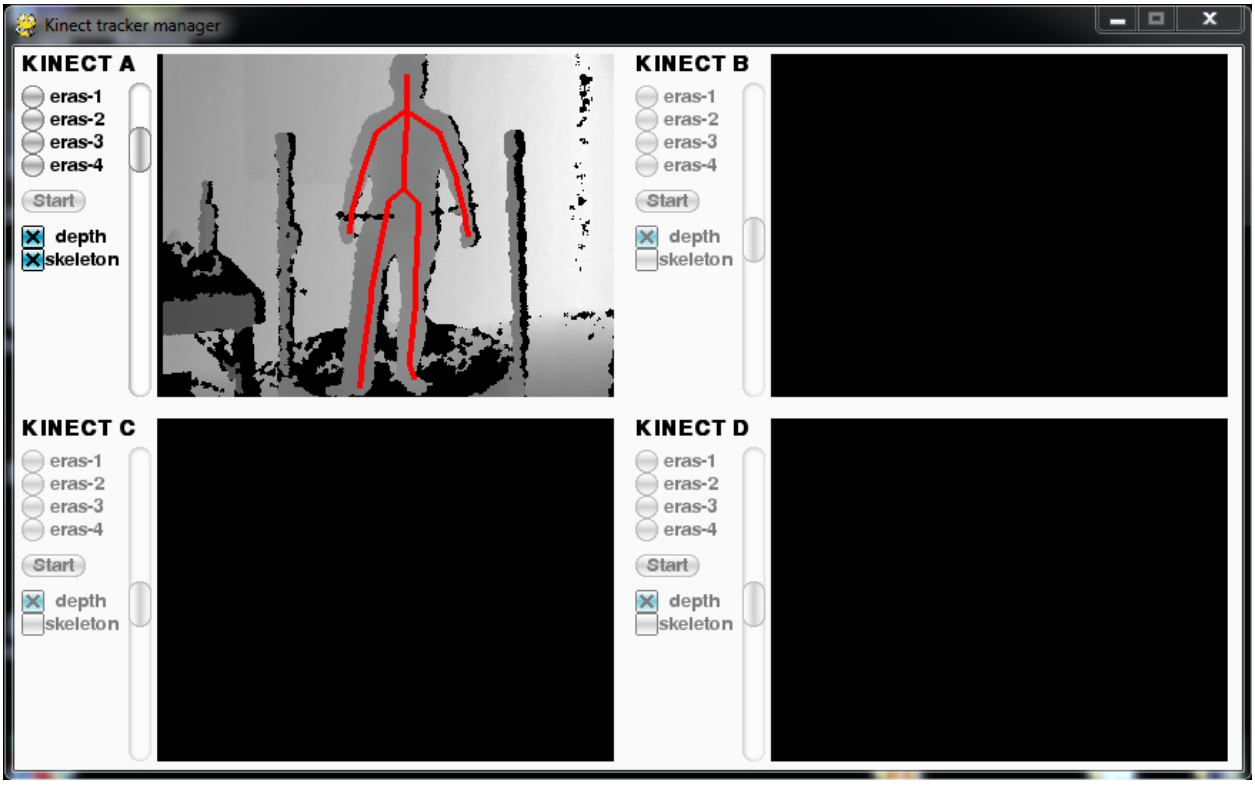


Fig. 5.2: Figure 2. GUI with a single Kinect available, and not yet connected to Tango

CLI (Command Line Interface) The GUI can be opened by executing:

```
python gui.py
```

This will allow a system manager to manage multiple Kinect from a single interface.

For testing purporses it is also possible to start the tracking process by using a single Kinect device. In this case, the command to execute is:

```
python tracker.py eras-X
```

where `eras-X` is the Tango device name (so X can be a value between and including 1 and 4).

The `tracker.py` script can also be used for simulation. The following command allows to record skeletal tracking data, and store them in a JSON file called `test.json`:

```
python tracker.py eras-X --log test.json
```

The outputted JSON file can be also used to simulate the tracking, without the need to use an actual device. To do this, just execute the following:

```
python tracker.py eras-X --sim test.json
```

To sum up how the `tracker.py` script works, here is the command line usage for it:

```
tracker.py {eras-1,eras-2,eras-3,eras-4} [-h] [--log FILENAME | --sim
FILENAME]
```

API (Application Programming Interface) TBD

Describes the application programming interface, if present. Foreach public interface function, the name, arguments, return values, examples of invocation, and interactions with other functions should be provided. If this package is a library, the functions that the library provides should be described here together with the parameters.

Hardware Interfaces

The system needs/supports the following hardware components: - Up to 4 Microsoft Kinect for Xbox 360 devices - A Kinect Power/USB Adapter for each Kinect device - A modern PC/Laptop with the following minimal hardware configuration:

- 32-bit (x86) or 64-bit (x64) processor
- Dual-core 2.66-GHz or faster processor
- Dedicated USB 2.0 bus for each Kinect
- 2 GB RAM or more

Software Interfaces

TBC

A high level description (from a software point of view) of the software interface if one exists. This section can refer to an ICD (Interface Control Document) that will contain the detail description of this interface.

Communication Interfaces

The skeletal joints and other data tracked by this module are sent to a Tango bus, so the machine that manages all the Kinects must include these capabilities.

Every other module can read skeletal data from the Tango bus. For instance, the Blender Game Engine can use position of skeletal joints to update the pose of a 3D astronaut model. In addition to this, walking speed and and body orientation are provided by this module via the Tango bus, to be used for user/rover navigation in Blender.

Performance Requirements

The system must track astronaut's skeletal joints in real-time. This allows the user to synchronize its body movements and gestures to what he/she sees and feels.

Logical View

TBD

Describe the architecturally significant logical structure of the system. Think of decomposition in terms of layers and subsystems. Also describe the way in which, in view of the decomposition, Use Cases are technically translated into Use Case Realizations

Layers

TBD

The ERAS software applicationg belong to the heterogeneous Distributed Control System (DCS) domain which can be represented as a layered architecture. This is a very common design pattern used when developing systems that consist

of many components across multiple levels of abstraction as in ERAS case. Normally, you should be developing components that belong to the Application layer

Subsystems

TBD

Describe the decomposition of the system in subsystems and show their relation.

Use Case Realizations

TBD

Give examples of the way in which the Use Case Specifications are technically translated into Use Case Realizations, for example, by providing a sequence-diagram.

Implementation View

TBD

This section describes the technical implementation of the logical view.

Deployment View

TBD

Describe the physical network and hardware configurations on which the software will be deployed. This includes at least the various physical nodes (computers, CPUs), the interaction between (sub)systems and the connections between these nodes (bus, LAN, point-to-point, messaging, etc.). Use a deployment diagram.

Development and Test Factors

Hardware Limitations

- Depth camera included in Microsoft Kinect works at no more than 30 frame per second. This limits the speed of an astronaut's movements: too fast gestures can result in tracking and/or recognition errors
- Microsoft Kinect may not work well outdoor, due to the IR-based technology used by this device: sunlight can interfere with IR rays used by Kinect, and invalidate depth and skeletal data
- Fields of view (see [3]) of multiple Kinect should never intersect, because this can invalidate depth and skeletal data

Software validation and verification

TBD

Give a detail requirements plan for the how the software will be tested and verified.

Planning

The development of this module is divided in the following phases:

- Implementation of a Python tracker based on PyKinect, which can track skeletal joints and send them on the Tango bus
- Implementation of a GUI for system managers, to support simultaneous use of multiple Kinects
- Definition and implementation of an algorithm to estimate user's step using skeletal joint data, in order to reproduce navigation paths defined by any astronaut walking inside a Motivity treadmill, or using Motigravity
- Integration of touchless gesture recognition [TBD]

Appendix A: Use Case template

TBD

Use Cases drive the whole software process and bind together all the phases from requirements capture to final delivery of the system and maintenance. They are a very effective way of communicating with customers and among team members. Before every discussion always provide the partners with a set of relevant Use Cases.

During meetings, they stimulate focused discussions and help identifying important details. It is important to keep in mind that Use Cases have to describe WHAT the system has to do in response to certain external stimuli and NOT HOW it will do it. The HOW is part of the architecture and of the design.

What follows is the empty template:

Use Case: <Name>

<Short description>

Actors

<List of Actors>

Priority

<Low, Normal, Critical>

Preconditions

<List of preconditions that must be fulfilled>

Basic Course

<Step-by-step description of the basic course>

Alternate Course

<Step-by-step description of the alternate course>

Exception Course

<Step-by-step description of the exception course>

Postconditions

<List of postconditions (if apply)>

Notes

5.1.2 Instructions for setting up a MS Windows machine for running body tracking

Author Vito Gentile

Change Record

13th Dec, 2014 - Document created.

3rd May, 2015 - Improved formatting (minor fix).

4th Jun, 2015 - Updated setup instructions for Python, Tango and PyKinect.

18th Jun, 2015 - Fix typos.

2nd Aug, 2015 - Added setup instructions for VPython.

Version of Microsoft Windows

The following instructions are based and were tested on Microsoft Windows 7 64-bit. After having installed the operating system, setup all drivers needed to use the machine in the right way.

Before starting with all mandatory software to run body tracking, may be useful to install all the following software:

- **Recommended:**
  - A modern browser (e.g. Firefox or Chrome)
  - 7zip
  - Geany (or any other programming-oriented text editor)
  - Daemon Tools Lite (useful for mounting .iso images, during some installation phases)
  - VirtualBox
  - TortoiseHG
- **Suggested:**
  - Adobe Flash Player
  - Adobe Reader (or another PDF reader)

Installing Python

To support PyKinect, you must install *Python 32-bit 2.7*. To install this version of Python, use this link: https://www.python.org/ftp/python/2.7/python-2.7.msi

It is recommended to install this version of Python in `C:\Python27_32bit\`.

To be able to excute Python from a command line, you must add the installation folder path to the `Path` variabe in Windows. You will also need to set the `PYTHONPATH` variable. To this end do the following:

- Right-click *My Computer* and select *Properties*
- Click the *Advanced System Settings* link in the left column
- The System Properties window will open. Here click on the *Advanced* tab, then click the *Environment Variables* button near the bottom of that tab
- In the *Environment Variables* window, highlight the `Path` variable in the *System variables* section and click the *Edit* button
- Append the following at the end of the value: `C:\Python27_32bit\`, then click on *Ok*
- In the *Environment Variables* window, check if there is a `PYTHONPATH` variable. If yes, highlight it and change its value in `C:\Python27_32bit\`; otherwise click on the *New* button, and create this environment variable

Installing Visual Studio and Kinect SDK

In order to use Kinect on Windows, you have to install Microsoft Visual Studio. Tests were done using *Visual Studio 2013*.

The installation process will be quite long, and it will probably require some reboots. After that, you have to install (in this order):

- *Kinect SDK 1.8*: <https://www.microsoft.com/en-us/download/details.aspx?id=40278>
- *Kinect Developer Kit 1.8*: <https://www.microsoft.com/en-us/download/details.aspx?id=40276>

Finally, simply plug-in the Kinect and let Windows Update to install its drivers.

Installing PTVS and PyKinect

Python Tools for Visual Studio (PTVS) "is a free, open source plugin that turns Visual Studio into a Python IDE". It can be useful if you want to develop in Python with Visual Studio, and it also provides some facilities for Kinect developers.

Installing and use PTVS on Visual Studio

In order to install PTVS, go to http://pytools.codeplex.com/releases and download the most recent version of PTVS that fits with your Visual Studio version. Tests were done using Visual Studio 2013 and *PTVS 2.1*, and the following documentation referes to these versions.

After having installed PTVS, open Visual Studio, and go to *File -> New -> Project*. Then, under *Template -> Python -> Samples*, select *PyGame using PyKinect*. This will create a new Python project, with a Python script structured to be used with PyGame and PyKinect.

Project configuration

In Visual Studio 2013, go to *Tools -> Options*. Then, under *Python Tools -> Environment Options*, select the *Python 32-bit 2.7* environment. If it is not available, select *Add Environment*, name it *Python 32-bit 2.7* and add the following fields:

- *Path*: `C:\Python27_32bit\python.exe`
- *Windows Path*: `C:\Python27_32bit\pythonw.exe`
- *Library Path*: `C:\Python27_32bit\lib`
- *Architecture*: `x86`
- *Language Version*: `2.7`
- *Path Environment Variable*: `PYTHONPATH`

Now open the Solution Explorer under the project name, right click on *Python Environments* and select *Add/Remove Python Environments...*. Then make sure that only the *Python 32-bit 2.7* environment is checked.

Installing PyGame

Go to http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame for downloading and installing PyGame for Python 32-bit 2.7. You can do it with `pip`, but also by following the instructions shown in Visual Studio after project creation. These instruction are summarized as follows, and can be generally used for installing any additional Python package.

- In the Solution Explorer, right click on *Python 32-bit 2.7* (under *Python Environments*) and then select *Install Python Packages...*
- **If you want to install a Python package without explicitly download it:**
  - Select *pip*, type the package name and then select *OK*
- If you want to install a downloaded .whl package (e.g. obtained from http://www.lfd.uci.edu/~gohlke/pythonlibs/):
  - Make sure to have the package `wheel` installed. If not, install it as described above
  - Select *pip*, type the full path to the file (wrapped by double quotes) and then select *OK*

Using the above instructions you will be able to install PyGame, by typing the double-quoted full path of the PyGame package downloaded from http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame. Make sure to select the last 32-bit version for Pythion 2.7 (the file name should be something like `pygameX.X.XXXcp27nonewin32.whl`.

Tests were done with PyGame 1.9.2a0 32-bit for Python 2.7.

Installing PyKinect

By following the above instructions for installing a Python package from Visual Studio, or simply using `pip` on a command line terminal, install the package `pykinect`.

Installing additional Python packages

Before continuing, you need also to install the following Python packages:

- *numpy*: required to install PyTango; it can be installed with `pip` or using the above instructions for installing a Python package from Visual Studio
- *PyTango*: download the last 32-bit version for Python 2.7, available from https://pypi.python.org/pypi/PyTango/

- *pgu*: download from https://code.google.com/p/pgu/ and install it with `pip` (follow the above instructions, as if the package you download is a .whl file)
- *VPython*: download the automatic installer from http://vpython.org/contents/download_windows.html (chose the Win-32 version, not the Win-64 one!)

Note: As a source to fine a lot of Python libraries, packed as Windows installers or as .whl files, you can refer to http://www.lfd.uci.edu/~gohlke/pythonlibs/

Installing Tango

Go to http://www.tango-controls.org/downloads/source/ and select the binary distribution for Windows 64 bits. Download and install it.

After the installation, you will be able to access to a lot of utility and tools to get information about Tango and the device servers (e.g. *Jive*). To use them, you must install *Java for Windows*; you can get it from https://java.com/download/

Configure Tango Host

To be able to get all Tango informations, you need to specify the address of the Tango host. Assuming that it is 198.168.1.100:10000, open the command line and type:

```
set TANGO_HOST=192.168.1.100:10000
```

Using a virtual machine manager

Installing a virtual machine manager like *VirtualBox* can be very useful in order to install Ubuntu or another Linux distribution on the same Windows machine.

If you want to do this, you are probably interested in setting up a shared folder between host and guest operating systems. To do this in VirtualBox, see: http://my-wd-local.wikidot.com/otherapp:configure-virtualbox-shared-folders-in-a-windows-ho

5.1.3 Testing Tango integration on a Windows host

Author Vito Gentile

Change Record

3rd May, 2015 - Document created.

18rd Jun, 2015 - Fix typos.

Purpose

This document explains the procedure to test Tango-based software on Windows. The basic idea is to provide clear instructions to setup all the necessary components to test, on a single machine, if data sent on Tango bus can be read from other clients.

In order to test Tango integration, VirtualBox needs to be installed on the Windows machine. This allows to create a virtual machine with an installation of Ubuntu. In this way, a Tango server can be installed in the virtual machine,

and will be possible to emulate Tango communications between Windows and other clients, by testing if data from Windows can be read also on Ubuntu.

Reference Documents

- [1] – *Instructions for setting up a MS Windows machine for running body tracking*: <https://eras.readthedocs.org/en/latest/servers/body_tracker_ms/doc/setup.html>

Version of Microsoft Windows

The following instructions are based and were tested on Microsoft Windows 7 64-bit. After having installed the operating system, setup all drivers needed to use the machine in the right way.

Furthermore, it is assumed that [1] has been read and used before of all.

Version of Ubuntu

The following instruction are based and were tested on Ubuntu 14.10 and Ubuntu MATE 14.10.

If you would like to use any other Linux distribution, consider the possibility of unexpected installation errors.

Setup of an Ubuntu virtual machine on VirtualBox

Setting up a virtual machine on VirtualBox is quite easy, and doesn't need further explanation.

After the Ubuntu installation, install the Guest Additions as follows:

- Start the Ubuntu virtual machine
- Click on the *Devices* menu and choose *Install Guest Additions...*
- This mounts the VBox Guest Additions ISO image. Run this as root:

```
# ./media/cdrom0/VBoxLinuxAdditions.run
```

- Shut down the virtual machine.

Then, after shutting down the virtual machine, go to *Settings* -> *Network* and select *Bridged networking*. This allows to assign an IP to the virtual machine (Ubuntu), that will be different from the IP of the host system (Windows).

Set a static local IP address for the virtual machine

Setting up a static IP address to the virtual machine can facilitate the communication between Windows client and the Tango server (that will be installed on Ubuntu).

To do this with Ubuntu in a semi-graphical way (preferred, due to automatic management of the Network Manager plugin), see: http://www.sudo-juice.com/how-to-a-set-static-ip-in-ubuntu/

It is also possible to configure a static IP manually. For more details, see: https://www.howtoforge.com/debian-static-ip-address

Installation of Tango on Ubuntu (guest)

The next step is to install Tango on the Ubuntu virtual machine. Please refer to the *Tango setup <https://eras.readthedocs.org/en/latest/doc/setup.html>* page of this documentation.

Set environment variable on Windows

In the *Instructions for setting up a MS Windows machine for running body tracking <https://eras.readthedocs.org/en/latest/servers/body_tracker_ms/doc/setup.html>*, you were asked to set up an address for the Tango server. What we will do here is to perform our tests with a simple Tango server installed in the virtual machine.

Assuming the the IP address of the virtual machine is 192.168.0.111, open a terminal on Windows and execute the following command:

```
set TANGO_HOST=192.168.0.111:10000
```

Perform tests

Now everything is ready for testing. You can publish your data on the Tango bus (from Windows), and then use **jive** from the virtual machine to see if your data have been correctly published.

5.2 ERAS Virtual Reality

5.2.1 Software Architecture Document for the ERAS VR Simulation

Author Alexander Demets

Change Record

23th May, 2014 - Document created.

Introduction

Purpose

Implementation of an interactive VR simulation of the Eras Mars Station.

A user will be represented by a virtual avatar, he fully controlls and with which he can interact with the environment. Using an Oculus Rift device, the user will get an immersive represantation of the ERAS Mars station, in which he can walk, look around and interact with.

Blender will be used for the development and its Blender Game Engine for the real time simulation.

Goal of the project is the implementation of full Oculus Rift support for the Blender Game Engine using the official Oculus SDK and a simulation in the BGE to demonstrate the the ERAS station.

Scope

Describes the scope of this requirements specification.

Applicable Documents

- [1] – Oculus Developer Platform
- [2] – Oculus SDK Overview v0.3.2
- [3] – Blender Python API

Reference Documents

- [1] – C3 Prototype document v.4
- [2] – Software Engineering Practices Guidelines for the ERAS Project
- [3] – ERAS 2014 GSoC Strategic Plan

Glossary

Overview

Make an overview in which you describe the rest of this document the and which chapter is primarily of interest for which reader.

Architectural Requirements

This section describes the requirements which are important for developing the software architecture.

Non-functional requirements

Hardware:

1. Fast GPU: For VR a steady framerate of at least 60 frames/second is needed for acceptable simulation
2. Oculus Rift device: a virtual reality headset is needed for proper headtracking and full immersion

Software:

1. A working installation of Blender for development
2. Ubuntu >13.10

Use Case View (functional requirements)

1. First person character controller with adjustable height and various input methods (keyboard, joystick, external hardware)
2. **Oculus SDK integration into Blender for:**
   (a) Sensor Fusion head-tracking
   (b) Barrel distortion rendering
3. Integration with hand tracking project

Interface Requirements

This section describes how the software interfaces with other software products or users for input or output. Examples of such interfaces include library routines, token streams, shared memory, data streams, and so forth.

User Interfaces

Describes how this product interfaces with the user.

GUI (Graphical User Interface) Describes the graphical user interface if present. This section should include a set of screen dumps or mockups to illustrate user interface features. If the system is menu-driven, a description of all menus and their components should be provided.

CLI (Command Line Interface) Describes the command-line interface if present. For each command, a description of all arguments and example values and invocations should be provided.

API (Application Programming Interface) Describes the application programming interface, if present. Foreach public interface function, the name, arguments, return values, examples of invocation, and interactions with other functions should be provided. If this package is a library, the functions that the library provides should be described here together with the parameters.

Hardware Interfaces

Oculus Rift VR headset, for documentation see [1] & [2]

Software Interfaces

A high level description (from a software point of view) of the software interface if one exists. This section can refer to an ICD (Interface Control Document) that will contain the detail description of this interface.

Communication Interfaces

Describe any communication interfaces that will be required.

Performance Requirements

Specifies speed and memory requirements.

Logical View

Describe the architecturally significant logical structure of the system. Think of decomposition in terms of layers and subsystems. Also describe the way in which, in view of the decomposition, Use Cases are technically translated into Use Case Realizations

Layers

The ERAS software applicationg belong to the heterogeneous Distributed Control System (DCS) domain which can be represented as a layered architecture. This is a very common design pattern used when developing systems that consist of many components across multiple levels of abstraction as in ERAS case. Normally, you should be developing components that belong to the Application layer

Subsystems

Describe the decomposition of the system in subsystems and show their relation.

Use Case Realizations

Give examples of the way in which the Use Case Specifications are technically translated into Use Case Realizations, for example, by providing a sequence-diagram.

Implementation View

This section describes the technical implementation of the logical view.

Deployment View

Describe the physical network and hardware configurations on which the software will be deployed. This includes at least the various physical nodes (computers, CPUs), the interaction between (sub)systems and the connections between these nodes (bus, LAN, point-to-point, messaging, etc.). Use a deployment diagram.

Development and Test Factors

Hardware Limitations

Describe any hardware limitations if any exist.

Software validation and verification

Give a detail requirements plan for the how the software will be tested and verified.

Planning

Describe the planning of the whole process mentioning major milestones and deliverables at these milestones.

Notes

Appendix A: Use Case template

Use Cases drive the whole software process and bind together all the phases from requirements capture to final delivery of the system and maintenance. They are a very effective way of communicating with customers and among team members. Before every discussion always provide the partners with a set of relevant Use Cases.

During meetings, they stimulate focused discussions and help identifying important details. It is important to keep in mind that Use Cases have to describe WHAT the system has to do in response to certain external stimuli and NOT HOW it will do it. The HOW is part of the architecture and of the design.

What follows is the empty template:

Use Case: <Name>

<Short description>

Actors

<List of Actors>

Priority

<Low, Normal, Critical>

Preconditions

<List of preconditions that must be fulfilled>

Basic Course

<Step-by-step description of the basic course>

Alternate Course

<Step-by-step description of the alternate course>

Exception Course

<Step-by-step description of the exception course>

Postconditions

<List of postconditions (if apply)>

Notes

5.2.2 Setup guide for pyOVR and the Blender integration

Authors Alexander Demets (2014), Siddhant Shrivastava (2015)

Workflow(updated August 2015)

- Download `version 0.5.0.1` of the Oculus SDK. The SDK can be found here , and the signature can be found here.
- Download Bindings for the Python Language from here
- Setup SDK after following the instructions in the `README.Linux` in the root directory of the SDK.
- Plugin the Oculus device (DK1 or DK2). Establish the connections appropriately.
- Start the Oculus VR daemon by running `ovrd` in the Linux terminal. `ovrd` must run throughout when Oculus is to be used.
- Configure and Identify Oculus by running `RiftConfigUtil` in another terminal.
- Paste the `oculusvr` folder from the Python Bindings `python-ovrsdk` folder in `eras/servers/erasvr/`.
- Use the bindings in Blender Game Engine.

Possible Issues and Troubleshooting (updated August 2015)

Library Dependency Related Errors

If you get an error of this kind -

```
ImportError: <root>/oculusvr/linux-x86-64/libOculusVR.so: undefined symbol:
glXMakeCurrent
```

This means that the shared object file in the preconfigured bindings dosen't identify the existing library symbolic links set up by the Linux kernel. Here's what you should do in this case-

- List all the dependencies of the shared object file `oculusvr.so`. You might get an output similar to the one shown in this post
- Trick the kernel into **preloading** the appropriate libraries into the memory before linking the shared object file dependencies at runtime. This is done by placing `LD_PRELOAD="/usr/lib/x86_64-linux-gnu/libGL.so"` in the `.bashrc` file.
- This must be followed by reloading `bash` by running `source bash`
- This should solve the errors.
- Keep checking the output of `ldd -a oculusvr.so` and repeat the above steps until all the dependencies are preloaded.

Setup guide for pyOVR

Generate libovr.so

Currently the default makescript for the Oculus SDK compiles the whole LibOVR package as a static library. For the *ctypes*-integration of LibOVR's C-API in Python, we need it as a dynamic library (shared object). To do this, the original makefile has to be replaced with a custom one.

First download and unzip the OculusSDK v0.3.2 (currently newest available for Linux) from the Oculus Developer Platform. To install all dependencies for compilation of the Oculus SDK, run following commands in the terminal:

```
tar xvzf ovr_sdk_linux_0.3.2.tar.gz
cd OculusSDK
./sh ConfigurePermissionsAndPackages.sh
```

And copy the custom makefile from your cloned *ERAS* repository over to the **LibOVR** folder, and compile with *make*:

```
cp -b path/to/eras/servers/erasvr/pyOVR/Makefile_for_LibOVR/Makefile LibOVR/Makefile
cd LibOVR
make
```

Note: Carefull! Copy the new Makefile into the *LibOVR* folder, **not** the *OculusSDK* folder. The *-b* argument for **cp** makes a backup copy for the original Makefile.

Configure pyOVR

At this point, there should be a *libovr.so* in the according platform folder of *LibOVR/Lib*. Of course the Python bindings have to know the location of the *libovr.so* file. So switch over to the *pyOVR* folder, and do **one** of those things:

1st option (Copy *libovr.so* into *pyOVR* folder):

```
cp OculusSDK/LibOVR/Lib/.../libovr.so eras/servers/erasvr/pyOVR/libovr.so
```

2nd option (Edit path directly in *pyOVR/pyOVR.py*):

```
LIBOVR_PATH = path/to/OculusSDK/LibOVR/Lib/.../libovr.so
```

Run Tests for pyOVR

Now that everything is setup, you can run the test scripts:

```
python3 pyOVR/Testsimple.py
python3 pyOVR/TestRiftDevice.py
```

If successfull, you will see the Oculus Rift identified, and its sensor data put out continuously:

```
user@ubuntu:~/Development/eras/servers/erasvr/pyOVR$ python3 TestSimple.py
OVR::DeviceManagerThread - running (ThreadId=0x7f950163d700).
OVR::DeviceManager - initialized.
*** SensorFusion Startup: TimeSeconds = 1408388773.073210
OVR::Linux::HIDDevice - Opened '/dev/hidraw1'
Manufacturer:'Oculus VR, Inc.'  Product:'Tracker DK' Serial#:'AAAAAAAAAAAA'
OVR::SensorDevice - Closed '/dev/hidraw1'
OVR::Linux::HIDDevice - HID Device Closed '/dev/hidraw1'
OVR::Linux::HIDDevice - HIDShutdown '/dev/hidraw1'
b'Oculus Rift DK1'
```

```
OVR::Linux::HIDDevice - Opened '/dev/hidraw1'
Manufacturer:'Oculus VR, Inc.'  Product:'Tracker DK'  Serial#:'AAAAAAAAAAAA'
Sensor created.
+1.00 +0.00 +0.00 +0.00
+1.00 +0.08 -0.00 +0.00
+1.00 +0.08 -0.00 +0.00
+1.00 +0.08 -0.00 +0.00
+1.00 +0.08 -0.00 +0.00
+1.00 +0.08 -0.00 +0.00
+1.00 +0.08 -0.00 +0.00
+1.00 +0.08 -0.00 +0.00
+1.00 +0.08 -0.00 +0.00
```

Blender Integration

Before attempint the Blender integration you have to make a simbolic link to the directory containing pyOVR from the directory containing the character controller blender file. In the current status of our software archivse this should be something like this: cd <archives root>/v-eras-blender/scenes ln -s ../../eras/servers/erasvr/pyOVR pyOVR

The Blender integration is practically a *drop-in* **character controller** for your BGE project. To test it out open the included *CharacterControllerOVR.blend* file in Blender v2.71+.

Overview of character controller

First of, that's how the object hierachy ob the character controller looks:

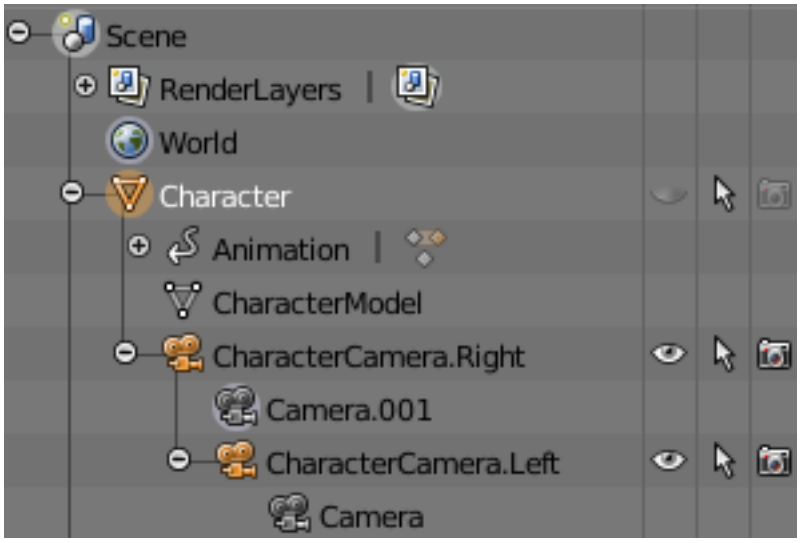


Fig. 5.3: Figure 1: Object Hierarchy for custom character controller.

The *Character* object is our rigidbody, and also defines the outer collision bounds. Its geometry is defined as invisible cylinder, and it's collision bounds are defined by a capsule (makes movement more fluid).

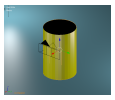
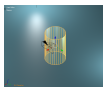
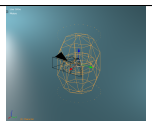
		

Fig. 2: *Character* object, solid outer hulls	Fig. 3: Wireframe of *Character* object	Fig. 3: Actual collision data used for *Character* object

The movement happens, by checking the sensor data (currently Keyboard & Mouse), and applying a force to the rigidbody *Character*. The scripts and actuators are setup like that:

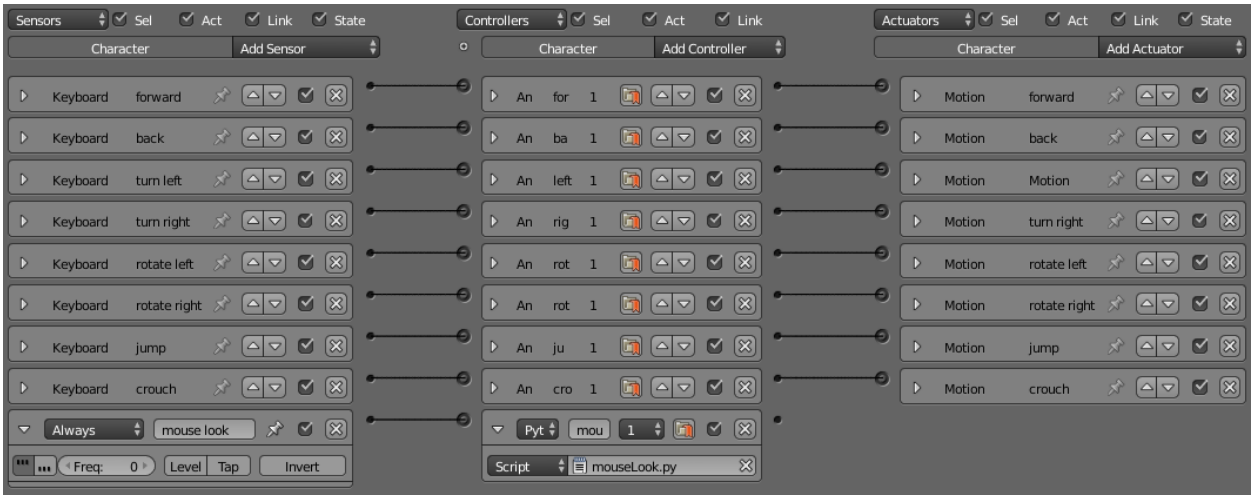


Fig. 5.4: Figure 5: Sensor/Actuator setup for Blender character controller

Looking at the setup the movement is mostly configured through Blender inbuild functions, only exception is the Mouse-look script. What it does is, it takes the relative movement of the Mouse its X-axis and *rotates* the whole *Character* rigidbody. This means **including** the two *Camera*-childs, which get rotated left/right like a fixed head to a body would do. The Oculus sensor data is then applied to the *Camera.right* and *Camera.left* object, by using following setup:

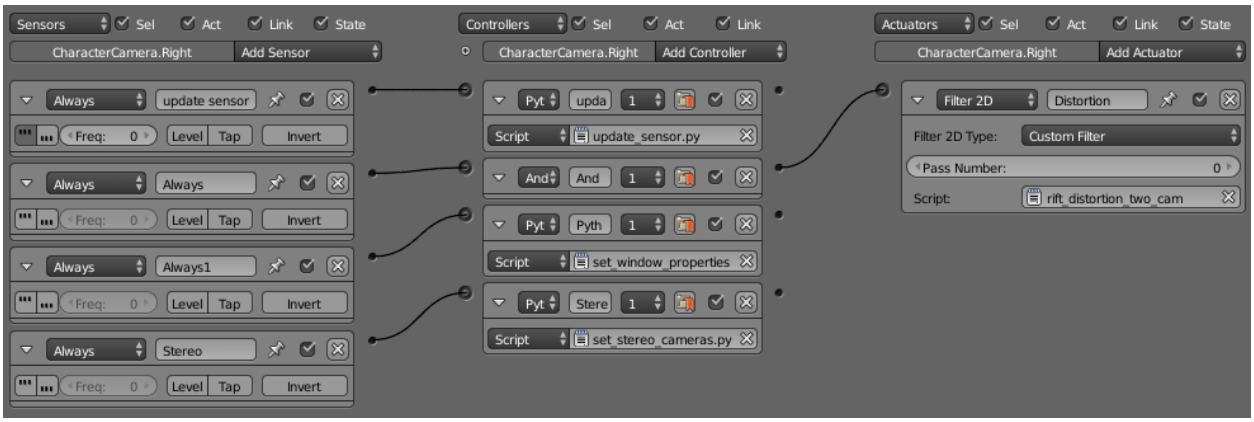


Fig. 5.5: Figure 6: Sensor/Actuator setup for head rotation via sensor data and Barrel rendering.

The *update_sensor.py* script initializes the Rift and applies the sensor data to the camera objects. The *Filter2D* post-process applies the barrel distortion to the resulting image.

Integration in own Blender scene

Integration of the character controller into a custom scene is very easy, just open your scene and go to:

```
File > Link/Append > CharacterController.blend > choose "Character"
```

This will either link or append the character controller into your Blender scene.

Render settings in Blender

These settings give good results, but can be modified as needed:

Applicable Documents

- [1] – Oculus Developer Platform
- [2] – Oculus SDK Overview v0.3.2
- [3] – Blender Python API

5.3 ROS/Gazebo

5.3.1 Software Architecture Document for ROS, Gazebo integration with Tango Controls

Author Kunal Tyagi

Change Record

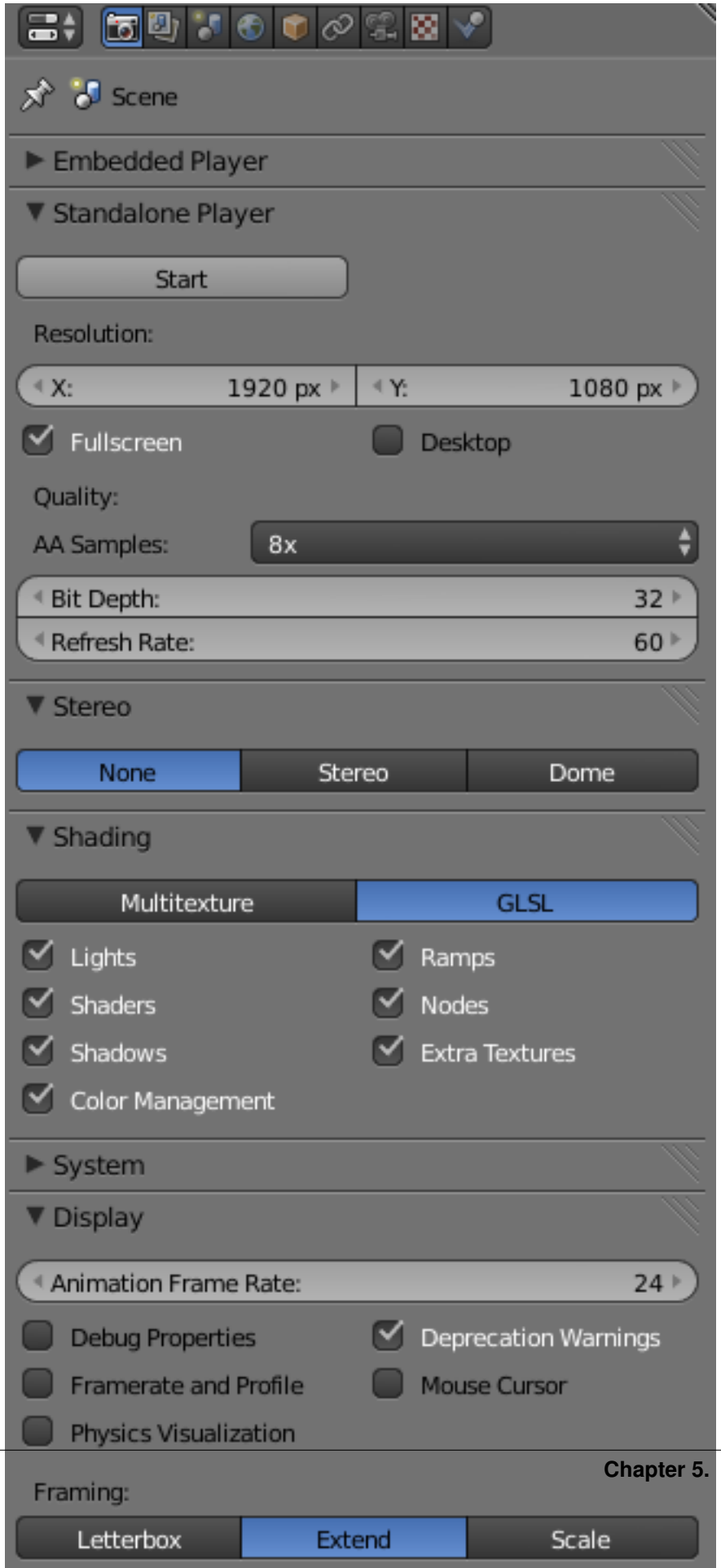
Introduction

Purpose

This document is intended to detail for developers and Users of the ERAS Tango Controls and other ERAS C3 components what changes the control system would undergo during integration with ROS and Gazebo, and what new features it has (or will have) as well as known issues and work left.

Scope

This requirements specification is intended to cover a software library and associated documentation.


Scene
Embedded Player
Standalone Player
Start
Resolution:
X: 1920 px
Y: 1080 px
Fullscreen
Desktop
Quality:
AA Samples: 8x
Bit Depth: 32
Refresh Rate: 60
Stereo
None
Stereo
Dome
Shading
Multitexture
GLSL
Lights
Ramps
Shaders
Nodes
Shadows
Extra Textures
Color Management
System
Display
Animation Frame Rate: 24
Debug Properties
Deprecation Warnings
Framerate and Profile
Mouse Cursor
Physics Visualization
Framing:
Letterbox
Extend
Scale

Applicable Documents

Reference Documents

Glossary

`TANGO` TANGO Control, an Object Oriented Distributed Control System using CORBA and Zeromq, used as primary Control System by ERAS

`ROS` Robot Operating System, a set of software libraries and tools very useful in building and controlling devices, especially Robots

`Gazebo` A robust physics engine with high-quality graphics, useful for more realistic simulations to test the behaviour of robot in different settings

Overview

This package will provide high-level access via TANGO to simulate Trevor in Gazebo using C++ plugins with Python and TANGO bindings. It will also contain files required to correctly use the simulator. The package may be expanded to include the files required for ROS integration also.

This document is divided into several parts.

1. For a typical **User**, the sections Interface Requirements, and Performance Requirements are of primary interest
2. For a **Beginner**, the section Logical View covers most of the information required to get started on their contributions.
3. For a **Developer**, Architectural Requirements and Implementation View are of high importance
4. For **Maintainers**, Deployment View and Development & Test Factors are a must-read apart from the aforementioned sections

PS: Start from 1 and make your way down towards any higher number

Architectural Requirements

Working knowledge of ROS, TANGO, Gazebo, as well as expertise in Build System, and Makefiles is a must

Non-functional Requirements

- ROS is independantly developed, and has an evolving build system. As a result, several features need to be modified with a new release of ROS, though most of the code is expected to work fine with only regular updates to API required to hanfle the upgrading process
- Gazebo is independantly developed, and has undergone several changes in its API, and expected to go many more. It is mostly developed by the same community as ROS, so changes will be uniformly spread over these 2 softwares. Also, Python API is in developement, so currently, only C++ API would be used here
- Security: ROS uses no authentication methods, so, TANGO developed in collboration with ESRF, would have to accomodate for this

Functional requirements *(use case view)*

Include things like

- availability of ROS-agnoistic packages
- overloading of some functions of ROS to publish/receive messages to enable the packages unknowingly using ROS format to achieve the same through TANGO device servers
- availability of messages to move the simulated Trevor

Interface Requirements

User Interfaces

The user can(or rather would be able to) use ROS libraries with TANGO just as without TANGO, except with a few changes.

Similar usage with Gazebo is expected. It would likely be able to be used just like EUROPA, a standalone plugin for existing software stack.

It would require modifications to be used without the sotware stack, but its presence would not affect the overall functionality of the software stack

GUI

No seperate GUI is provided except from the existing ones by ROS and Gazebo. Qt is heavily used by them

CLI

No seperate CLI is to be created due to no foreseeable use.

API

Maybe non-existant @TODO

Hardware Interfaces

None

Software Interfaces

None

Communication Interfaces

None, apart from the exising interfaces for running TANGO

Performance Requirements

Gazebo has two modes:

1. With GUI
2. Headless, without GUI

Running Gazebo headless impairs its visual functionality, however, all plugins work fine (testing to be done for camera plugins).

On the other hand, headlesss is suitable for machines with low computational power since extra computations must be done for rendering the simulation in the GUI mode, which would be done only for the sensors in case of a headless run.

Logical View

Gazebo is used to replace an actual device whose mathematical model is known and check if the simulation matches with the actual observations

It is not used for say, finding out the flow of fluid through some device, but rather used for simulating the device given the flow through a model as a replacement for the actual hardware.

Each feature of the robot is implemented through an independant plugin, eg:

- Move the catterpillar drive robot
- Pan/Zoom the camera
- Actuate the gripper
- Report the temperature, humidity, etc.

The feature may be a sensor or an actuator. The actuators have a bare basic model (for GUI mode) covered by a high definition STL to reduce computational load on the simulator. The sensors on the other hand are sometime modeled by only a dot.

As a result, hardware-in-loop simulation is achieved easily by enabling or disabling the required plugins

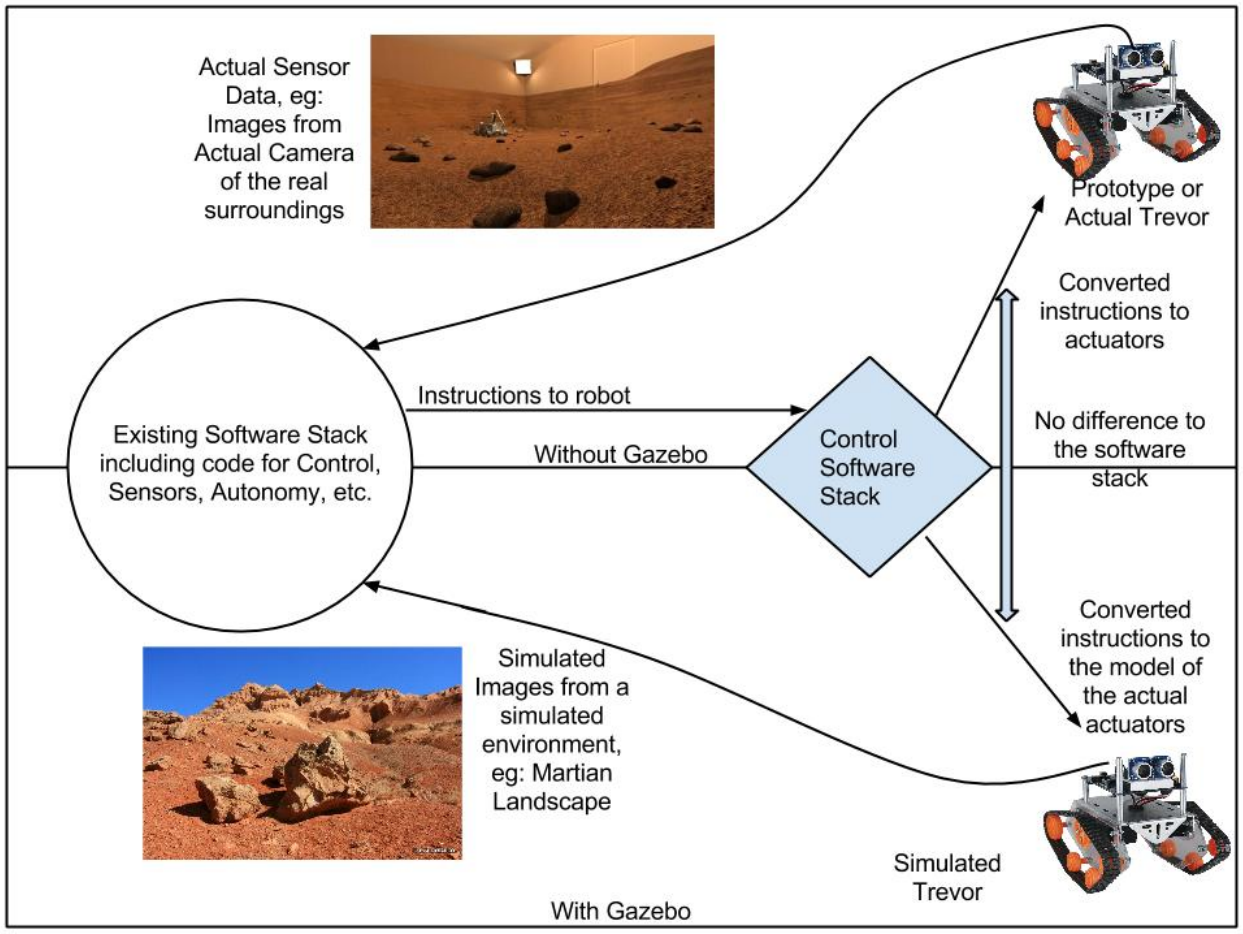
Layers

1. Top Layer: 3D environment changing with time, allowing us to view the STL
2. The simpler models(like cylinders, spheres for more complex surfaces) covered by the STL
3. Mathematical model, used for solving differential equations required for modelling the above models and rendering the images for the top layer
4. Plugins to update the model parameters, pose (x,y,z; r,p,y), forces, torques, constraints, etc. of each of the models. The plugins would be in C++
5. A method to control the plugins, say via TANGO Events/Requests. The events/requests can be sent via a Python/Java/C++ interface
6. An independant user/autonomous program to send the required data for simulation and store/view/use the results

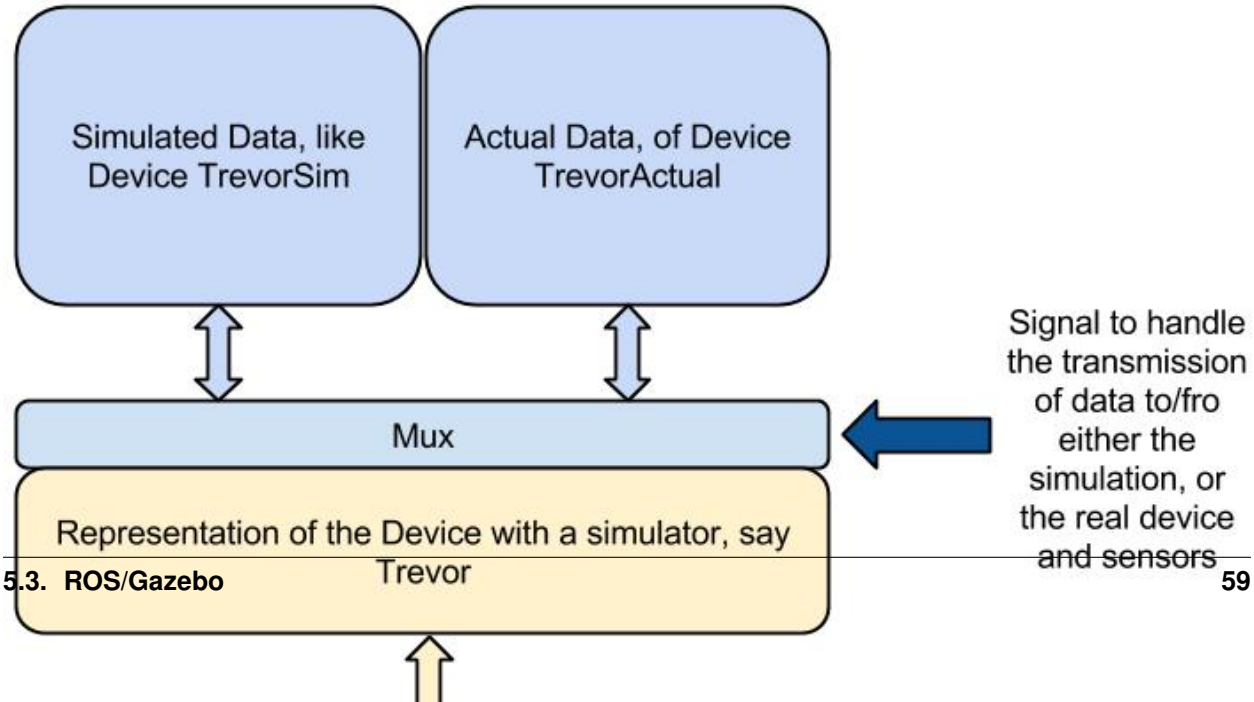
Sub-systems

Use Case Realizations

Implementation View



Deployment View



Actors

Priority

Preconditions

Basic Course

Alternate Course

Exception Course

Postconditions

Notes

5.4 ERAS Habitat Monitoring

5.4.1 1 Habitat Monitor Software User and Maintenance Manual

Author Ambar Mehrotra

1.1 Introduction

1.1.1 Purpose

This document describes the installation, use and maintenance of the Health Monitor.

1.1.2 Applicable Documents

- [1] – Software Engineering Practices Guidelines for the ERAS Project
- [2] – Software Architecture Document for the Habitat Monitor
- [3] – TANGO distributed control system
- [4] – PyTANGO - Python bindings for TANGO
- [5] – Tango Setup
- [6] – Qt Installation
- [7] – PyQt Installation
- [8] – Adding a new Server in Tango

1.1.3 Glossary

API Application Programming Interface

AS Aouda Device Server

ERAS European Mars Analog Station

GUI Graphic User Interface

HM Habitat Monitor Device Server

IMS Italian Mars Society

TBC To Be Confirmed

TBD To Be Defined

1.2 Installation Guide

The first step is to download the component to install (Health Monitor Daemon, Aouda Daemon or Health Monitor GUI) in the machine that is going to run it. The components can be installed all in the same computer, all in different computers or any combination thereof.

1.2.1 Installing the Central Tango Daemon on the Central Tango Server

You can install this component following the Tango Setup guide. Tango's libraries must be installed in all computers.

1.2.2 Installing the Health Monitor Daemon

1.2.2.1 Prerequisites

- Python 2.7
- **Python modules:**
  - python-qt4
  - numpy >= 1.8.1
  - pandas >= 0.14.0
  - pip >= 1.5.4
  - pyqtgraph
  - PyTango >= 8.1.5
  - pymongo >= 3.0.3
- libboost-python-dev >= 1.54
- MongoDB

1.2.2.1.1 Python 2.7, numpy, pip and scipy Python 2.7 comes pre-installed, but just in case you can install it, together with numpy, pip and scipy, with:

```
sudo apt-get install -y libboost-python-dev python2.7 python-pip python-numpy python-scipy
```

1.2.2.1.2 PyTango

sudo pip install PyTango –egg

1.2.2.1.3 MongoDB

1. Import the public key used by the package management system. The Ubuntu package management tools (i.e. dpkg and apt) ensure package consistency and authenticity by requiring that distributors sign packages with GPG keys. Issue the following command to import the MongoDB public GPG Key:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
```

2. Create a list file for MongoDB. Create the /etc/apt/sources.list.d/mongodb-org-3.0.list list file using the following command:

```
echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release -sc)"/mongodb-org/3.0 multiverse" |
```

3. Reload local package database.

```
sudo apt-get update
```

4. Install the latest stable version of MongoDB.

```
   sudo apt-get install -y mongodb-org

Directly from binaries:
~~~~~~~~~~~~~~~~~~~~~~
Follow the following tutorial - http://docs.mongodb.org/manual/tutorial/install-mongodb-on-linux
```

1.2.2.1.4 pymongo

sudo pip install pymongo

1.2.2.1.5 pyqtgraph:

Download the appropriate installer for your Operating System and install from the following link:
http://www.pyqtgraph.org/

1.3 Walkthrough

1.3.1 Start the GUI

1. Navigate to the mongodb directory and start the mongodb daemon

```
sudo ./mongod
```

You can also start it by issuing the following command if you have exported it to system path

```
sudo mongod
```

2. Start the servers that you want to monitor through the GUI. For example:

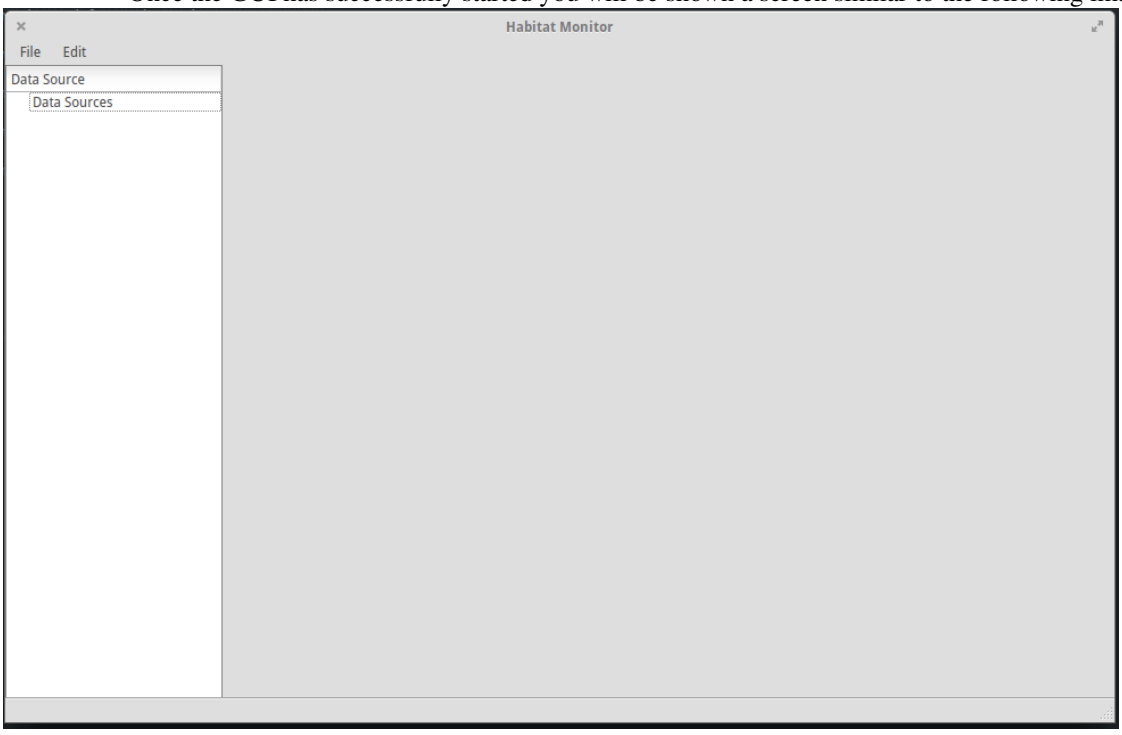
```
sudo aouda 1 simulate_data
```

If you want to add an aouda server to the GUI to monitor.

3. Navigate to the Habitat Monitor directory and start the application by issuing the following command:

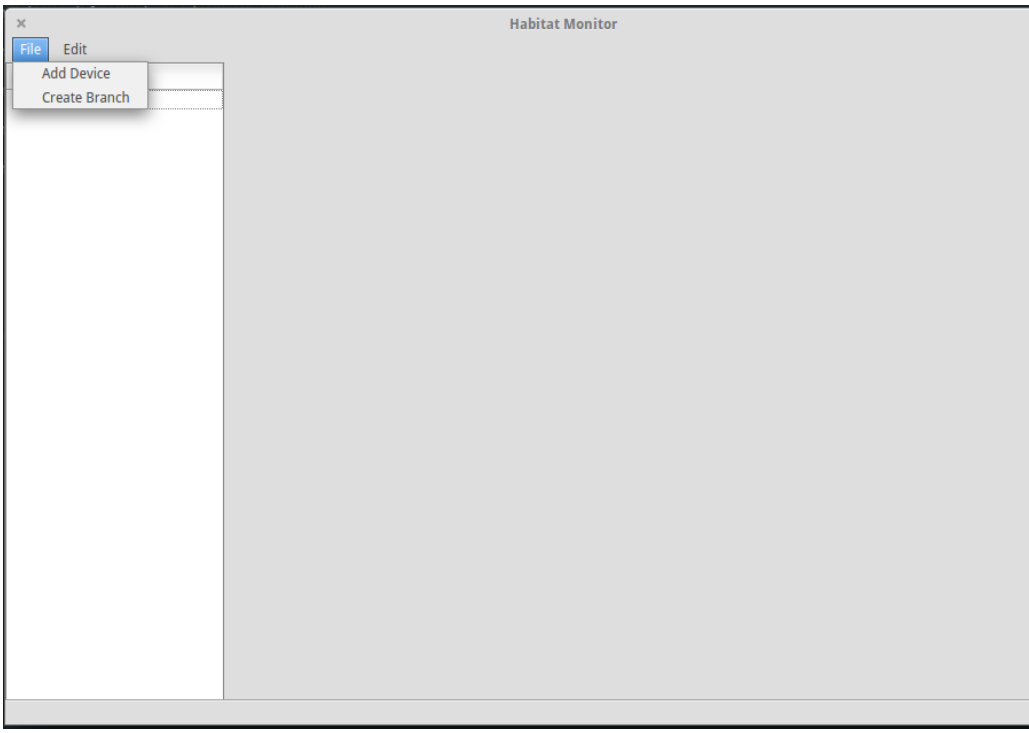
```
python app.py
```

- Once the GUI has successfully started you will be shown a screen similar to the following image:

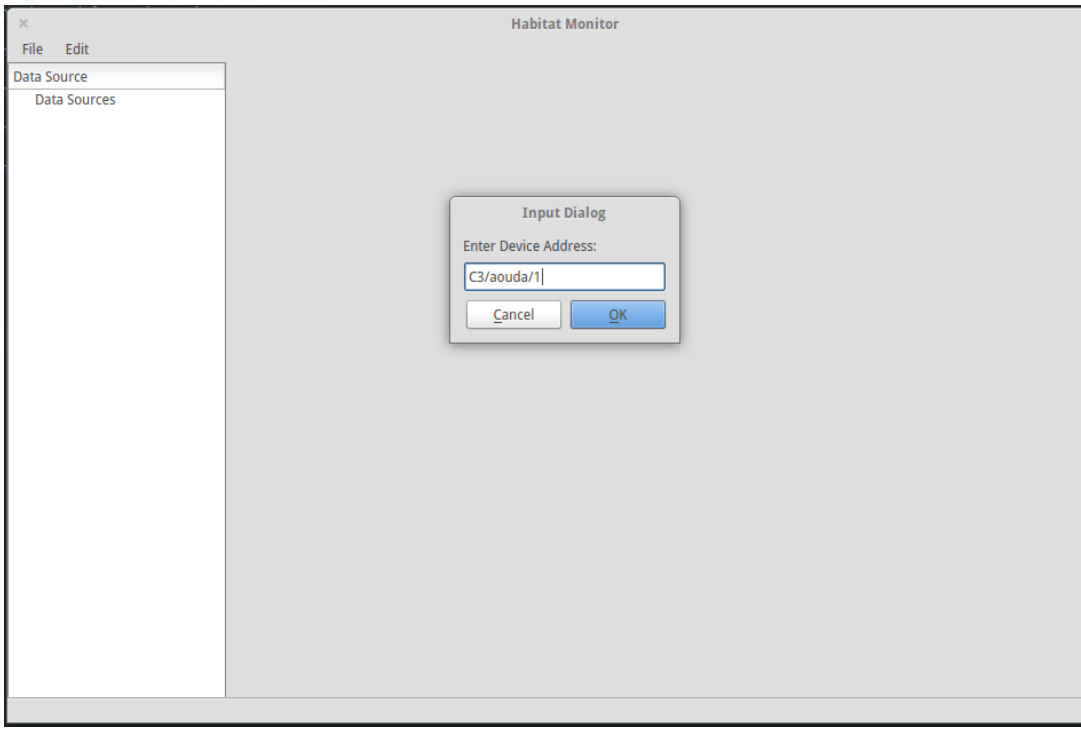


1.3.2 Adding Devices

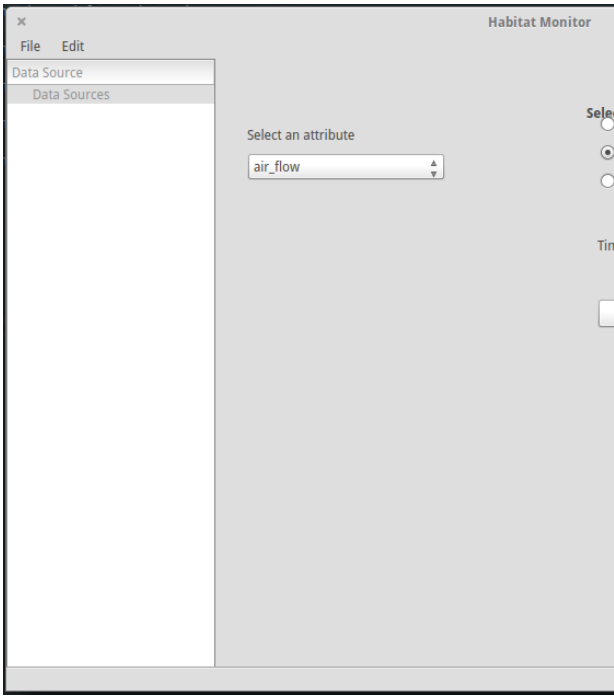
1. Click on ‘File’ menu.



2. **Click on ‘Add Device’ option.**

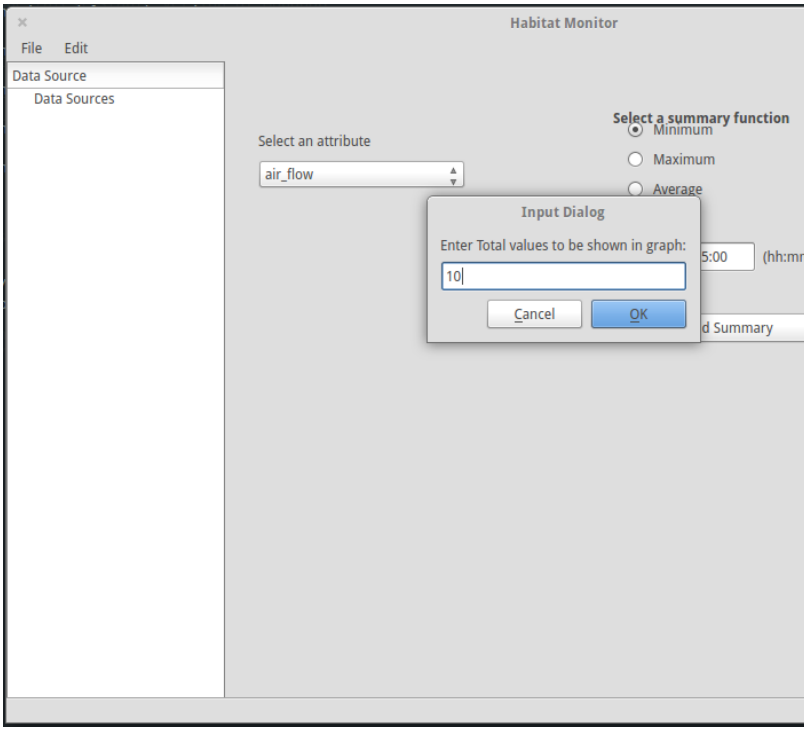


3. **Enter the device address.**
4. **Select and attribute from the device and a summary function. Also enter the summary period.**

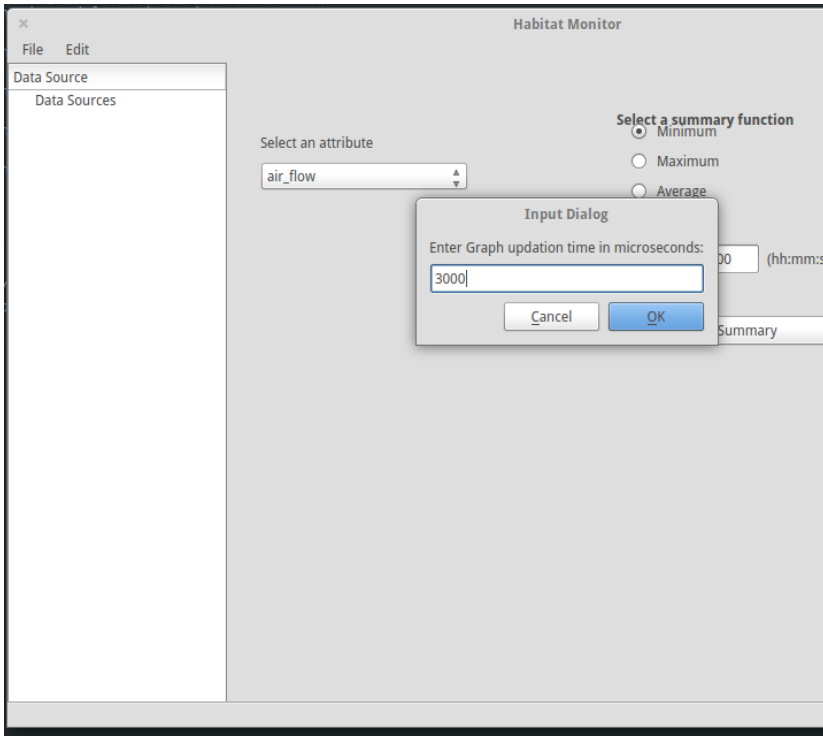


The summary period should be in the format 'hh:mm:ss.ms'.

5. Click on 'Add Summary' button.

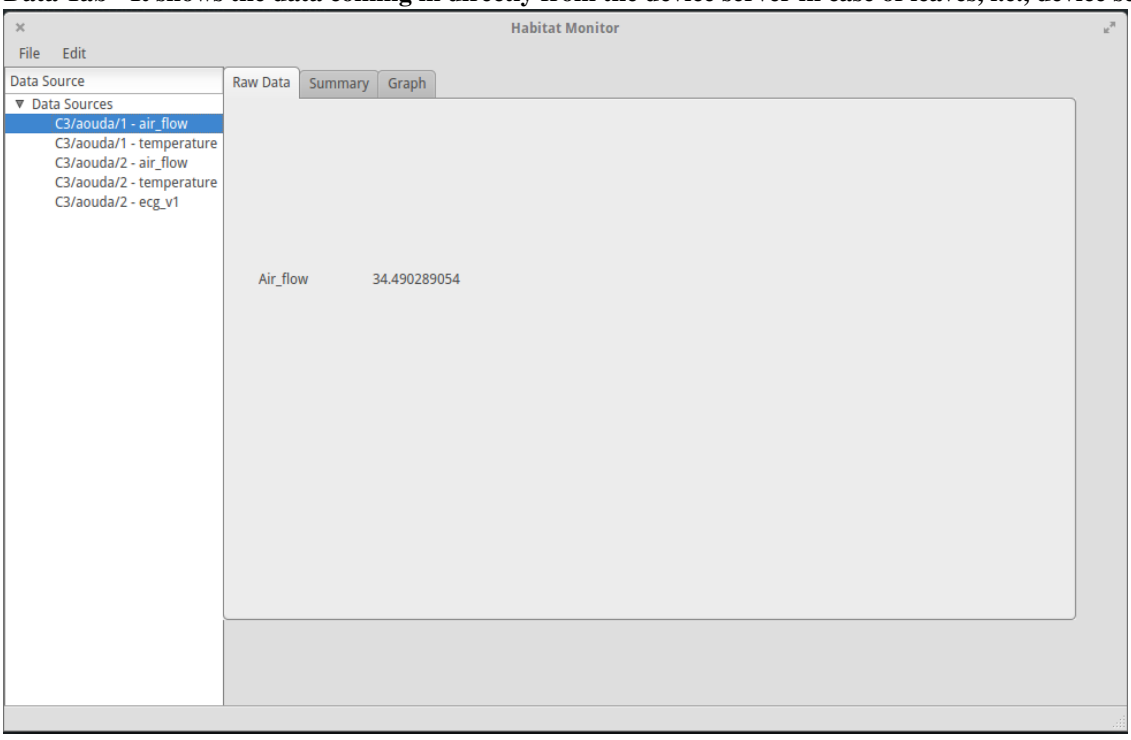


6. **Enter the 'Total values to be shown in the graph'.**

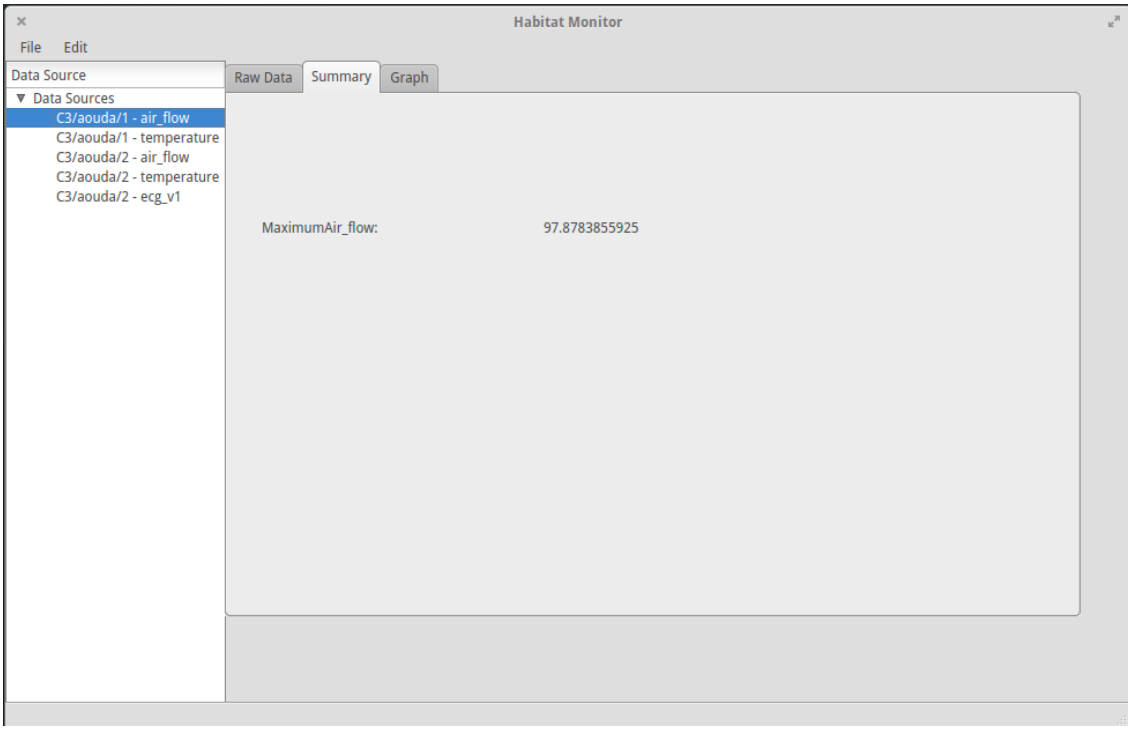


7. **Enter the graph updation time in microseconds.**

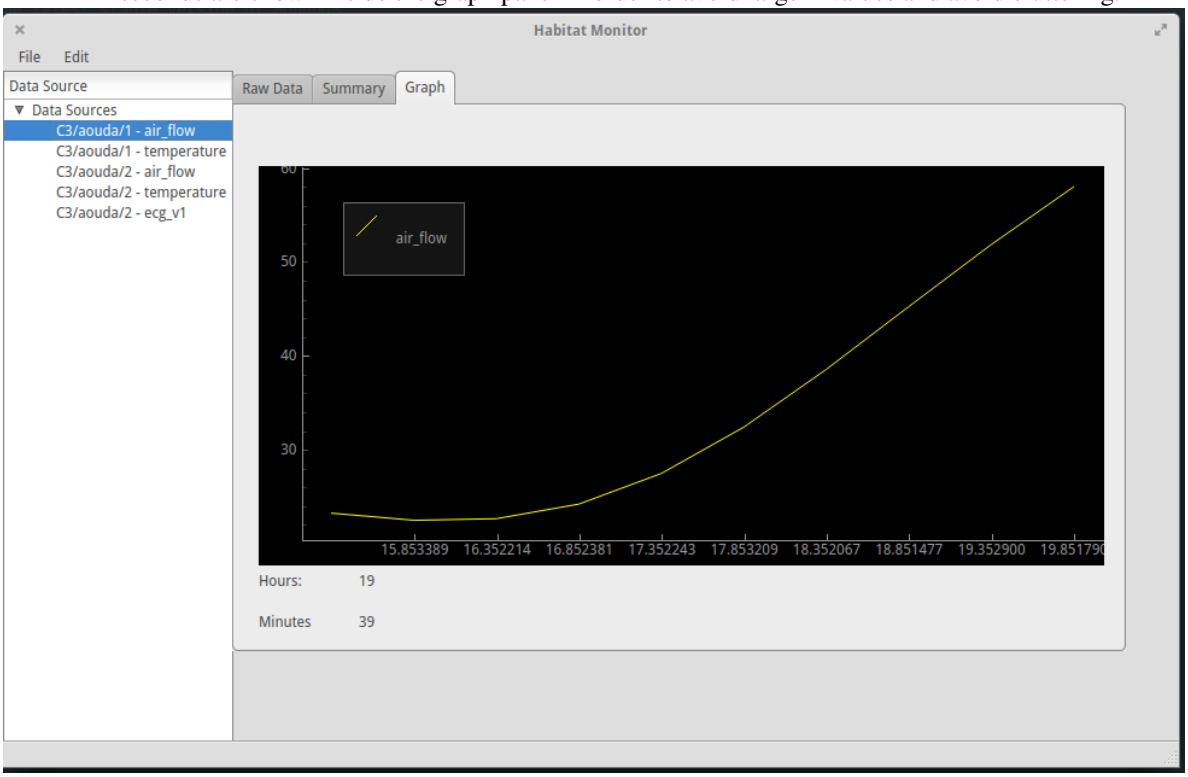
8. **Raw Data Tab - It shows the data coming in directly from the device server in case of leaves, i.e., device servers.**



9. **Summary Tab - It shows the summary as calculated by the summary function in the provided time period**

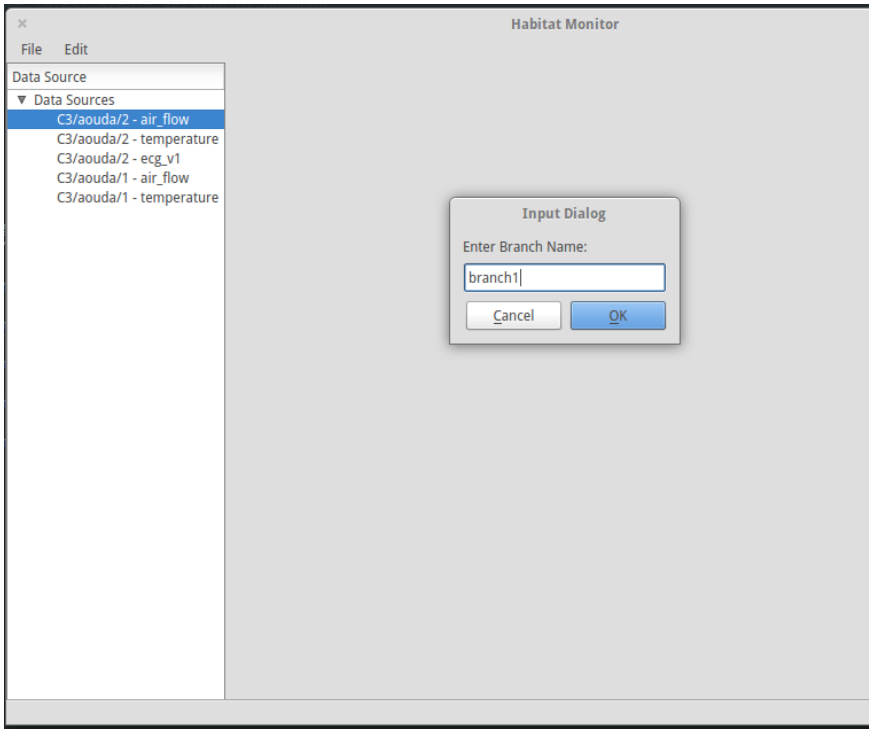


10. Graph Tab - It shows the real-time graph of the raw data according to the total number of values mentioned while adding a device and the graph updation frequency. User can edit these values via the 'graph_config' file inside the application directory. The hours and minutes are shown below of the graph panel while seconds and milliseconds are shown inside the graph panel in order to avoid large x-values and avoid cluttering.

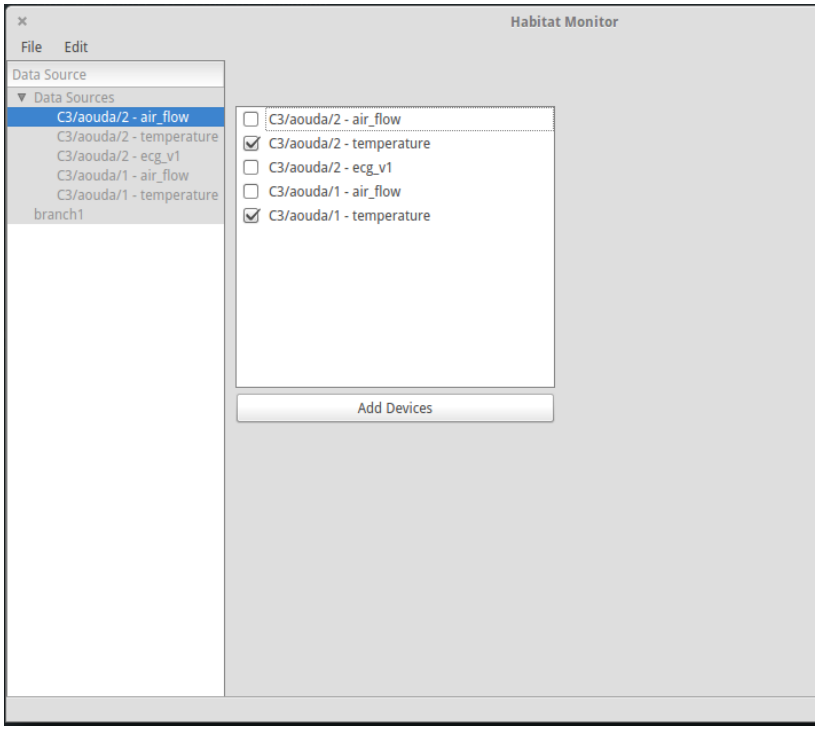


1.3.3 Creating Branches

1. Click on ‘File’ menu.
2. Click on ‘Create Branch’ option.

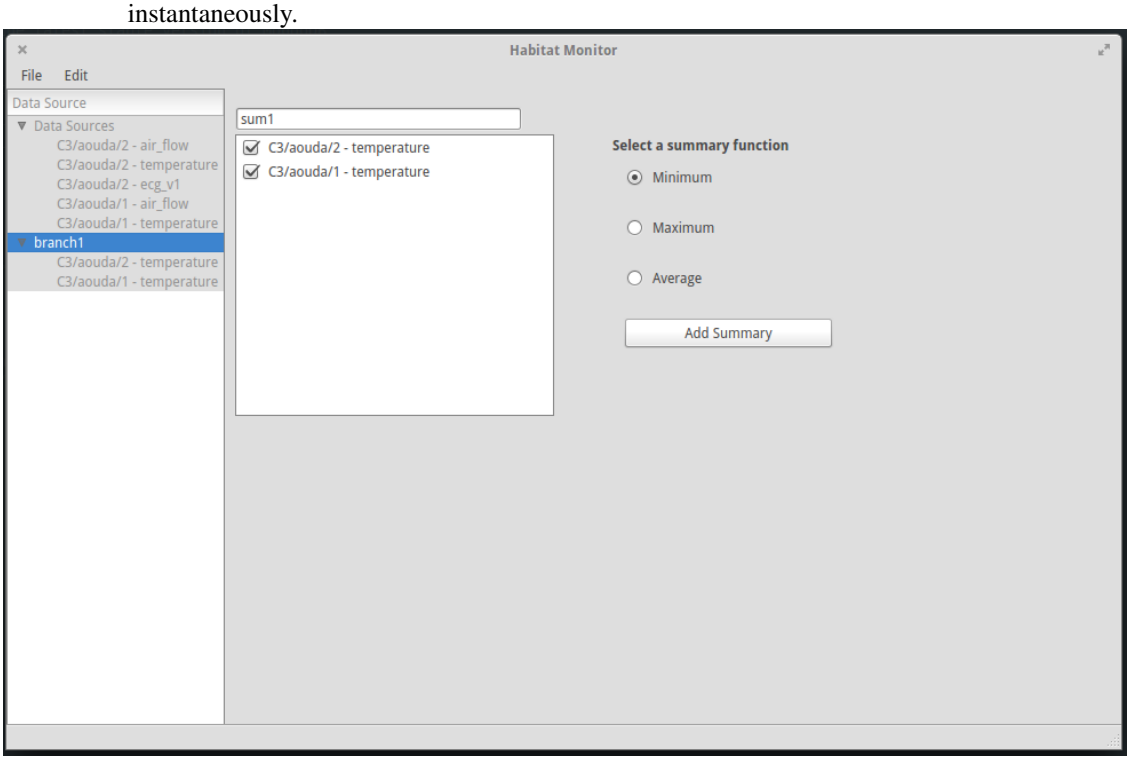


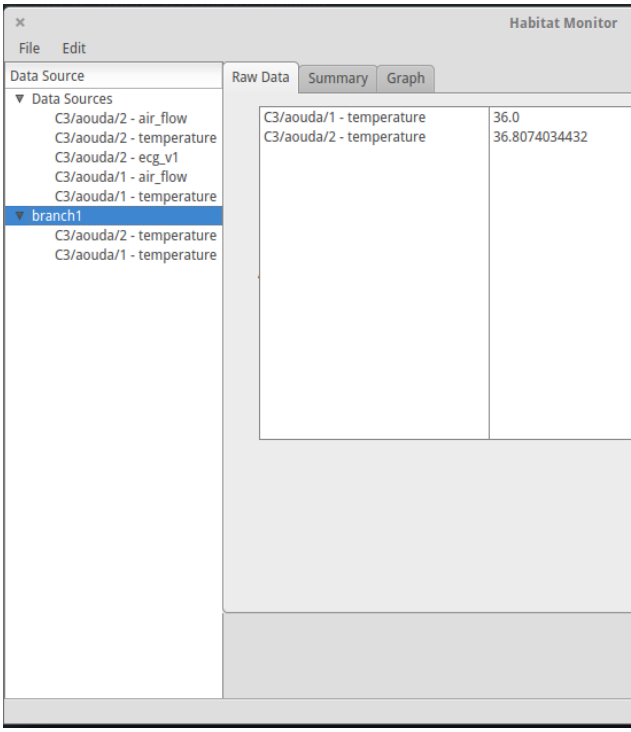
3. **Enter the branch name and click ok button.**



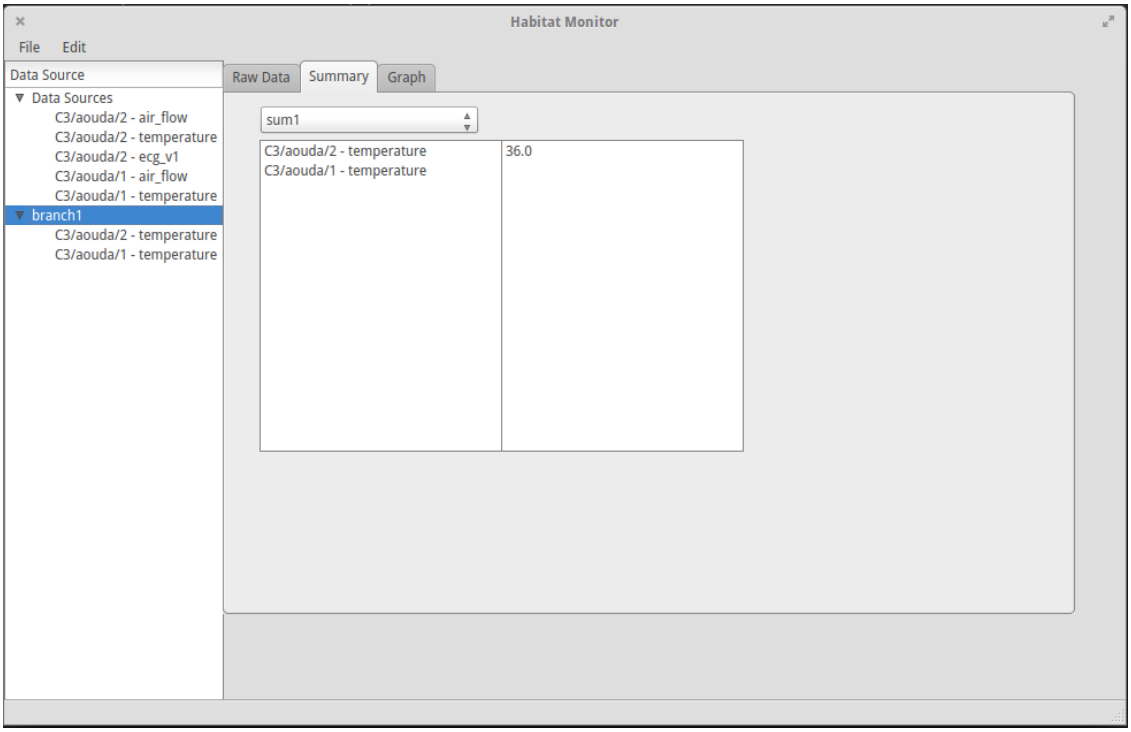
4. **Select the devices you want to add to the branch.**
5. **Summary Creation - Enter summary name and select a summary function.** Unlike the leaf summary, there in no option to provide the summary time in case of branches as branch summary is calculated

instantaneously.





6. **Raw Data Tab - Shows the raw data for each child of the branch.**
7. **Summary Tab - Shows the summary as defined by the summary function and summary name in the drop down box.**

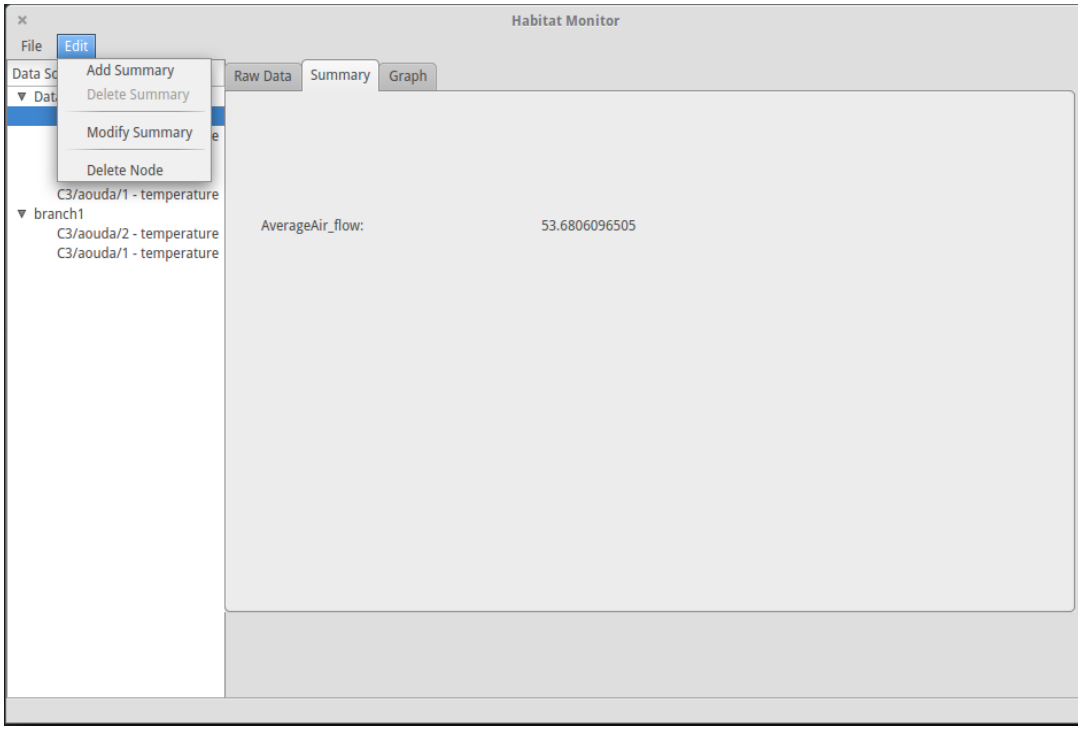


8. **Graph Tab - Shows the graph of the raw data. I avoided showing all graphs in the same pannel so as to avoid cluttering.** You can select the child from the dropdown menu and view its graph. .. image:: images/tutorial/15.PNG

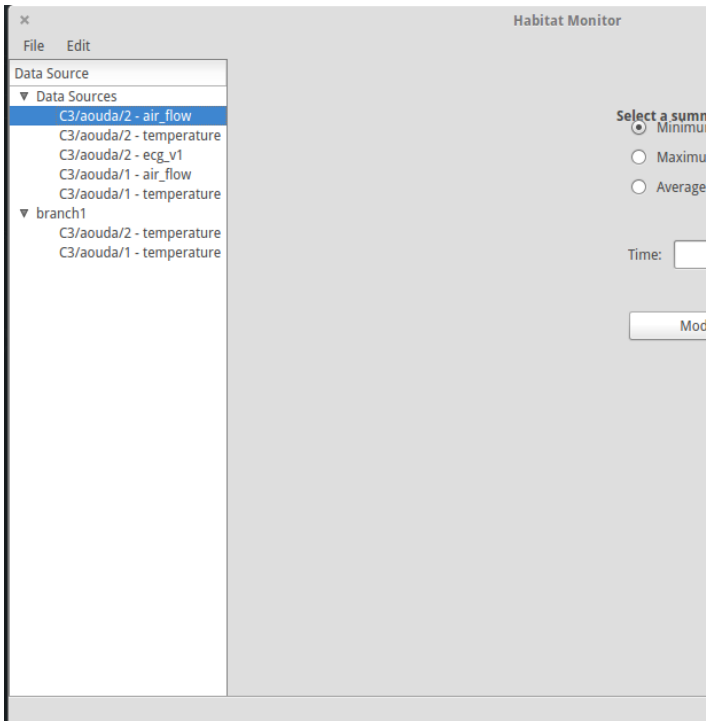
1.3.4 Modifications

1.3.4.1 Summary Modification

- Summary modification is only for leaves, i.e., the data sources



- **Click on the 'Edit' menu.**
- Click on the 'Modify Summary' option.

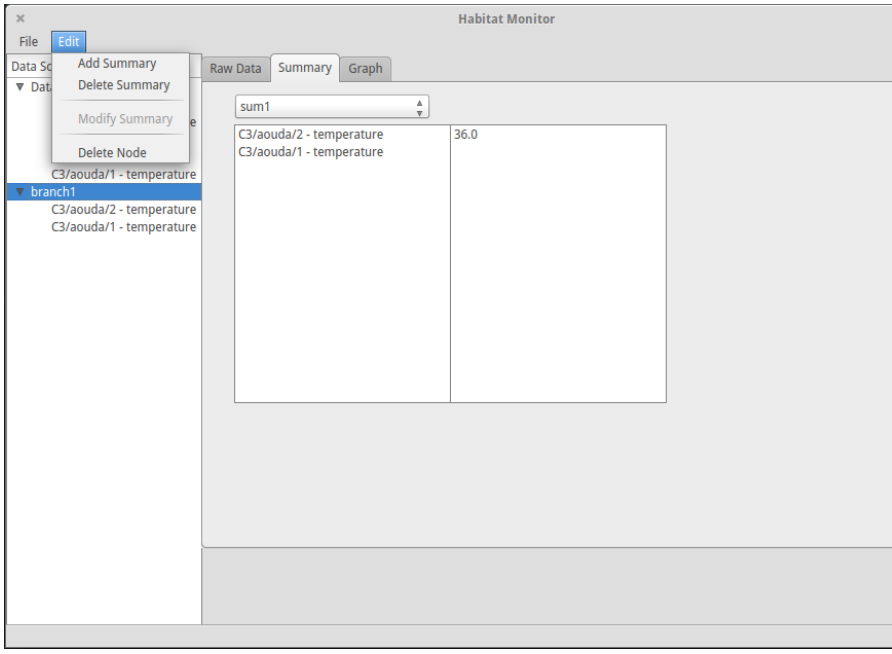


- **Select the summary function and enter the summary time.**
- Click on 'Modify Summary' button.

1.3.4.2 Summary Addition

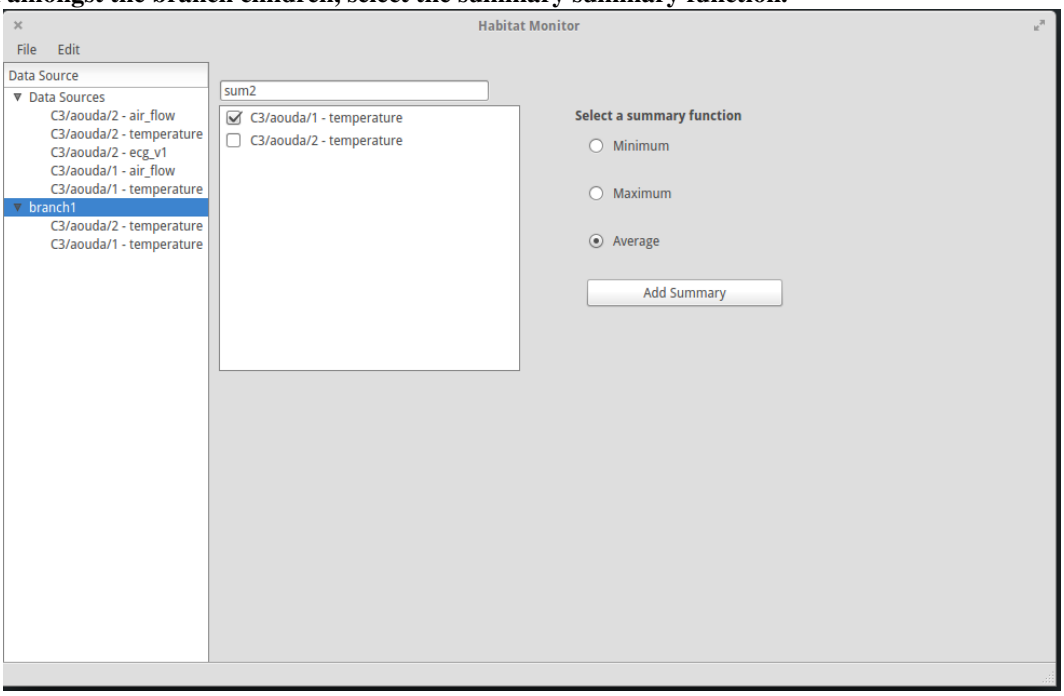
- Summary addition works only for branches.

- Click on the 'Edit' menu.



- **Click on the 'Add Summary' option.**
- Enter the summary name, select the nodes you want to be added to the summary

from amongst the branch children, select the summary summary function.

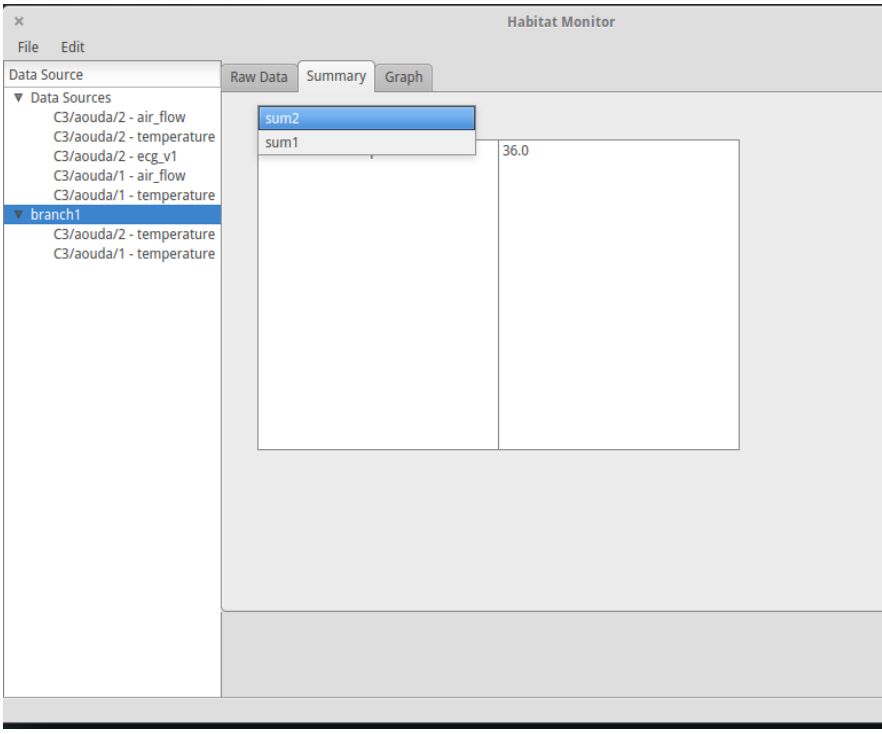


- Click on 'Add Summary' button.

1.3.4.3 Summary Deletion

- Summary addition works only for branches.
- Click on the branch in the left pane.

- Navigate to the 'Summary' tab.



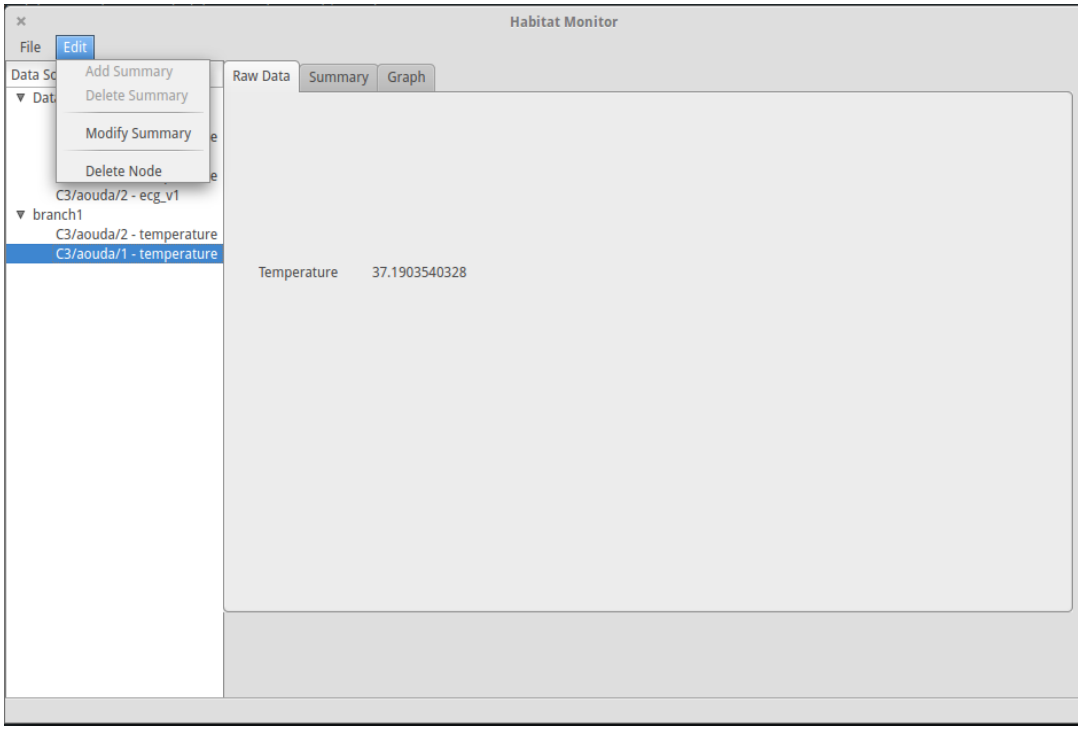
- **Select a summary from the dropdown box.**
- Click on the 'Edit' menu.
- Click on the 'Delete Summary' option.
- Click on 'Yes' option in the popup to delete the summary.

1.3.5 Node Deletion

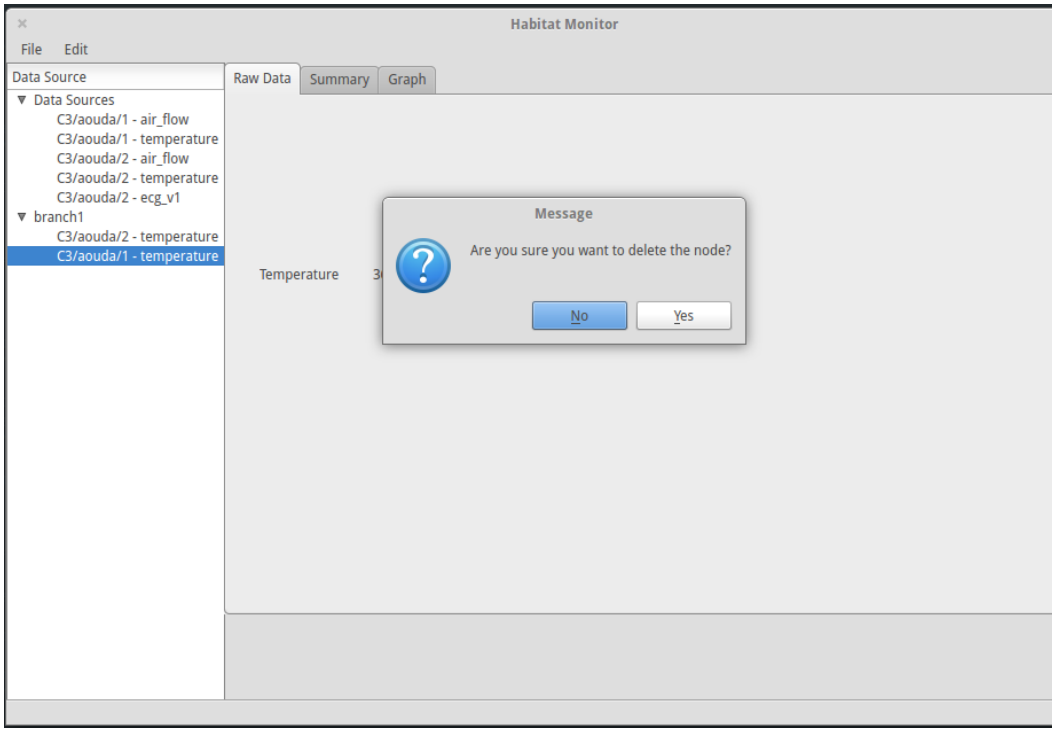
If you delete a node from the data source, then it will be deleted from everywhere, i.e., from under all branches. But if you delete it from under a specific branch, it will only be deleted from there.

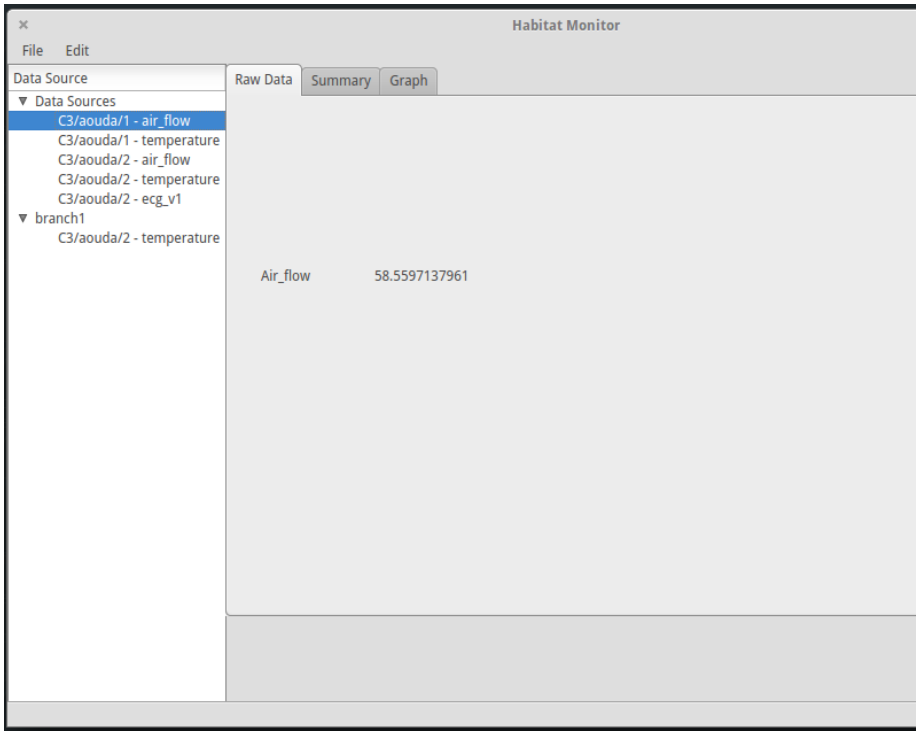
- Click on the node you want to delete.

- **Click on the 'Edit' menu.**



- **Select 'Delete Node' option.**





- **Click on 'Yes' option to delete the node.**

5.4.2 2 Software Architecture Document for the Habitat Monitoring and Alarming Interface

Author Ambar Mehrotra

2.1 Change Record

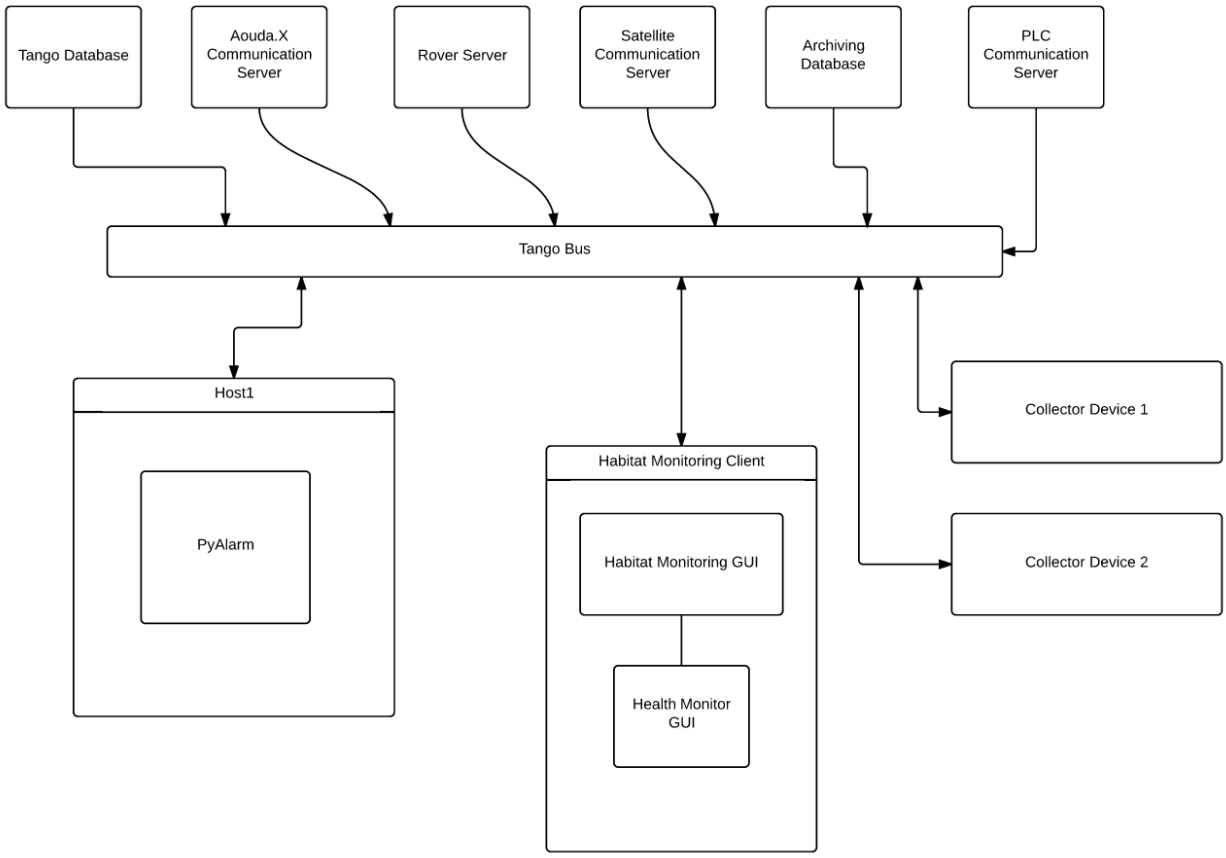
25th May, 2015 - Document Created

2.2 Introduction

2.2.1 Purpose

The purpose of Alarming and Monitoring interface is to manage all the relevant information. The GUI will give the user a complete overview of the entire habitat and how all the instruments are functioning.

The Health Monitor *GUI* will be dedicated to provide information about health of the astronauts performing EVA through biosensors (ECG, air flow sensor , etc.).



2.2.2 Scope

Describes the scope of this requirements specification.

2.2.3 Applicable Documents

- [1] – How to use Tango Controls
- [2] – How to PyTango
- [3] – PyQt4 Reference Guide

2.2.4 Reference Documents

- [1] – C3 Prototype document v.4
- [2] – Software Engineering Practices Guidelines for the ERAS Project
- [3] – Tango Setup
- [4] – Adding a new server in Tango
- [5] – Extending Alarm Handling in Tango

2.2.5 Glossary

IMS Italian Mars Society

ERAS European MaRs Analogue Station for Advanced Technologies Integration

V-ERAS Virtual-ERAS

GUI Graphic User Interface

API Application Programming Interface

PANIC Package for Alarms and Notification of Incidences from Controls

TBD To be defined

TBC To be confirmed

2.2.6 Overview

Make an overview in which you describe the rest of this document the and which chapter is primarily of interest for which reader.

2.3 Architectural Requirements

This section describes the requirements which are important for developing the software architecture.

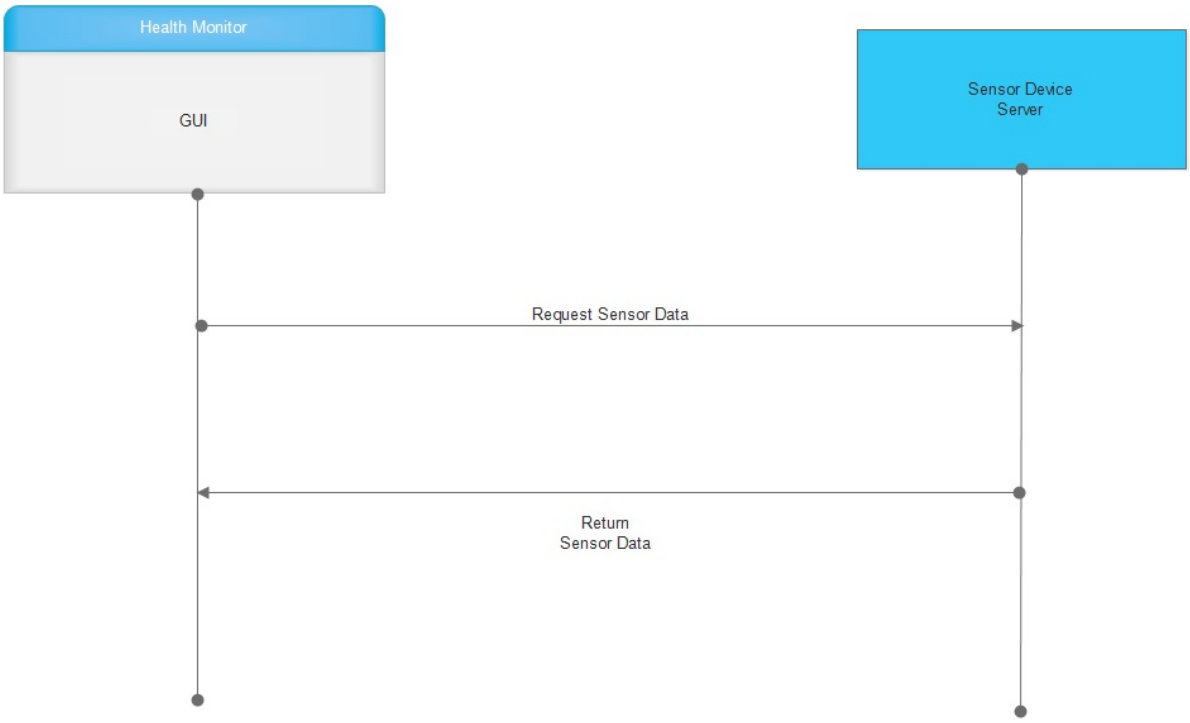
2.3.1 Non-functional requirements

1. **Implementation Constraints**
    (a) **Language** The application should be written in python.
    (b) **Operating System** The application should be run on Ubuntu distributions.
    (c) **Software** PyQt Library PyTango Library Tango server(pyTango), Python 2.x, Pep8,
2. **Supportability**
    (a) **Ease of Installation** System requires installation of PyQt and Tango server.

2.3.2 Use Case View (functional requirements)

The goal of this project is to build a service that allows the central monitoring of the entire habitat. A GUI will request the data from the database using the *PANIC API*, summarize it and present it to an overseer in a way that allows him/her to detect problems at a glance.

2.3.2.1 Request for sensor data The Client requests a Network Device Server for the sensor data of the last T seconds.



2.3.2.1.1 Actors

- Client: Habitat Monitoring *GUI*.
- Server: the Device TANGO server.

2.3.2.1.2 Priority High

2.3.2.1.3 Preconditions The Server is running and its DevState is ON.

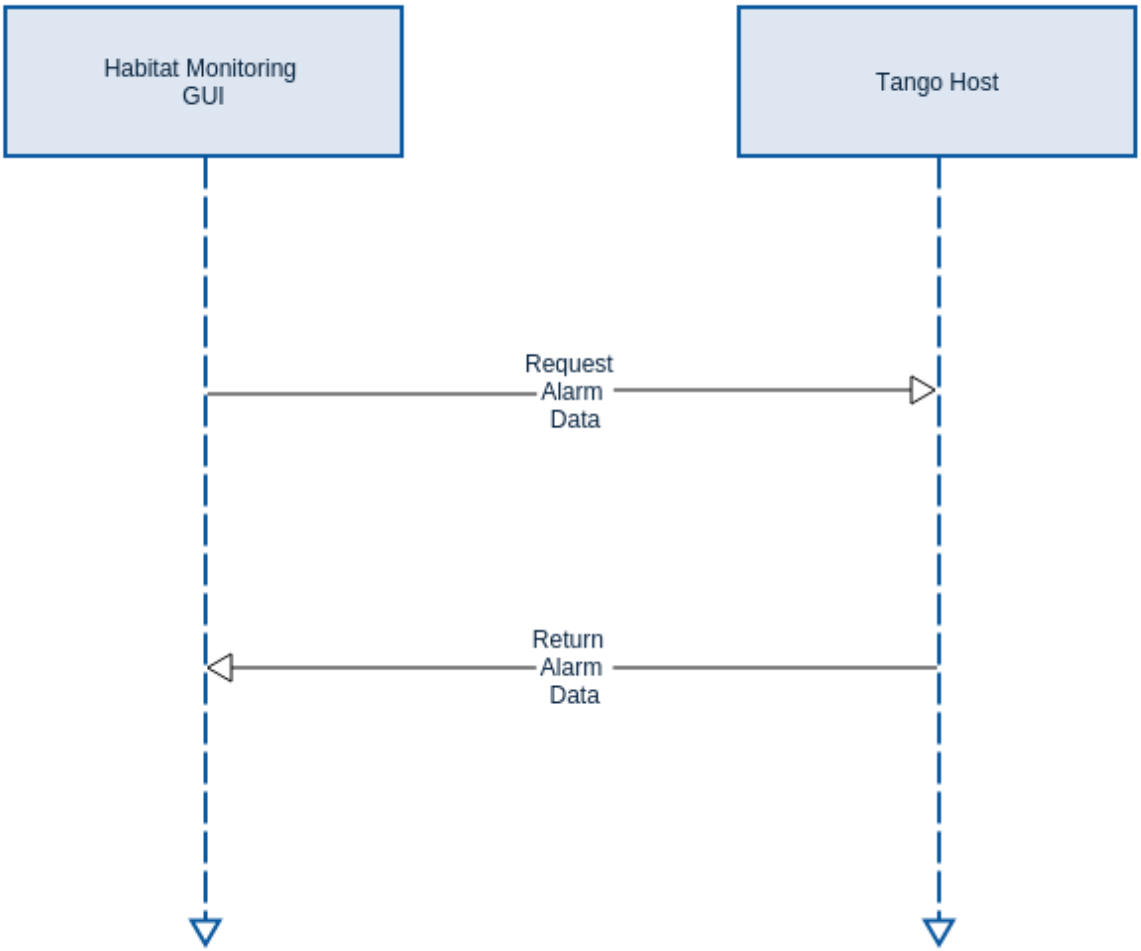
2.3.2.1.4 Basic Course

1. The Client calls the appropriate method on the Server, passing T as argument.
2. The Server searchs its buffer for the appropriate records.
3. The Server returns the records found.

2.3.2.1.5 Alternate Course None

2.3.2.1.6 Postconditions The server returns the data requested or an empty array if no data is available.

2.3.2.2 Request for alarms The Client request the Tango host for the alarm data of the last T seconds.



2.3.2.2.1 Actors

- Client: Habitat Monitoring GUI.
- Server: The Tango host responsible for collecting alarm data.

2.3.2.2.2 Priority High

2.3.2.2.3 Preconditions The Server is running and its DevState is ON.

2.3.2.2.4 Basic Course

1. The Client calls the appropriate method on the Server, passing T as argument.
2. The Server searchs the database for the appropriate records.
3. The Server returns the records found.

2.3.2.2.5 Alternate Course

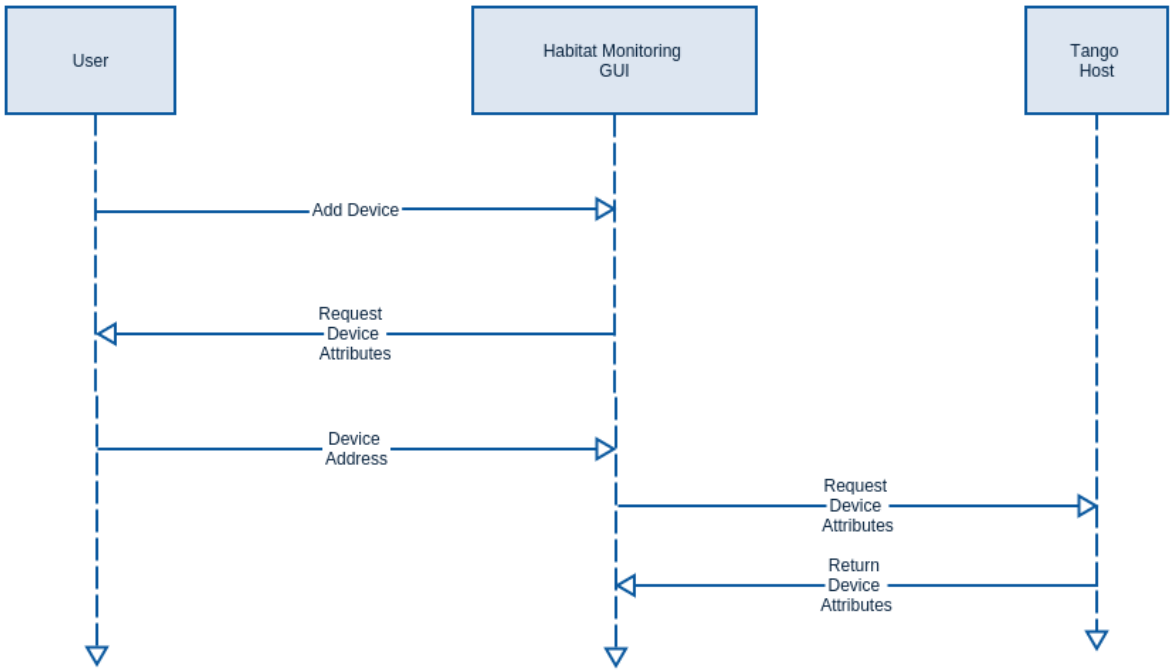
None

2.3.2.2.6 Postconditions

The server returns the data requested or an empty array if no data is available.

2.3.2.3 User requests to add new device

The user wants to add a new Tango Device on the network to monitor using the *GUI*.



2.3.2.3.1 Actors

- User: The user who wants to add a new Device.
- Client: the Habitat Monitor TANGO client.
- Device Server: Tango server running on the network.

2.3.2.3.2 Priority

High

2.3.2.3.3 Preconditions

The Server is running and its DevState is ON. The *GUI* is running.

2.3.2.3.4 Basic Course

1. The user requests the GUI to add a new Device.
2. The *GUI* asks the user for the internal Tango Device address.
3. The user responds with the device address.
4. The *GUI* queries the device for attributes.
5. The device returns the required attributes.

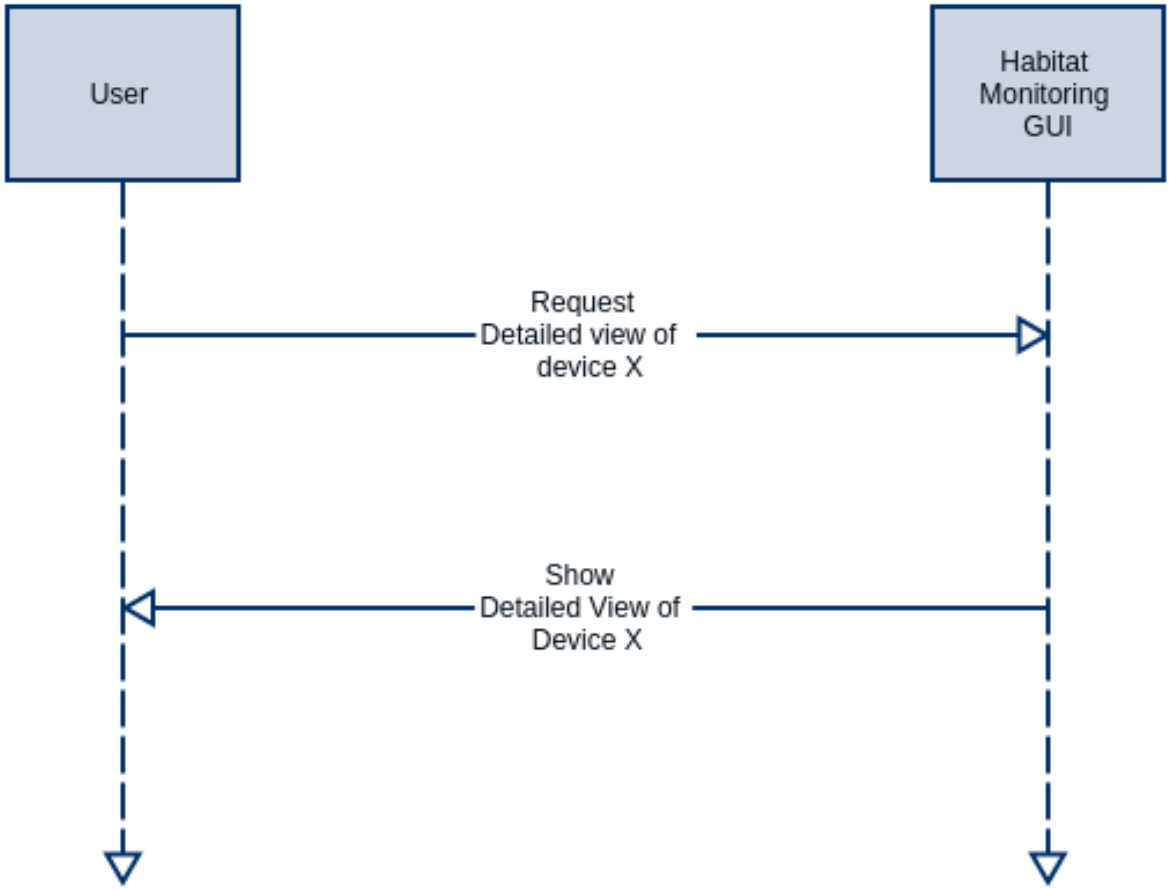
6. The GUI starts showing its data.

2.3.2.3.5 Alternate Course None

2.3.2.3.6 Exception Course None

2.3.2.3.7 Postconditions The Device data is shown on the screen or *GUI* shows an error message if the device is not found on the given address.

2.3.2.4 A User requests a device's detailed data A user requests the detailed data for a given device and the GUI complies.



2.3.2.4.1 Actors

- User: a user of the GUI.
- GUI: a GUI with an embedded TANGO client.

2.3.2.4.2 Priority High

2.3.2.4.3 Preconditions The Server is running and its DevState is ON.

2.3.2.4.4 Basic Course

1. The User clicks on the icon of device.
2. The GUI hides the summarized view for device.
3. The GUI shows the detailed view for device.

2.3.2.4.5 Alternate Course None

2.3.2.4.6 Exception Course None

2.3.2.4.7 Postconditions The selected device's detailed view is shown on the GUI.

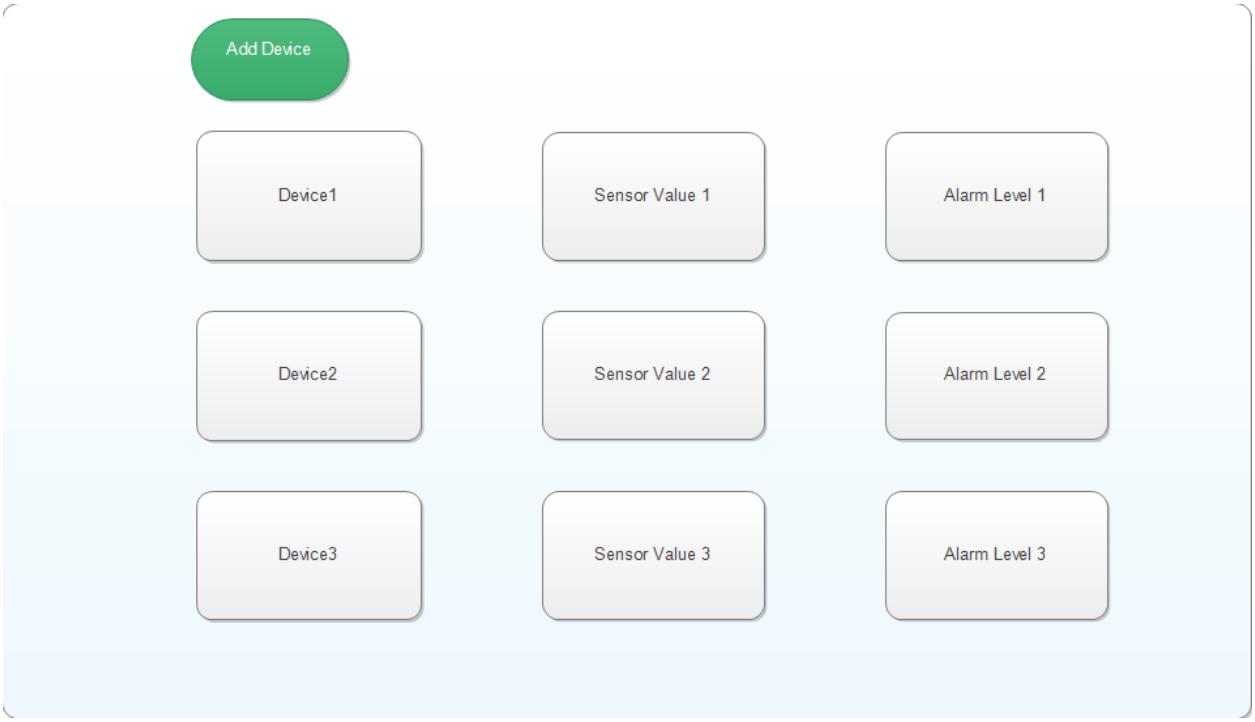
2.4 Interface Requirements

2.4.1 User Interfaces

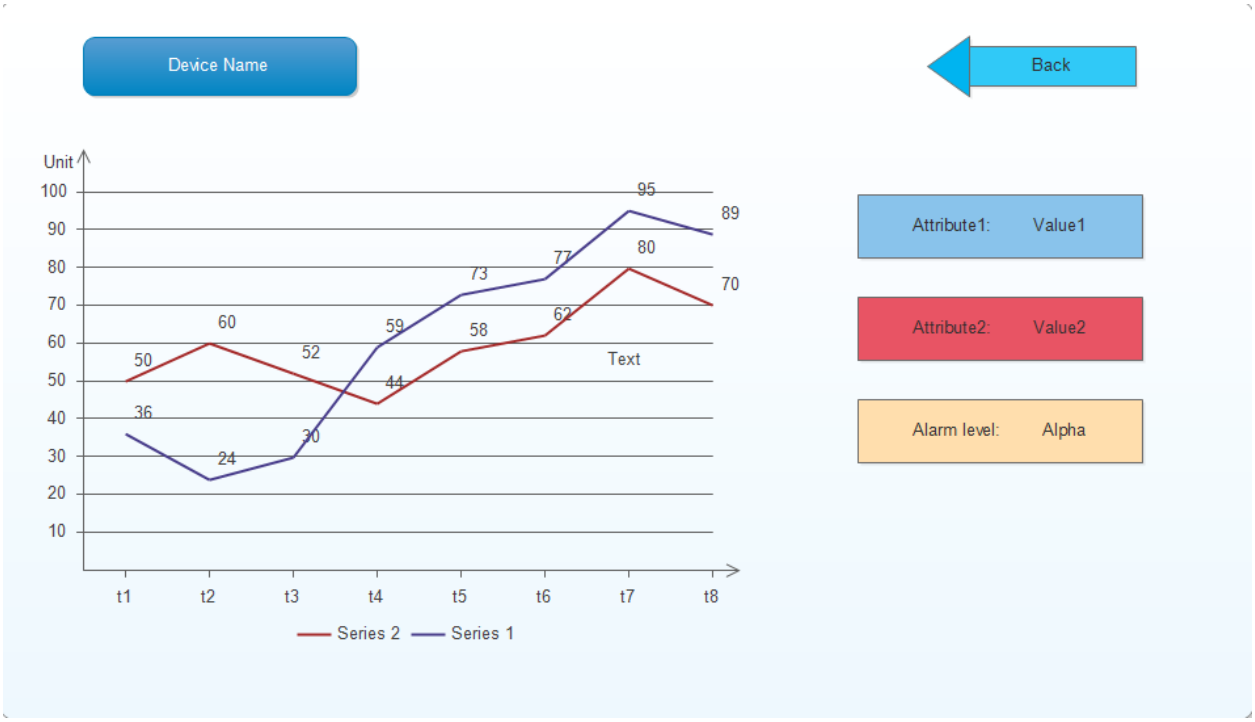
Describes how this product interfaces with the user.

Bellow are two mockups that cover the two current Use Cases that concern the GUI.

2.4.1.1 Graphical User Interface



2.4.1.1.1 Overview



2.4.1.1.2 Detailed View

2.4.2 Software validation and verification

The *GUI* will be implemented as a Tango Client that will fetch data from the varios device servers and show it in a concise manner.

2.4.3 Planning

The development of the GUI will be done in primarily the following phases.

- Building the skeleton for the GUI. This is the primary portion of the project

and will require the work on the following areas.

- Allowing the GUI to add additional data channels.
- Integration with the Tango Alarms System
- Integrating the monitoring system with the plottings coming in from the

various biometric devices using a generic mechanism.

- Development of the Health Monitoring module as a sub-GUI of the habitat

monitoring interface.

5.5 Health Monitor Server

5.5.1 1 Health Monitor Software User and Maintenance Manual

Author Mario Tambos

1.1 Change Record

2014.05.16 - Document created. 2014.08.20 - Actual manual written. 2015.05.27 - Manual completed

1.2 Introduction

1.2.1 Purpose

This document describes the installation, use and maintenance of the Health Monitor.

1.2.2 Applicable Documents

- [1] – C3 Prototype document v.4
- [3] – Software Engineering Practices Guidelines for the ERAS Project
- [4] – PAMAP2 Physical Activity Monitoring
- [5] – Software Architecture Document for the Health Monitor
- [6] – TANGO distributed control system
- [7] – PyTANGO - Python bindings for TANGO
- [8] – Tango Setup
- [9] – wxPython Installation
- [10] – Adding a new Server in Tango

1.2.3 Glossary

AD Anomaly Detection

API Application Programming Interface

AS Aouda Device Server

ERAS European Mars Analog Station

GUI Graphic User Interface

HM Health Monitor Device Server

HMGUI Health Monitor Graphical User Interface

IMS Italian Mars Society

TBC To Be Confirmed

TBD To Be Defined

1.3 Overview

1.3.1 Hardware Architecture

The different hardware components that need to be taken into account are shown in the Deployment Diagram below. As done at the moment all software components can be run in a single computer, they can however also be run each in a different machine.

One key assumption is that one instance of the *AS* will monitor one single Suit. In other words, one instance of the *AS* is needed for each crew member during EVA.

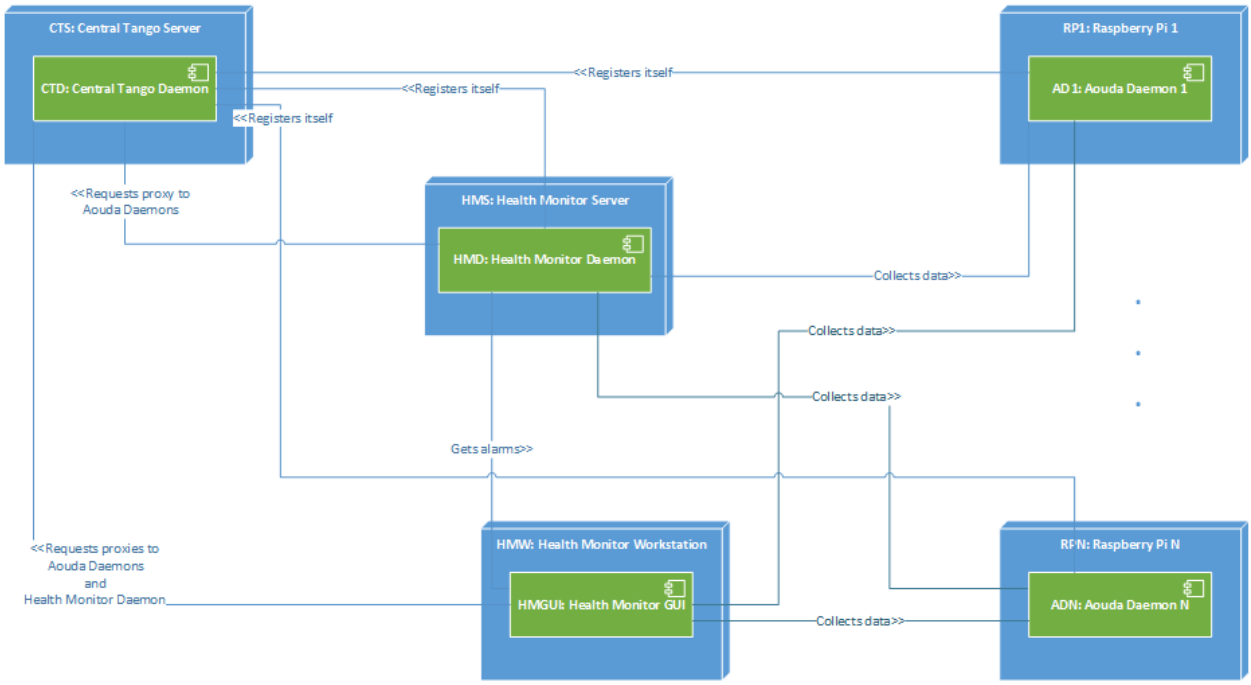
1.3.2 Software Architecture

The components involved can be divided in five categories:

1. The Central Tango Daemon: It keeps track of the existing Tango Device Servers. For details refer to [7] and [8]. In the context of deployment, the computer that runs the Central Tango Daemon is called "Central Tango Server".

2. The Aouda Device Server (*AS*): This componen can either run on a Raspberry Pi and read the sensors provided by an e-Health shield, or run on any machine and simulate the sensors. In the context of deployment, the computers that run the *AS* are called "Raspberri Py i" and the *AS* themselvs are called "Aouda Daemon i", where i is an integer.
3. The Health Monitor Device Server (*HM*): Collects data from the *AS* and performs anomaly detection on their sensor readings. In the context of deployment, the computer that runs the *HM* is called "Health Monitor Server" and the *HM* itself is called "Health Monitor Daemon".
4. The Health Monitor Graphic User Interface (*HMGUI*): allows the user to oversee the crew's status. It collects data from the *AS* and the *HM*. In the context of deployment, the computer that runs the *HMGUI* is called "Health Monitor Workstation" and the *HMGUI* itself is called "Health Monitor GUI".

1.3.3 Deployment Diagram



1.4 Installation Guide

The first step is to download the component to install (Health Monitor Daemon, Aouda Daemon or Health Monitor GUI) in the machine that is going to run it. The components can be installed all in the same computer, all in different computers or any combination thereof.

1.4.1 Installing the Central Tango Daemon on the Central Tango Server

You can install this component following the Tango Setup guide. Tango's libraries must be installed in all computers.

1.4.2 Installing the Health Monitor Daemon

1.4.2.1 Prerequisites

- Python 2.7
- **Python modules:**

  - inflect >= 0.2.4
  - numpy >= 1.8.1
  - pandas >= 0.14.0
  - pip >= 1.5.4
  - PyTango >= 8.1.2
  - scipy >= 0.14.0
  - SQLAlchemy >= 0.9.4
  - PyTango >= 8.1.5
- libboost-python-dev >= 1.54

1.4.2.1.1 Python 2.7, numpy, pip and scipy Python 2.7 comes pre-installed, but just in case you can install it, together with numpy, pip and scipy, with:

```
sudo apt-get install -y libboost-python-dev python2.7 python-pip python-numpy python-scipy
```

1.4.2.1.2 inflect, pandas, PyTango and SQLAlchemy inflect, pandas and SQLAlchemy can be installed the normal way:

```
sudo pip install inflect pandas SQLAlchemy
```

PyTango, however, needs an additional parameter:

```
sudo pip install PyTango --egg
```

1.4.2.2 Registering the Health Monitor Daemon

On a Python or IPython console write:

```
import PyTango
dev_info = PyTango.DbDevInfo()
dev_info.server = "health_monitor/1"
dev_info._class = "HealthMonitorServer"
dev_info.name = "C3/health_monitor/1"
db = PyTango.Database()
db.add_device(dev_info)
```

1.4.3 Installing the Aouda Daemon

In order to use the Aouda Daemon to read from the eHealth shield, you need both the shield and a Raspberry Pi running Raspbian. The first step is to get a SD card ready; you can do that by following the official Raspbian installation guide.

Once the SD card is ready, start the Raspberry Pi and log into it.

You can also run the Aouda Daemon in simulation mode from any Ubuntu machine. In this case neither the shield nor the swig package are needed.

1.4.3.1 Prerequisites

- Python 2.7
- **Python modules:**

  - MDP >=3.3
  - numpy >= 1.8.1
  - Oger == 1.1.3
  - pandas >= 0.14.0
  - pip >= 1.5.4
  - PyTango >= 8.1.2
  - scipy >= 0.14.0
  - setuptools >= 3.3
- libboost-python
- libboost-python-dev
- Swig >= 3.3

1.4.3.1.1 Python 2.7, libboost-python, MDP, numpy, pip, scipy and Swig Python 2.7 comes pre-installed, but just in case you can install it, together with numpy, pip and scipy, with:

```
sudo apt-get install -y python2.7 libboost-python-dev python-mdp python-numpy python-pip python-scipy
```

1.4.3.1.2 Pandas and PyTango Pandas can be installed the normal way:

```
sudo pip install pandas
```

PyTango, however, needs an additional parameter:

```
sudo pip install PyTango --egg
```

1.4.3.1.3 Oger This module has to be installed manually. In order to do it, first download the source code: Oger-1.1.3. Then decompress the file and:

```
cd Oger-1.1.3
sudo python setup.py install
```

1.4.3.2 Registering the Aouda Daemon On a Python or IPython console write:

::code

import PyTango dev_info = PyTango.DbDevInfo() dev_info.server = “aouda/1” dev_info._class = “AoudaServer” dev_info.name = “C3/aouda/1” db = PyTango.Database() db.add_device(dev_info)

If you need more than one Aouda Daemon, replace the “1” in the device’s name and server with a different number for each daemon to register.

1.4.4 Installing the Health Monitor GUI

1.4.4.1 Prerequisites

- Python 2.7
- Python modules: + matplotlib>=1.3.1 + numpy>=1.8.1 + pandas>=0.14.0 + PyTango>=8.1.2 + wxmplot>=0.9.14 + wxpython>=3.0

1.4.4.1.1 Python 2.7, matplotlib, numpy and scipy Python 2.7 comes pre-installed, but just in case you can install it, together with numpy, pip and scipy, with:

```
sudo apt-get install -y python2.7 python-numpy python-scipy python-matplotlib
```

1.4.4.1.2 wxPython and wxmplot You can install wxPython following the wxPython Installation guide.

To install wxmplot just open a Terminal and write:

```
sudo easy_install -U wxmplot
```

1.4.4.1.3 Pandas and PyTango Pandas can be installed the normal way:

```
sudo pip install pandas
```

PyTango, however, needs an additional parameter:

```
sudo pip install PyTango --egg
```

1.5 User Manual

1.5.1 Configuration

First you need to download the latest version of the software from *TBD*. The file contains, bar the prerequisites, all needed to run the *HM*, the *AS*, including the simulated data, and the *HMGUI*. Once decompressed you need to (all paths are relative to the archive's root):

1. Configure each *HRM* instance Now you need to configure each *HRM* instance's connection string. To do it open each instance's configuration file (**src/hr_monitor.cfg**) and modify the *conn_str* variable as needed. A sample connection string is provided with the configuration file.
2. Configure Aouda Server's Tango Device Name in each *HRM*'s configuration file (**src/hr_monitor.cfg**); the variable you need to modify is *aouda_address*.
3. Configure the *HRM*'s Tango Device Name in the GUI configuration file (**src/gui/hr_monitor_gui.cfg**); the variable you need to modify is *monitor_address*.

1.5.2 Running the programs

TBD

5.5.2 2 Software Architecture Document for the Health Monitor

Author Mario Tambos

2.1 Change Record

2014.05.18 - Document created. 2014.08.20 - Updated document to reflect implementation.

2.2 Introduction

2.2.1 Purpose

Crew monitoring is an integral part of any manned mission. Since automated diagnosis tools are as yet not advanced enough, there is the problem of providing a human overseer with enough information to allow her to spot possible health problems as soon as possible.

Taking this into account this project is divided in three parts:

1. Investigation of consumer-level biomedic devices available in the market.
2. The implementation of a service to gather health metrics of all the crew. A priori this project will focus on the crew performing EVA, since, through their suits, their health information is readily available.
3. The implementation of a *GUI*, where the collected data shall be summarized and presented to an overseer for evaluation.

2.2.2 Scope

Describes the scope of this requirements specification.

2.2.3 Reference Documents

- [1] – C3 Prototype document v.4
- [2] – Software Engineering Practices Guidelines for the ERAS Project
- [3] – ERAS 2014 GSoC Strategic Plan

2.2.4 Glossary

ERAS European Mars Analog Station

GUI Graphic User Interface

IMS Italian Mars Society

EVA Extra-Vehicular Activity

TBD To Be Defined

TBC To Be confirmed

2.2.5 Overview

Make an overview in which you describe the rest of this document the and which chapter is primarily of interest for which reader.

2.3 Architectural Requirements

This section describes the requirements which are important for developing the software architecture.

2.3.1 Non-functional requirements

- As mentioned before, it should be possible to view the developed GUI from inside a virtual environment.
- The framework selected for the GUI development should be multiplatform. It should also have as few prerequisites as possible.
- **Biometric devices:**
  - Their cost should not exceed us$200.
  - They should be easy to integrate with TANGO.
  - Tey should not be cumbersome.
  - If possible each device should integrate several functions.
- The data collector should gather data from any device deemed relevant. Any relevant device not available should be simulated.

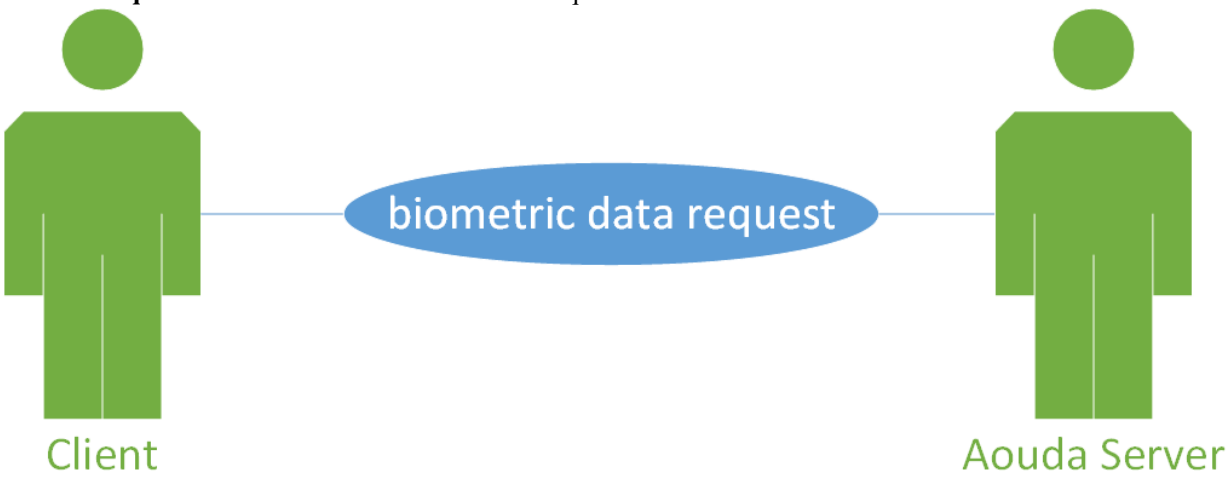
2.3.2 Use Case View (functional requirements)

The goal of this project is to build a service that allows the central monitoring of the crew's health. It will do this by collecting, processing and presenting data from the crew space suits. A data collector, implemented as a TANGO server, should gather the data from all available space suits and store it in a database. A GUI should then request the latest data from the database, summarize it and present it to an overseer in a way that allows him/her to detect problems at a glance. The GUI should also be able to be viewed from inside a VR-environment. At the moment a possible solution to this is to stream a computer screen/window to a surface texture in Blender, as shown in this video.

Additionally, it will be investigated what biometrics devices could be used in VR-simulations to monitor the crew participating in it. For selected devices a Tango server will be developed, from which then the collector will also gather data.

2.3.2.1 Request for biometric data

The Client request the Server the biometric data of the last T seconds.



2.3.2.1.1 Actors

Client: a TANGO client that makes the request. Server: the Aouda TANGO server.

2.3.2.1.2 Priority

High

2.3.2.1.3 Preconditions

The Server is running and its DevState is ON.

2.3.2.1.4 Basic Course

1. The Client calls the appropriate method on the Server, passing T as argument.
2. The Server searchs its buffer for the appropriate records.
3. The Server returns the records found.

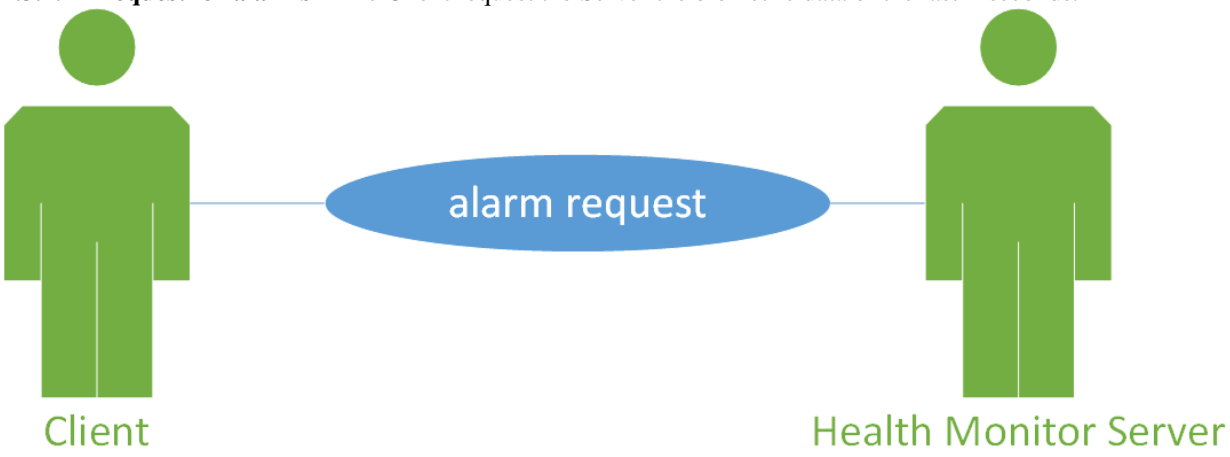
2.3.2.1.5 Alternate Course

None

2.3.2.1.6 Postconditions

The server returns the data requested or an empty array if no data is available.

2.3.2.2 Request for alarms The Client request the Server the biometric data of the last T seconds.



2.3.2.2.1 Actors Client: a TANGO client that makes the request. Server: the Health Monitor TANGO server.

2.3.2.2.2 Priority High

2.3.2.2.3 Preconditions The Server is running and its DevState is ON.

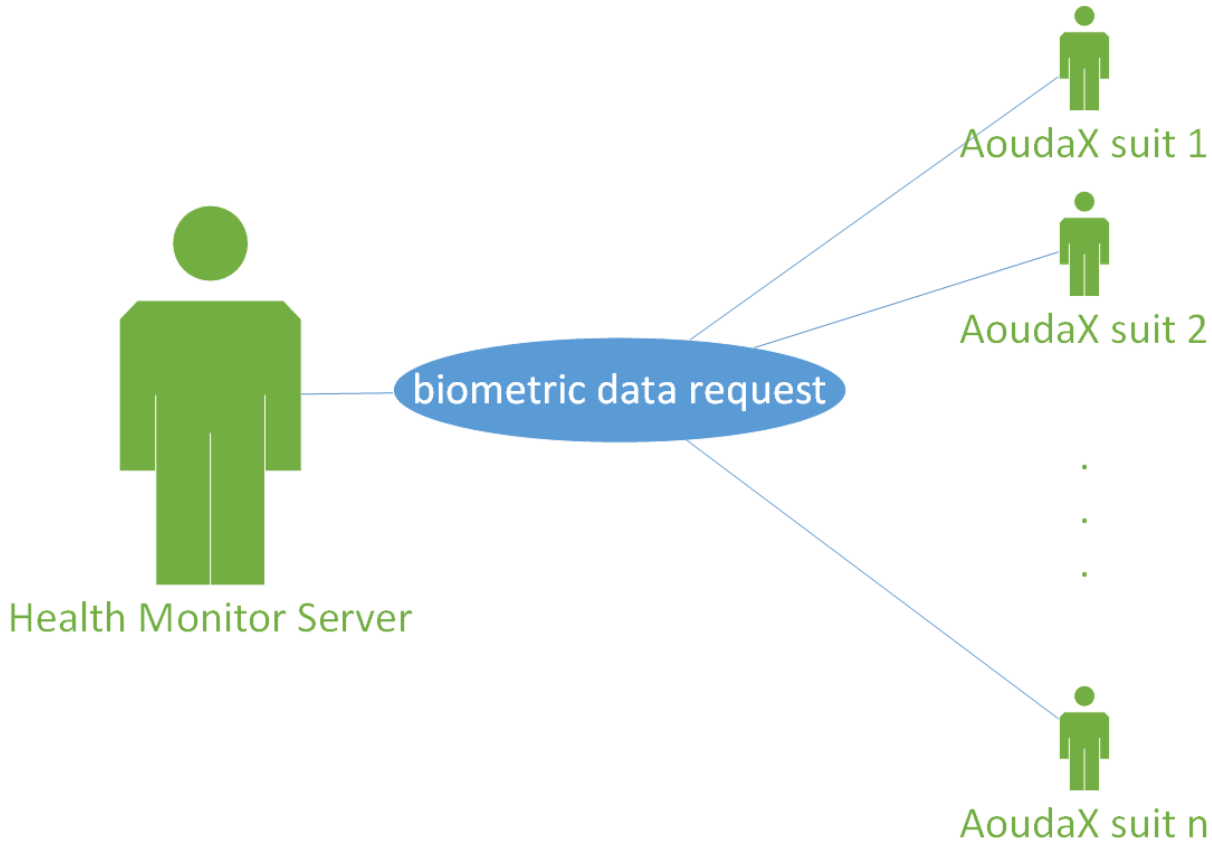
2.3.2.2.4 Basic Course

1. The Client calls the appropriate method on the Server, passing T as argument.
2. The Server searchs the database for the appropriate records.
3. The Server returns the records found.

2.3.2.2.5 Alternate Course None

2.3.2.2.6 Postconditions The server returns the data requested or an empty array if no data is available.

2.3.2.3 Server requests new data The Server reads the data of each available Aouda suit from the Framework Software Bus, ands stores it in the database.



2.3.2.3.1 Actors Server: the Health Monitor TANGO server. Aouda Server: Tango server that provides the Aouda Suit simmulated data.

2.3.2.3.2 Priority High

2.3.2.3.3 Preconditions The Server is running and its DevState is ON.

2.3.2.3.4 Basic Course

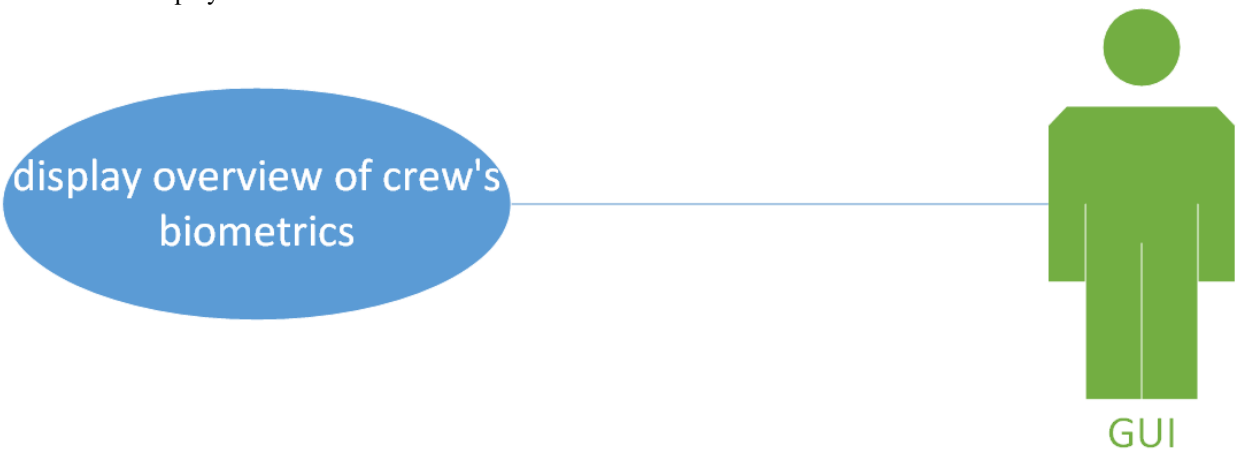
1. The Server request new data from each available Aouda Server.
2. The Aouda Servers returns the data available.
3. The Server stores the data of each suit.

2.3.2.3.5 Alternate Course None

2.3.2.3.6 Exception Course None

2.3.2.3.7 Postconditions The data from the suits is stored in the database.

2.3.2.4 The GUI shows overview of crew's biometrics The GUI gets all data from the previos T seconds, summarizes it and displays it.



2.3.2.4.1 Actors GUI: a GUI with an embedded TANGO client. Server: the Health Monitor TANGO server.

2.3.2.4.2 Priority High

2.3.2.4.3 Preconditions The Server is running and its DevState is ON.

2.3.2.4.4 Basic Course

1. The GUI calls the appropriate method on the Server, passing T as argument.
2. The Server searchs the database for the appropriate records.
3. The Server returns the records found.
4. For each available suit *s*:
   (a) The GUI calls the appropriate method on itself, in order to summarize the biometrics of *s*.
   (b) The GUI calls the appropriate method on itself, in order to display the summarized biometrics of *s*.

2.3.2.4.5 Alternate Course None

2.3.2.4.6 Exception Course None

2.3.2.4.7 Postconditions The crew biometric's overview is shown on the GUI.

2.3.2.5 A User requests a crewmember's detailed biometrics A user requests the detailed biometrics for a given crewmember and the GUI complies.

2.3.2.5.1 Actors User: a user of the GUI. GUI: a GUI with an embedded TANGO client.

2.3.2.5.2 Priority High

2.3.2.5.3 Preconditions The Server is running and its DevState is ON.

2.3.2.5.4 Basic Course

1. The User clicks on the icon of crewmember *c*.
2. The GUI hides the summarized view for *c*.
3. The GUI shows the detailed view for *c*.

2.3.2.5.5 Alternate Course None

2.3.2.5.6 Exception Course None

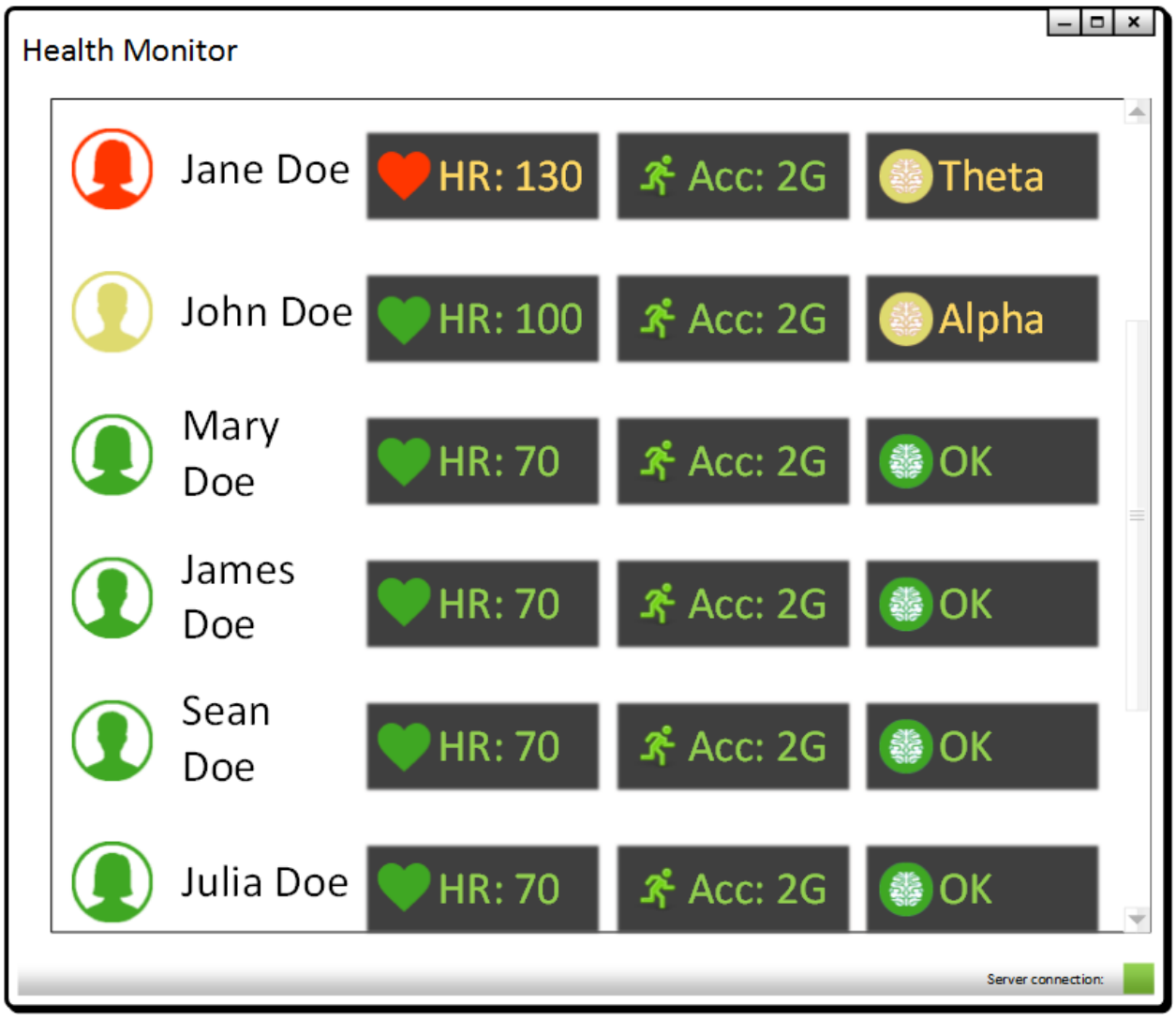
2.3.2.5.7 Postconditions The selected crewmember biometric's detailed view is shown on the GUI.

2.4 Interface Requirements

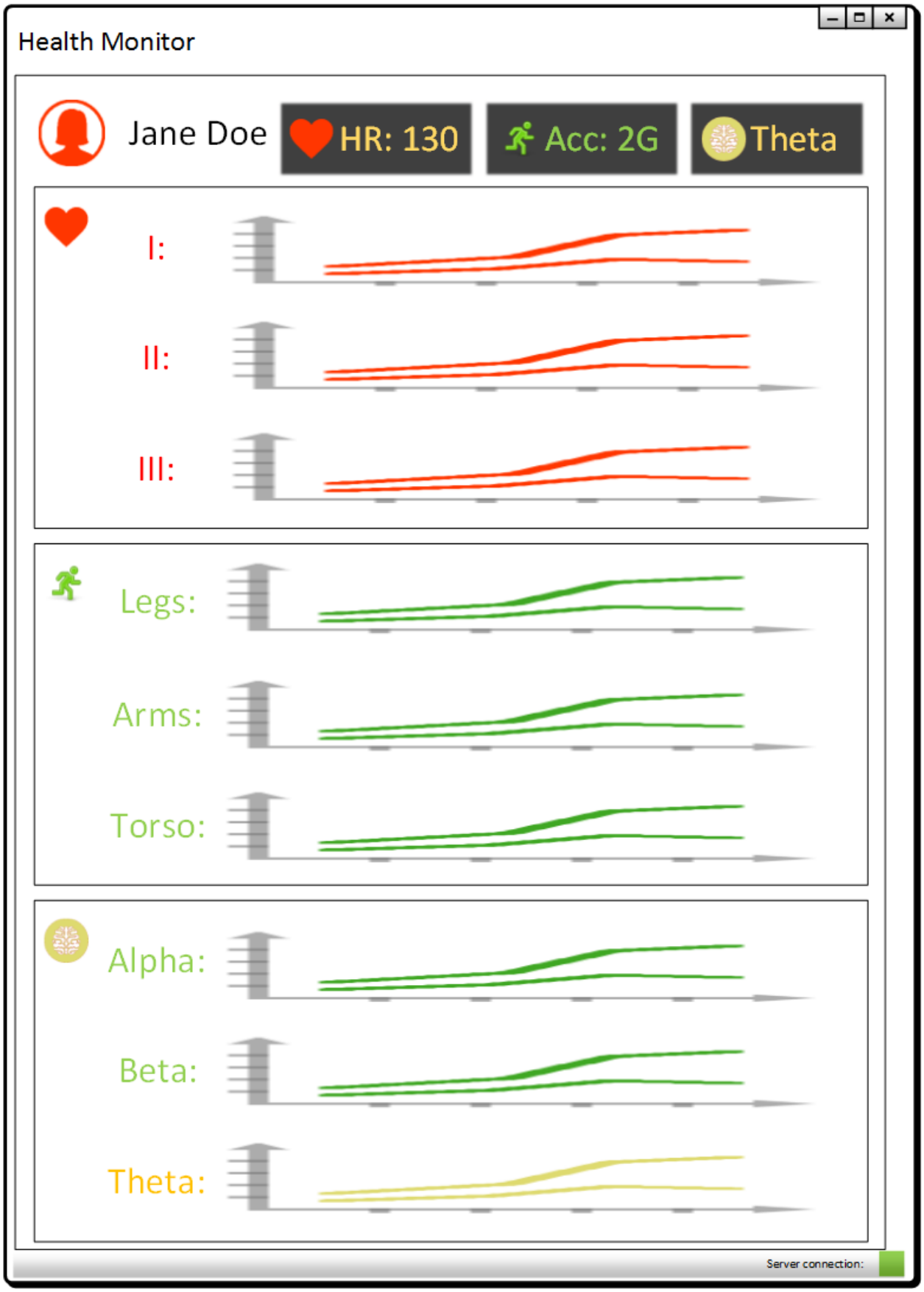
2.4.1 User Interfaces

2.4.1.1 GUI (Graphical User Interface) The GUI should use this interface to take information from the data collector, avoiding direct acces to the data storage.

Bellow are two mockups that cover the two current Use Cases that concern the GUI.



2.4.1.1.1 Overview


Health Monitor
Jane Doe
HR: 130
Acc: 2G
Theta
I:
II:
III:
Legs:
Arms:
Torso:
Alpha:
Beta:
Theta:
Server connection:

2.4.1.1.2 Detailed View

2.4.1.2 API (Application Programming Interface) *TBD*

2.4.2 Hardware Interfaces

Here should be referenced the hardware interfaces of the biometric devices. Specifics are *TBD*.

2.4.3 Software Interfaces

The data collector module will be implemented as a Python TANGO server, which will expose methods to request raw as well as summarized data.

2.5 Performance Requirements

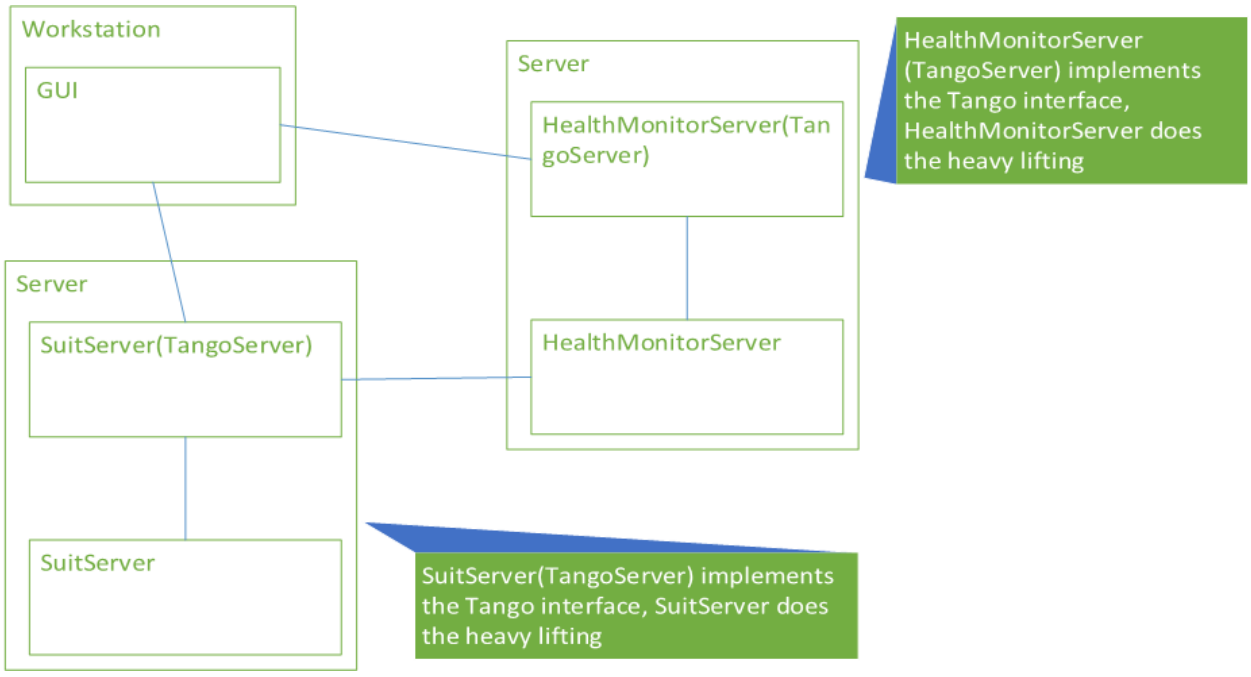
The software should allow the monitoring of health metrics in real-time, therefore any preparation of the data should be quick enough as to be non-noticeable.

Furthermore, the suit simulator should be able to run in a RasperryPi or similar.

Specifics are *TBD*.

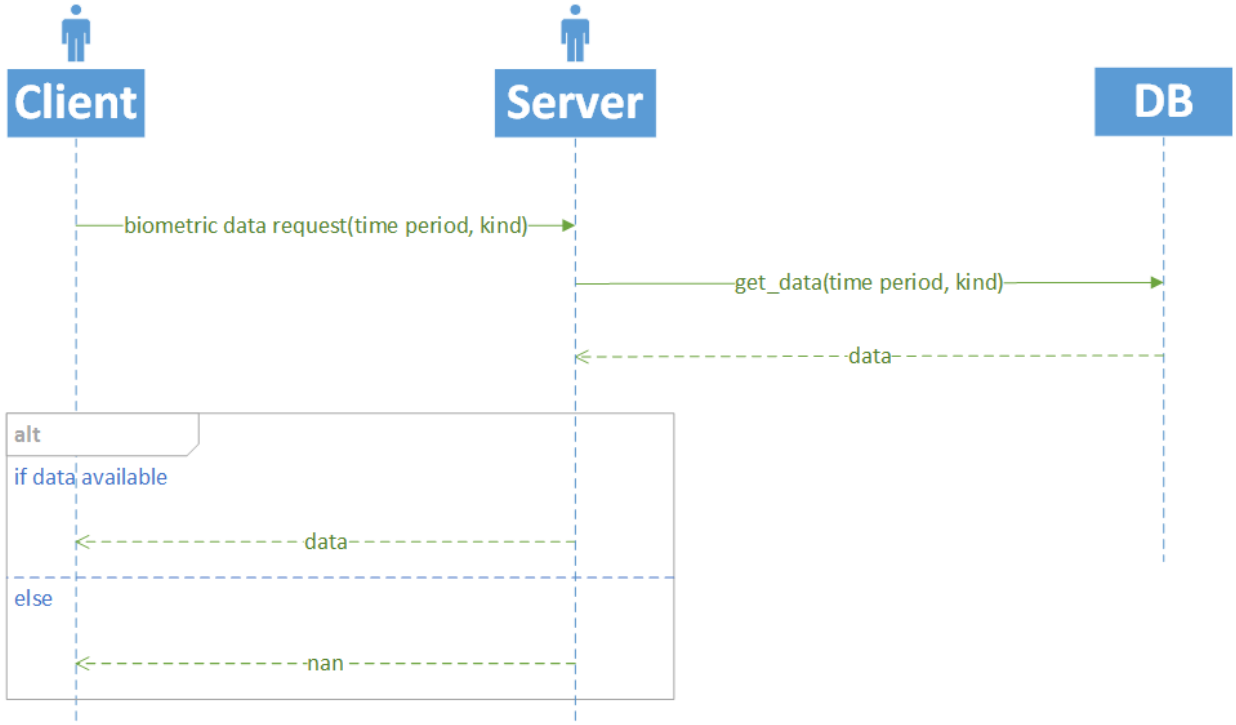
2.6 Logical View

2.6.1 Layers & Subsystems



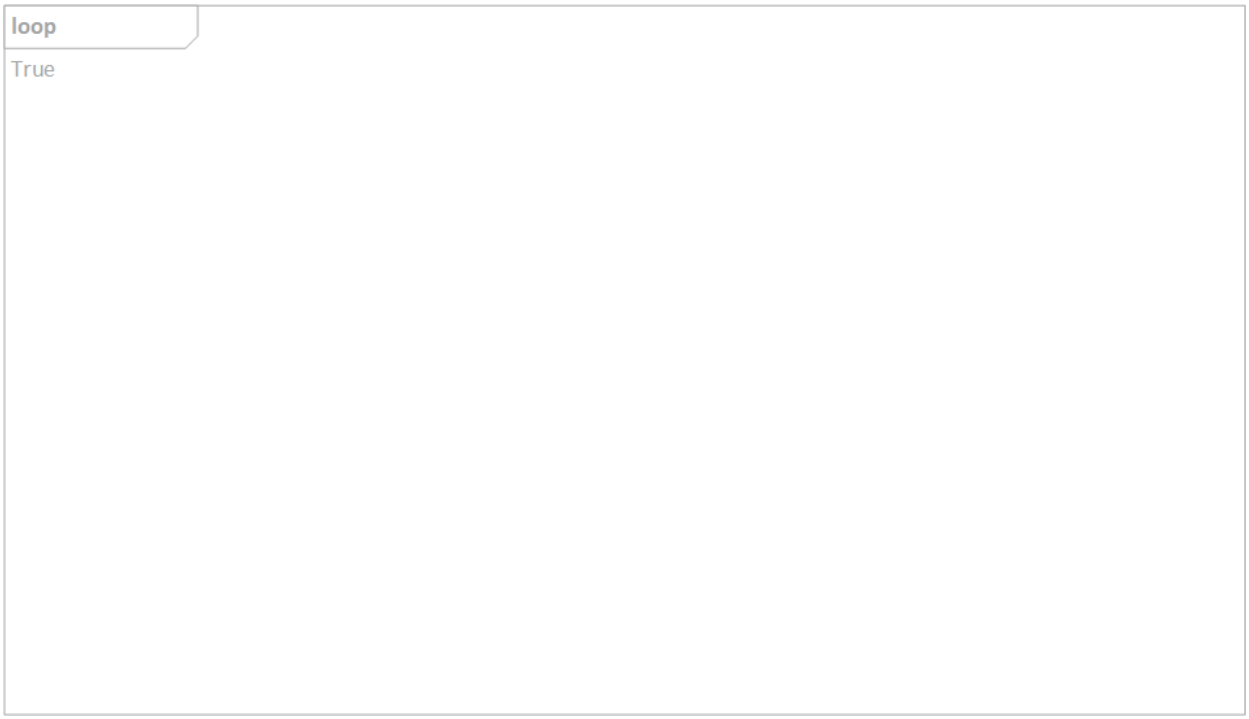
2.6.2 Use Case Realizations

2.6.2.1 Request for biometric data



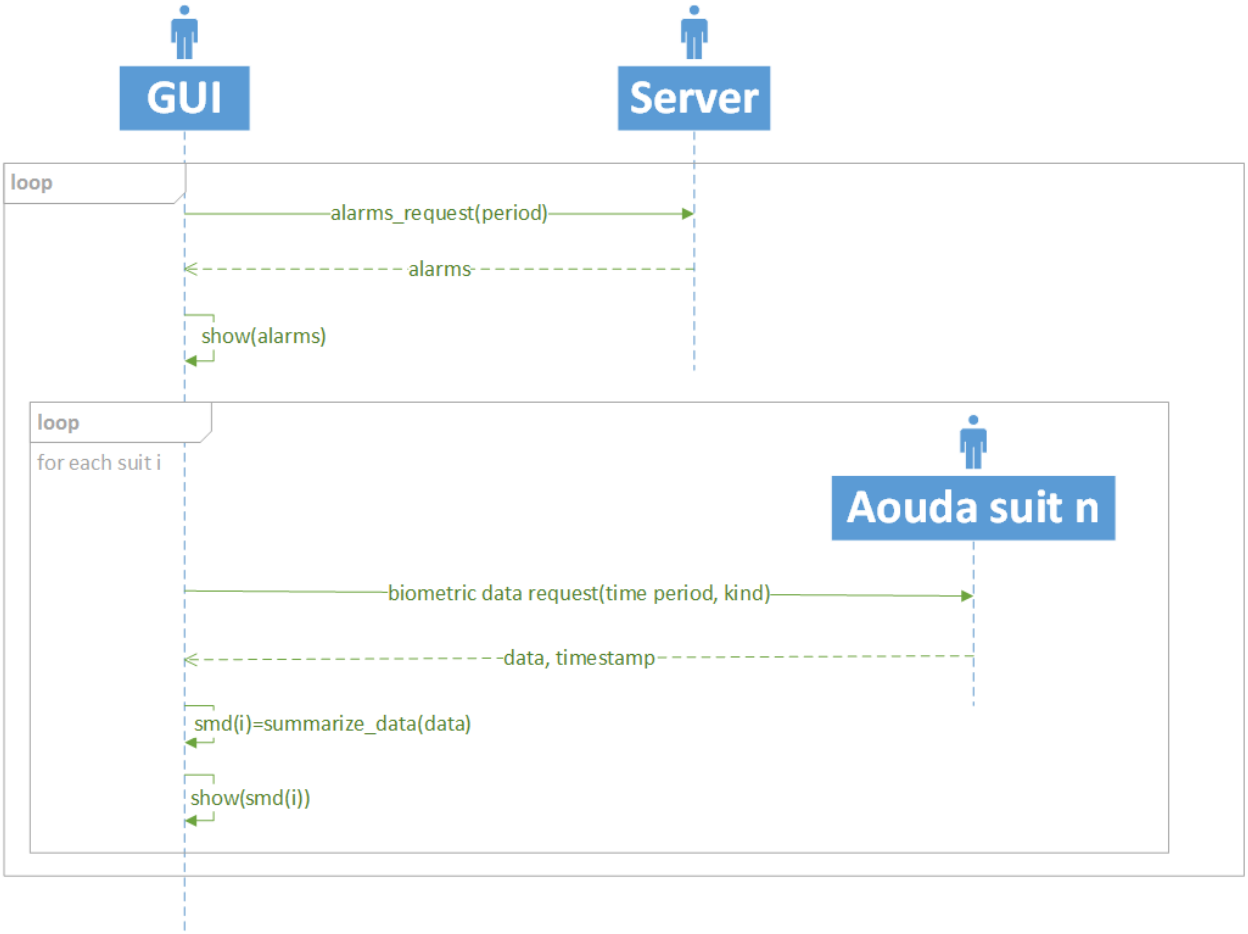
2.6.2.1.1 Sequence diagram

2.6.2.2 Server requests new data



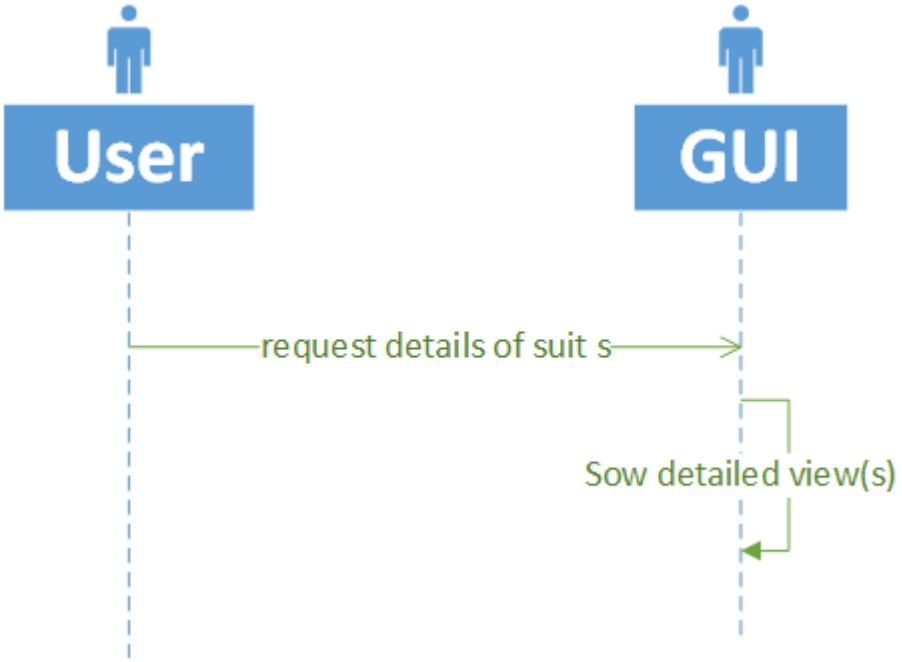
2.6.2.2.1 Sequence diagram

2.6.2.3 The GUI shows overview of crew's biometrics



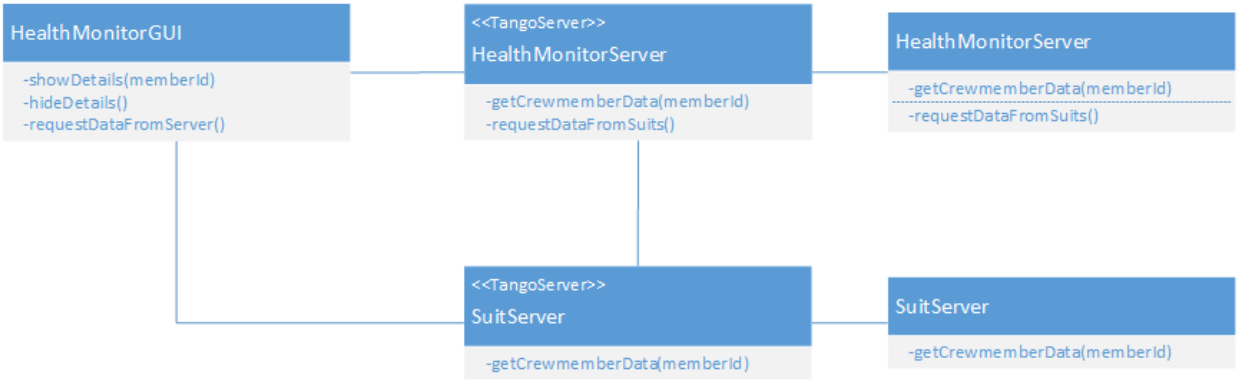
2.6.2.3.1 Sequence diagram

2.6.2.4 A User requests a crewmember's detailed biometrics



2.6.2.4.1 Sequence diagram

2.7 Implementation View



2.8 Deployment View

TBD

2.9 Development and Test Factors

2.9.1 Standards Compliance

The guidelines defined in [2] should be followed.

2.9.2 Planning

The high-level schedule is defined in [3], with deliverables as follows:

- A TANGO server that implements the data collector.

- A GUI that presents summarized and detailed data of the crew's biometrics.
- A document describing the biometric devices selected for the project.
- A space suit simulator that integrates the real devices.
- Testing
  - Test environment to help diagnose the server's work.
  - A set of integration tests between the collector and the GUI.
  - A set of interface tests for the GUI.
- Documentation.
  - User requirements (this document).
  - Design Study document.
  - User Manual.

The time for this particular project is devided as follows:

1. Refine schedule with mentor 1. Discuss architecture depth. 2. Account for side requirements. 3. Define nice-to-haves. 4. Draft User requirements Document.	1/2 week
2. Find appropriate biometric device(s)	1 weeks
3. Build space suite simulator 1. Gather data (ECG, EEG, accelerometer, heart rate). 2. Build randomizer. 3. Integrate with real devices	3 weeks
4. Build data collector 1. Define database. 2. Define configuration variables. 3. Write Unit tests	3 weeks
5. Build GUI 1. Define basic interface. 2. Build XML-based interface customizer. 3. Write interface tests.	2 weeks
6. Write integration tests	1 week
7. Finish writing documents	1 week
8. Buffer time 1. Cover for unforeseeables. 2. Implement nice-to-haves. 3. Improve code readability. 4. Improve documents readability.	

5.6 Heart Rate Monitor Server

5.6.1 1 Heart Rate Monitor Software User and Maintenance Manual

Author Mario Tambos

1.1 Change Record

2013.09.11 - Document created.

2013.09.17 - Update format with user manual template and corresponding content.

1.2 Introduction

1.2.1 Purpose

This document describes the installation, use and maintenance of the Heart Rate Monitor.

1.2.2 Applicable Documents

- [1] – C3 Prototype document v.4
- [2] – PAMAP2 Physical Activity Monitoring
- [3] – Software Engineering Practices Guidelines for the ERAS Project
- [4] – ERAS 2013 GSoC Strategic Plan
- [5] – Software Requirements Specification for the Heart Rate Monitor
- [6] – Software Design Study for the Heart Rate Monitor

- [7] – TANGO distributed control system
- [8] – PyTANGO - Python bindings for TANGO
- [9] – Tango Setup
- [10] – wxPython Installation
- [11] – Adding a new Server in Tango
- [12] – Wei, Li, et al. "Assumption-Free Anomaly Detection in Time Series." SSDBM. Vol. 5. 2005. APA

1.2.3 Glossary

AD Anomaly Detection

API Application Programming Interface

ERAS European Mars Analog Station

GUI Graphic User Interface

HRM Heart Rate Monitor

IMS Italian Mars Society

TBC To Be Confirmed

TBD To Be Defined

1.3 Overview

1.3.1 Hardware Architecture

The different hardware components that need to be taken into account are shown in the Deployment Diagram below. As done at the moment all software components can be run in a single computer, they can however also be run each in a different machine. One key assumption is that one instance of the *HRM* will monitor one single Suit. In other words, one instance of the *HRM* is needed for each crew member during EVA. This instances can be hosted in the same computer or in different ones, they can also use different MySql instances or the same one. The only caveat is that in the later case (shared MySql server instance), different databases MUST be used.

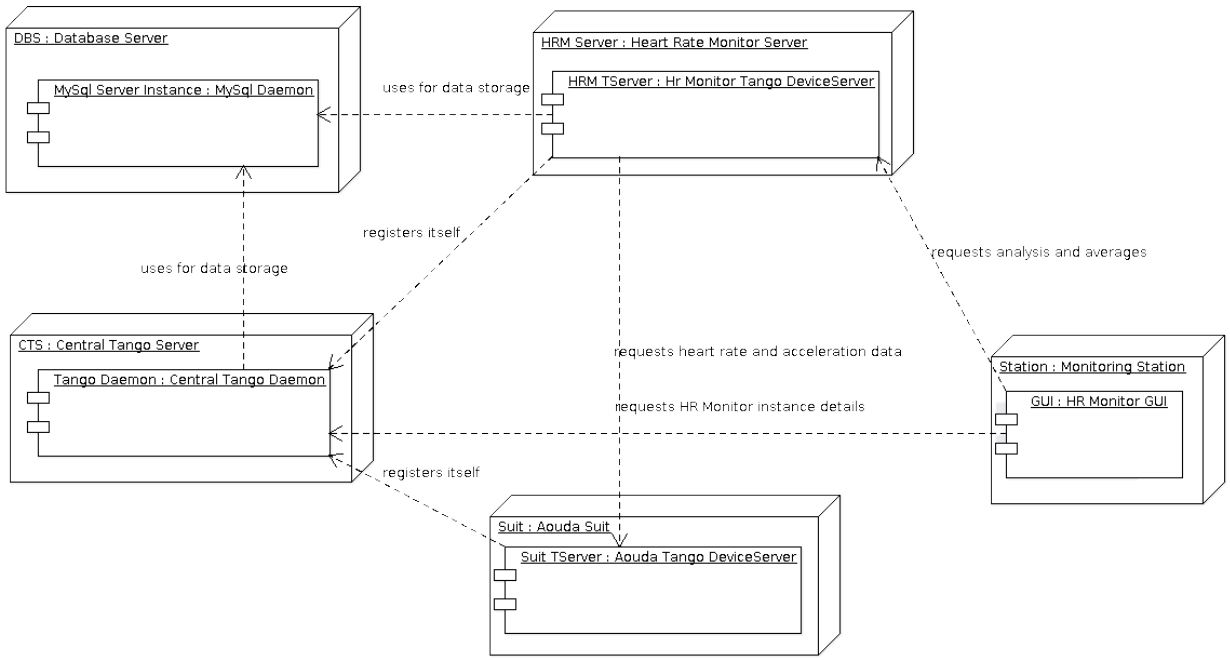
1.3.2 Software Architecture

The components involved can be divided in five categories:

1. The central Tango Daemon: It keeps track of the existing Tango Device Servers. For details refer to [7] and [8].
2. The MySql Server instance (or instances): At least one is used by the Tango Daemon. The *HRM* also needs an instance, that can be the same one used by the Tango Daemon. Each instance of the *HRM* needs a different database to store its data, that means having one database on one MySql Server instance for each *HRM* instance, or using the same MySql Server instance for all *HRM* instances, but creating and configuring different databases. For details see the *Installation Guide*
3. The Aouda Suit Device Server: Currently this component only simmulates the Suit taking data from one of the datasets in [2]. Provides data to the *HRM*.
4. The Heart Rate Monitor Device Server: The core of the project.

5. The Heart Rate Monitor Graphic User Interface: Just a proof of concept at the moment. It allow the user to see some statistics provided by the *HRM*. As it is, the *GUI* can only connect to a single *HRM* instance at a time, so one needs to start as many *GUI*s as *HRM* instances one wants to oversee.

1.3.3 Deployment Diagram



1.4 Installation Guide

1.4.1 Prerequisites

- Python 2.7
- Tango Controls v7.2.2
- MySql Server
- **Python modules:**
  - numpy
  - scipy
  - pandas
  - matplotlib
  - PyTango v7.2.2
  - sqlalchemy
  - wxpython
  - wxmplot

1.4.1.1 Installing Prerequisites in Ubuntu System

1.4.1.1.1 Python 2.7 Python 2.7 comes pre-installed, but just in case you can install it with:

```
sudo apt-get install python2.7
```

1.4.1.1.2 Tango Controls v7.2.2, PyTango and MySql You can install these three components following the Tango Setup guide.

Besides that, the HR Monitor just needs the MySql-Python connector:

```
sudo apt-get install python-mysqldb
```

The first line will open the MySql console, the second will create the database and the third will show you the existing database, to confirm everythong is OK.

1.4.1.1.3 Numpy, Scipy, Pandas and Matplotlib Since these three modules rely on C libraries, it's recommended to install them using apt-get instead of easy_install/pip. They should have been installed during the PyTango installation, but if not:

```
sudo apt-get install python-numpy python-scipy python-matplotlib
sudo pip install pandas
```

1.4.1.1.4 SqlAlchemy You can install it from PyPi with:

```
sudo apt-get install python-pip
sudo pip install SQLAlchemy
```

1.4.1.1.5 wxPython and wxmplot You can install wxPython following the wxPython Installation guide.

To install wxmplot just open a Terminal and write:

```
sudo easy_install -U wxmplot
```

1.4.2 Heart Rate Monitor Installation

First you need to download the latest version of the software from *TBD*. The file contains, bar the prerequisites, all needed to run the *HRM*, the Aouda Server, including the simmulated data, and the *HRM GUI*. Once decompressed you need to (all paths are relative to the archive's root):

1. Create a database for each *HRM* you want to run: You can do this by opening a shell in the computer you have your MySql Server instance running and typing the following for each instance, which will create a database and show the list of databases for confirmation.

```
mysql -u [user] -p[password]
> create database [HRM instance name];
> show databases;
```

2. Configure each *HRM* instance Now you need to configure each *HRM* instance's connection string. To do it open each instance's configuration file (**src/hr_monitor.cfg**) and modify the *conn_str* variable as needed. A sample connection string is provided with the configuration file.
3. Register both the Aouda and HR Monitor Tango Servers: To do it, just follow Adding a new Server in Tango. In both cases the class name is 'PyDsExp', without quotation marks.
4. Configure Aouda Server's Tango Device Name in each *HRM*'s configuration file (**src/hr_monitor.cfg**); the variable you need to modify is *aouda_address*.

5. Configure the *HRM*'s Tango Device Name in the GUI configuration file (**src/gui/hr_monitor_gui.cfg**); the variable you need to modify is *monitor_address*.

Once all this is done, all is in place to start running the programs.

1.5 User Manual

1.5.1 Application Start-up

In the following we will assume a single instance of the *HRM*, Aouda Device Server and *HRM GUI*. For more instances just repeat the steps for each instance.

First of all, you'll have to run the Aouda Tango Server. This server will simmulate the Aouda Suit, making data available for the *HRM* instance to consume. To do this, just open a Terminal and type:

```
cd /path/to/hr_monitor/src
python aouda [instance name]
```

Once done, you can start the *HRM* itself with:

```
python hr_monitor [instance name]
```

The simmulation has approximately 45 minutes of data. After that time it willll start repeating the data from the beginning.

Now if you want to see the alarm levels, you can do it by starting the *HRM GUI*. To do it type the following in a Terminal:

```
export TANGO_HOST=[IP:Port of the Tango central server]
cd /path/to/hr_monitor/src/gui
python app.py
```

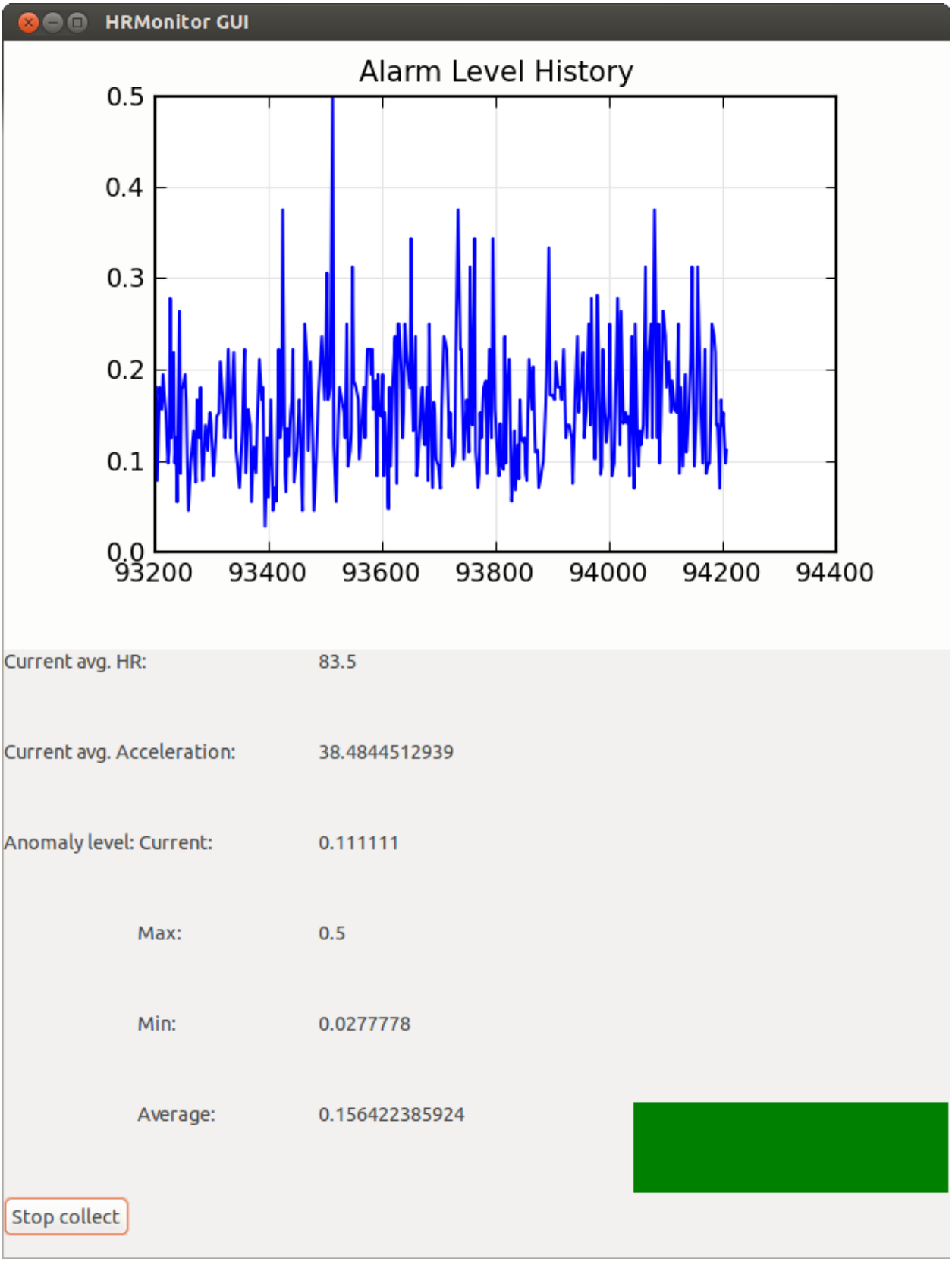
To start collecting data from the *HRM* instance, press the "Collect data" button.

1.5.2 Application Shut-down

The *HRM GUI* can be shut down with its window's close button (x). Both *HRM* and Aouda Device Server must be shut down with the key combination *Ctrl+C*. There's a pending bug that causes the *HRM* to need two presses of *Ctrl+C*.

1.5.3 GUIs

The *HRM GUI* is shown in the following image:



The top panel provides a history of the alarm level values. The left panel shows different statistics over those values as

also over the heart rate and acceleration values. The right panel shows during operation a color-coded severity of the latest alarms; green for low, yelow for medium and red for high severity. You can adjust the thresholds for the colors from the configuration file (**src/gui/hr_monitor_gui.cfg**), the variables you need to modify are *yellow_alrm_thrsh* and *red_alrm_thrsh*. From the same configuration file you can also configure how often the *HRM GUI* must poll its associated *HRM* instance, by modifying the *sleep_time* variable, in seconds.

1.6 Maintenance Manual

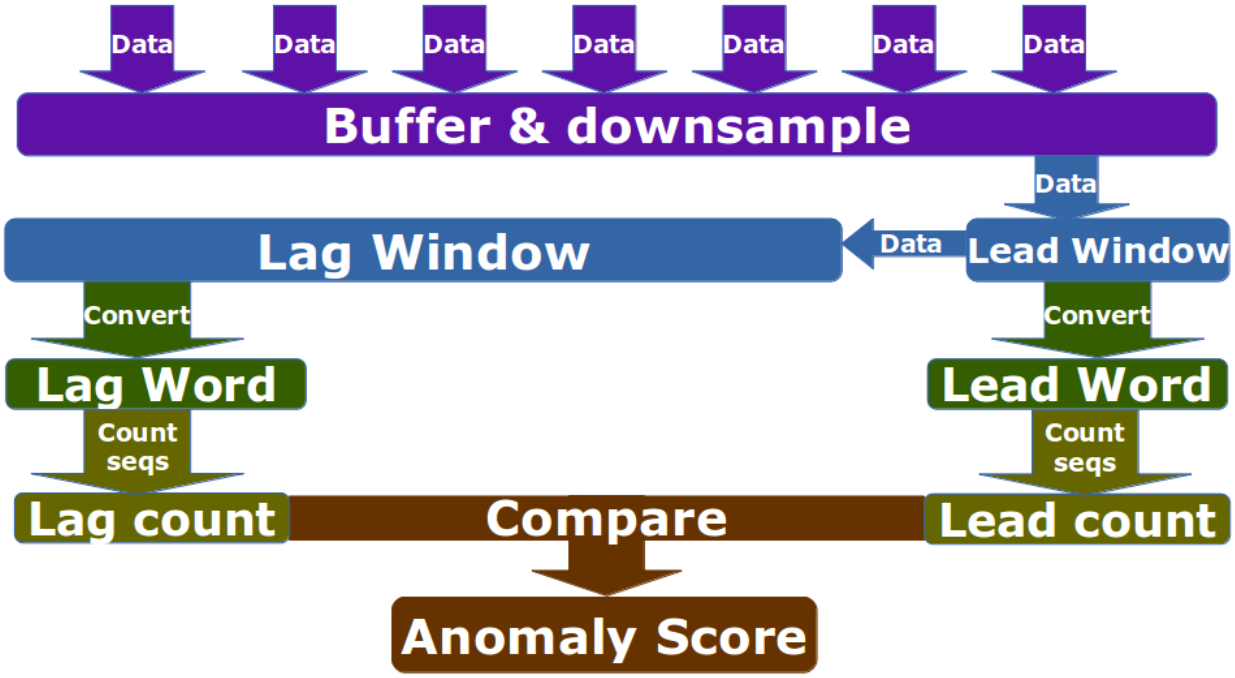
For maintenance purposes it is higly recomended to first read [5] and [6].

1.6.1 Internals

In short the anomaly detection progresses in the following steps:

1. We combine heart rate and acceleration into a single feature: *HA = [heart rate]/[acceleration's magnitude]* Therefore our anomalies will be marked with a HA value very close to 0 or very big, when the heart is beating too slow/quickly.
2. We downsample to *[resolution]* seconds. So datapoints occurring at a higher frequency become a single one (their mean). It bears mentioning that by doing this we are not losing a lot of information.
3. Then we follow [12]:
   (a) Divide data into a lead window (current situation) of size *[word_size] * [window_factor] * [lead_window_factor]*, which collects the new datapoints, and a lag window (historical trend) of size *[word_size] * [window_factor] * [lag_window_factor]*, which collects the data overflowed from the lead window.
   (b) Convert each window into a single *[word_size] * [window_factor]* length word, using a four char alphabet (using SAX algorithm).
   (c) Count the 2 char combination's frequencies.
   (d) Use as anomaly score the differences in the counts between the lead and lag window's words.

A schematic diagram of the preceding steps is shown in the following figure:



The *[variables]* are explained in some better detail in the *Additional Configuration* section.

1.6.2 Application Program Interface (API)

The *HRM GUI* does not have any programmable interface. The Aouda Device Server publishes one Tango Command and two Tango Attributes:

1. DevVarDoubleStringA get_data(DevLong period): Returns an array with all datapoints from last query untilthe current date and time. The format is:

```
[
   [hr1, acc_x1, acc_y1, acc_z1,
    hr2, acc_x2, acc_y2, acc_z2,
    ...
    hrN, acc_xN, acc_yN, acc_zN],
   [timestamp1, timestamp1, timestamp1, timestamp1,
    timestamp2, timestamp2, timestamp2, timestamp2,
    ...
    timestampN, timestampN, timestampN, timestampN],
]
```

2. DevFloat heart_rate - scalar, read-only: Returns the instantaneous heart rate. The underlying device's sampling frequency is 9Hz.
3. DevFloat acc_magn (scalar, read-only): Returns the instantaneous acceleration vector's magnitude. The underlying device is an accelerometer placed on the torso, with a +-16g range, 13 bit precission and 100Hz sampling frequency.

The *HRM* publishes four Tango Commands:

1. DevVoid register_datapoint(DevVarDoubleArray datapoint): Registers a new datapoint in the database and launches a new thread to analyze the data collected so far. datapoint must have the format [timestamp, hr, acc_x, acc_y, acc_z].
2. DevFloat get_avg_hr(DevLong period): Returns average heart rate over the given [period] (in seconds).

3. DevFloat get_avg_acc(DevLong period) Returns average acceleration over the given [period] (in seconds).
4. DevVarDoubleStringArray get_current_alarms(DevLong period): Returns the alarm scores generated in the last [period] seconds. The return value format is [[alarm_lvl1, ..., alarm_lvlN], [timestamp1, ..., timestampN]]

Additionally, uppon start up the *HRM* launches a thread that polls the Aouda Server periodically, and then calls the register_datapoint method to store the data returned. You can adjust the polling period by changing the *polling_frequency* variable from the configuration file (**src/hr_monitor.cfg**), in seconds.

1.6.3 Additional Configuration

The *HRM*'s configuration file (**src/hr_monitor.cfg**) exposes, aside the values explained above, the following variables:

1. [Detector]:
   (a) word_size: It affects the length of features used to detect anomalies.
   (b) window_factor: Also affects the length of the features, each feature comprises window_factor * word_size datapoints.
   (c) lead_window_factor Affects the length of the lead window, which consists of window_factor * word_size datapoints * lead_window_factor datapoints.
   (d) lag_window_factor Affects the length of the lag window, which consists of window_factor * word_size datapoints * lag_window_factor datapoints.
2. [Monitor]:
   (a) resolution: In milliseconds, it affects the data's sample frequency to be considered. If finer grained data is available it is down sampled to [resolution] ms frequency.

To better understand what the different variables mean, please refer to the *Internals* section. All of the above are set with values that should provide good analysis results. However, as this is a highly experimental field, you should feel free to experiment with them in order to make the most of this tool.

1.7 Troubleshooting

Since this is the first version of the product, it is difficult to foresee what problems could be encountered during the execution of the project's components. The previous guides have been written trying to forestall any difficulties during the initial set up and running. If you happen to encounter problems, first please double check that all prerequisites are met, and that you follow to the letter all instructions; in the ideal case starting with a fresh installation. Were the problem not resolved, or were you left with unaswered questions, please don't hessitate to contact us.

5.6.2 2 Software Requirements Specification for the Heart Rate Monitor

Author Mario Tambos

2.1 Change Record

2013.06.18 - Document created.

2013.06.26 - Fixed typos and formatting.

2013.06.26 - Removed unused sections.

2013.06.26 - Added Use Cases and Sequence diagrams.

2013.09.18 - Updated to reflect current requirements. Some formatting changes.

2.2 Introduction

2.2.1 Scope

This document describes the top level requirements for the Heart Rate Monitor module, which in turn is part of the Crew Mission Assistant system.

2.2.2 Reference Documents

- [1] – C3 Prototype document v.4
- [2] – PAMAP2 Physical Activity Monitoring
- [3] – Software Engineering Practices Guidelines for the ERAS Project
- [4] – ERAS 2013 GSoC Strategic Plan
- [5] – Wei, Li, et al. “Assumption-Free Anomaly Detection in Time Series.” SSDBM. Vol. 5. 2005. APA

2.2.3 Glossary

ERAS European Mars Analog Station

IMS Italian Mars Society

EVA Extra-Vehicular Activity

TBD To Be Defined

TBC To Be confirmed

2.3 General Description

2.3.1 Problem Statement

Monitoring of the crew's health is a critical part of any mission. Said monitoring includes, among other things, overseeing the heart rate of the crew members, in order to take preventive actions if it takes on abnormal values. However, this task can prove difficult to accomplish, due to the fact that a persons heart rate not only hasn't a constant "normal" value, but also depends on a number of factors, for instance the level of physical stress that the person endures at the moment. In [1] it is stated that the current approach to solve this problem is the use of a Feed-Forward Neural Network to predict the heart rate from the body acceleration reported by the Aouda.X suit. This can prove troublesome, given that regression techniques have difficulties achieving high recall when the training set is skew, as in this case (there are going to be far more normal than abnormal datapoints).

I believe using Anomaly Detection techniques would prove a more reliable method of alert for this use, because these techniques are specifically designed for the purpose, i.e. are not affected by imbalances in the cardinality of the datapoint classes.

2.3.2 Functional Description

The goal of this module is to oversee the heart rate and physical activity of each crew member performing EVA, in order to raise alarms if the reported crew member's heart rate is abnormal for his or her current physical stress level.

2.3.3 Constraints

As described in [1], the available heart rate and accelerometer data from the Aouda.X suit is too unreliable to be of any use for this module prototype. Therefore the data found in [2] will be used instead as a way to simulate a reliable data stream from the suit.

2.4 Interface Requirements

2.4.1 Software Interfaces

2.4.1.1 Communication Interfaces This module will be implemented as a Python TANGO server, which will expose methods to request the heart rate, level of physical activity and heart rate alarms, if they exist. Moreover the alarms should be optionally declared as events, to enable push request from the server to the clients.

2.5 Development and Test Factors

2.5.1 Standards Compliance

The guidelines defined in [3] should be followed.

2.5.2 Planning

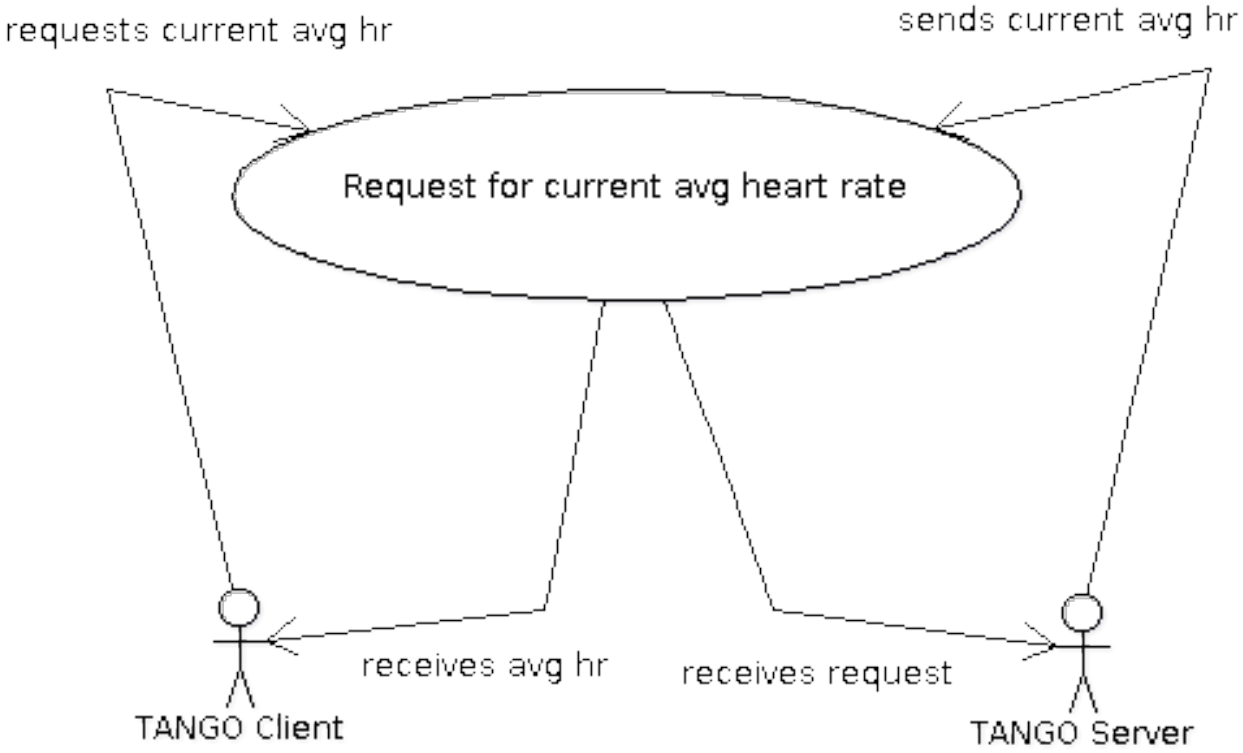
The schedule is as defined in [4], with deliverables as follows:

- TANGO server that implements the anomaly detector.
- **Test environment to help diagnose the server's accuracy.**
  - Train and test data sets.
- **Documentation.**
  - User requirements (this document).
  - Design Study document.
  - User Manual.

2.6 Use-Cases

2.6.1 Request for current average Heart Rate

The Client request the Server the average Heart Rate over the last T seconds.



2.6.1.1 Actors Client: a TANGO client that makes the request. Server: the Heart Rate Monitor TANGO server.

2.6.1.2 Priority Normal

2.6.1.3 Preconditions The Server is running and its DevState is ON.

2.6.1.4 Basic Course

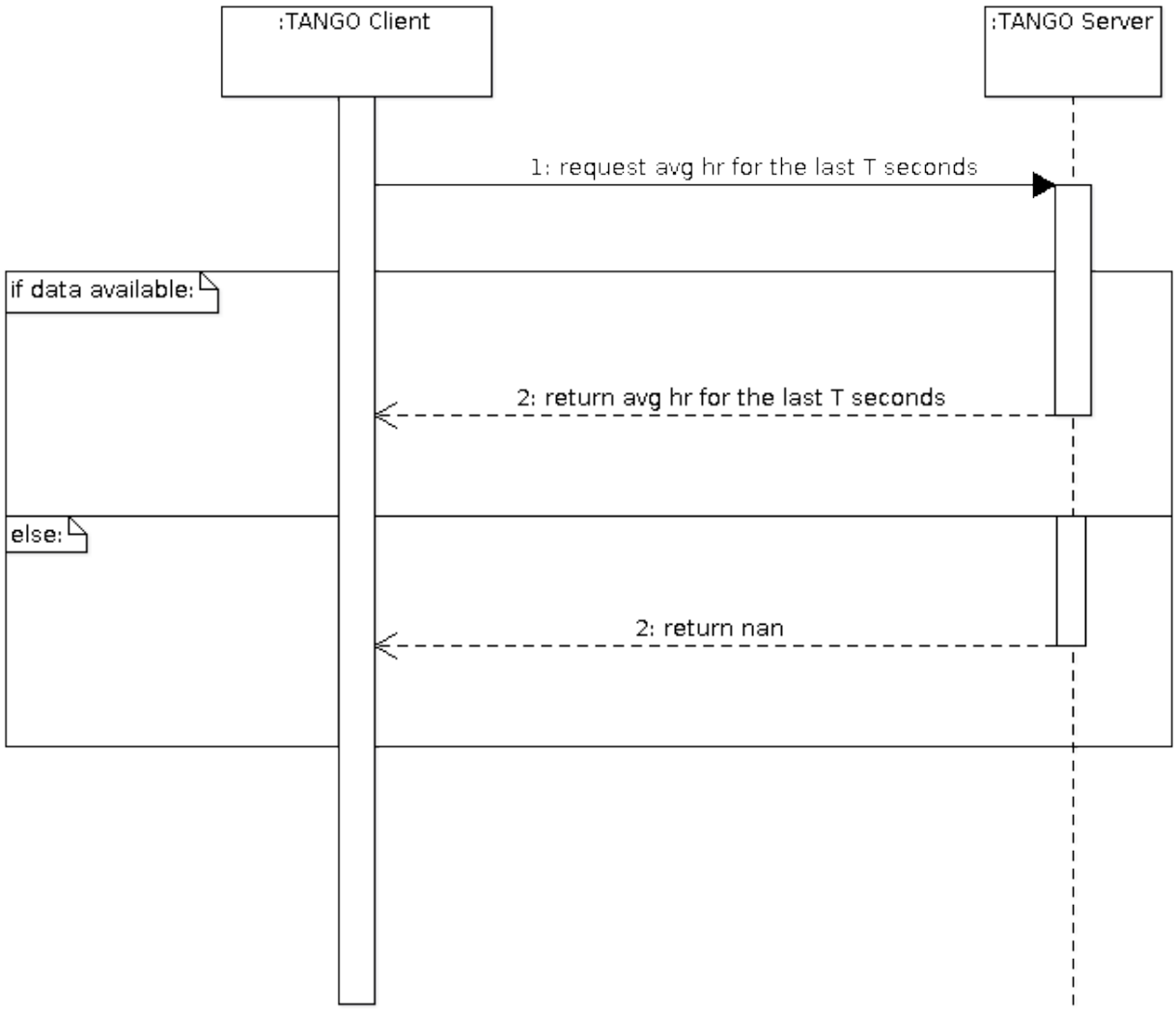
1. The Client calls the appropriate method on the Server, passing T as argument.
2. The Server calculates the average heart rate over the last T seconds.
3. The Server returns the calculated value.

2.6.1.5 Alternate Course None

2.6.1.6 Exception Course

1. The Client calls the appropriate method on the Server, passing T as argument.
2. The Server tries calculates the average heart rate.
3. No data is available.
4. The Server returns an error.

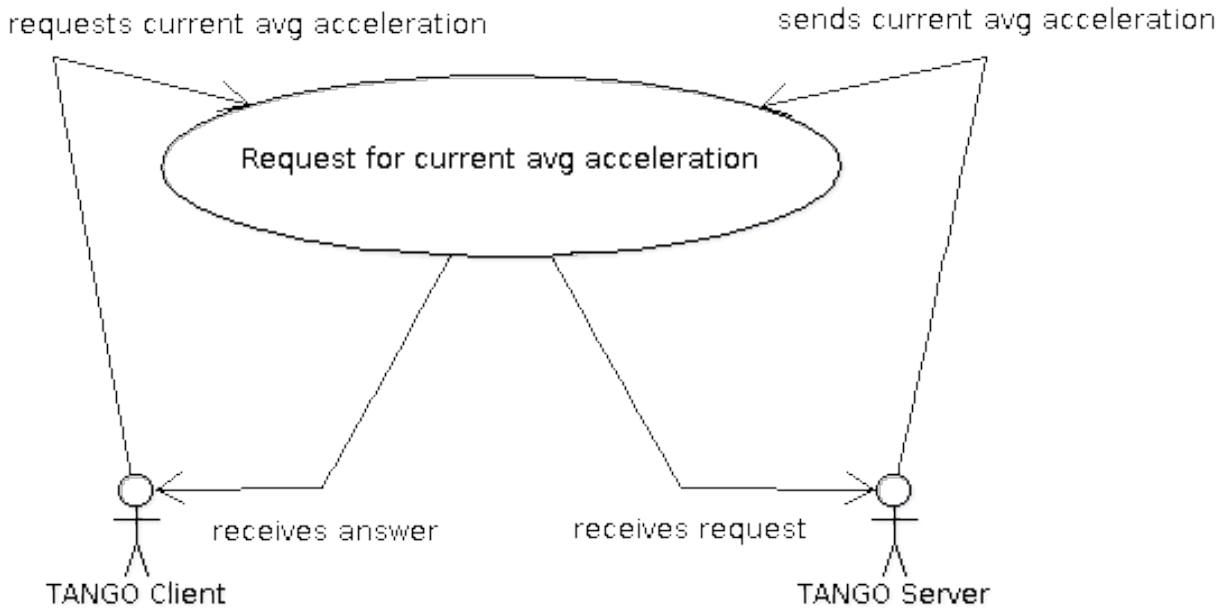
2.6.1.7 Postconditions None



2.6.1.8 Sequence diagram

2.6.2 Request for current average acceleration

The Client request the Server the average level of physical activity over the last T seconds.



2.6.2.1 Actors Client: a TANGO client that makes the request. Server: the Heart Rate Monitor TANGO server.

2.6.2.2 Priority Normal

2.6.2.3 Preconditions The Server is running and its DevState is ON.

2.6.2.4 Basic Course

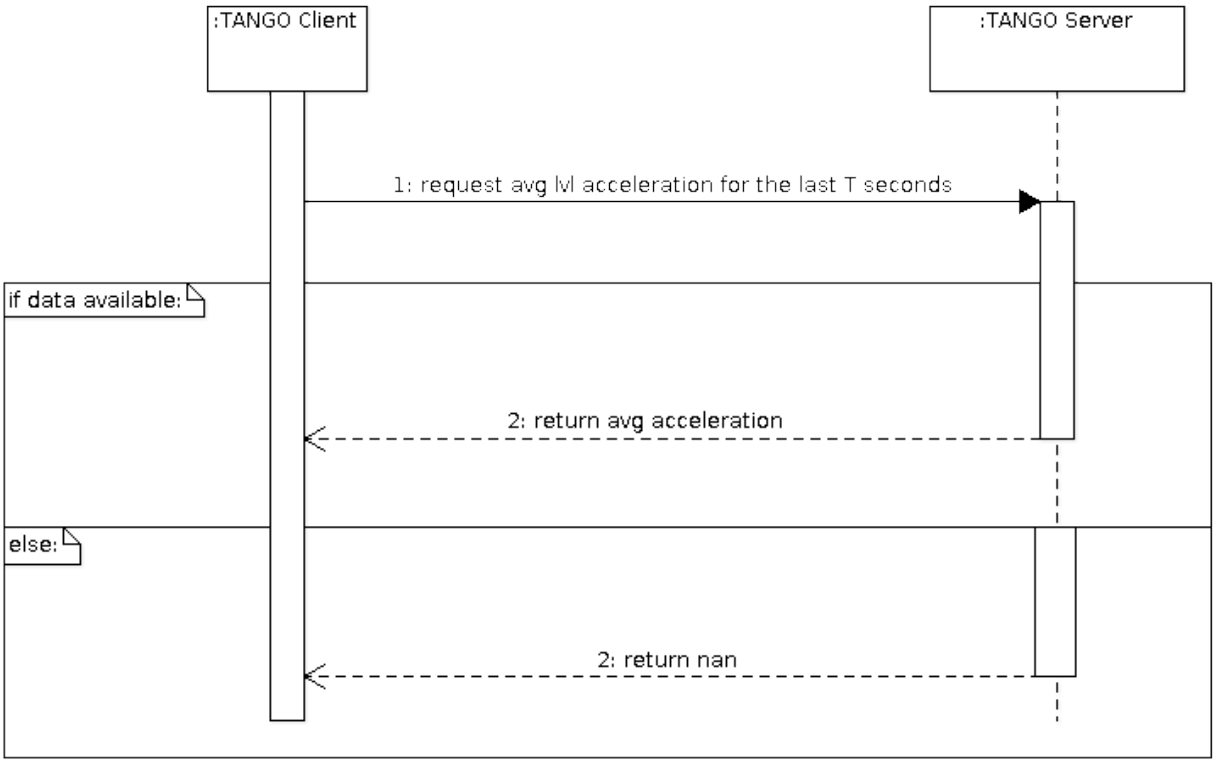
1. The Client calls the appropriate method on the Server, passing T as argument.
2. The Server calculates the average level of physical activity over the last T seconds, based on accelerometer data.
3. The Server returns the calculated value.

2.6.2.5 Alternate Course None

2.6.2.6 Exception Course

1. The Client calls the appropriate method on the Server, passing T as argument.
2. The Server tries calculates the average level of physical activity.
3. No data is available.
4. The Server returns an error.

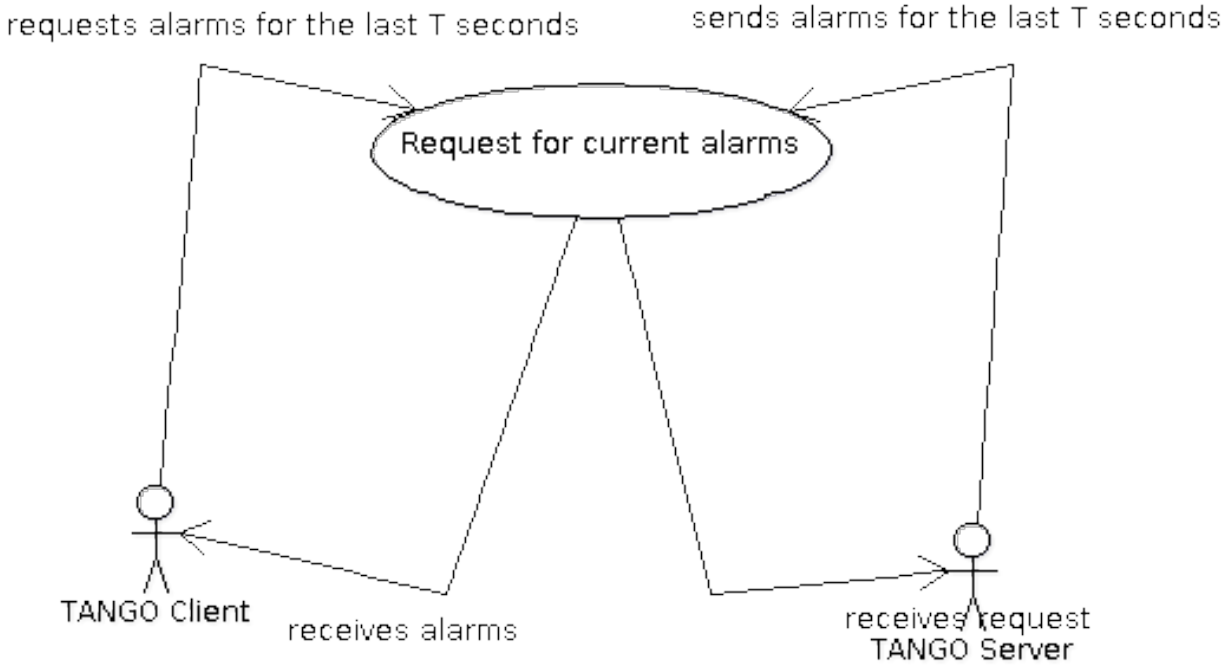
2.6.2.7 Postconditions None



2.6.2.8 Sequence diagram

2.6.3 Request for current alarms

The Client request the Server the list of alarms raised over the last T seconds.



2.6.3.1 Actors Client: a TANGO client that makes the request. Server: the Heart Rate Monitor TANGO server.

2.6.3.2 Priority High

2.6.3.3 Preconditions The Server is running and its DevState is ON.

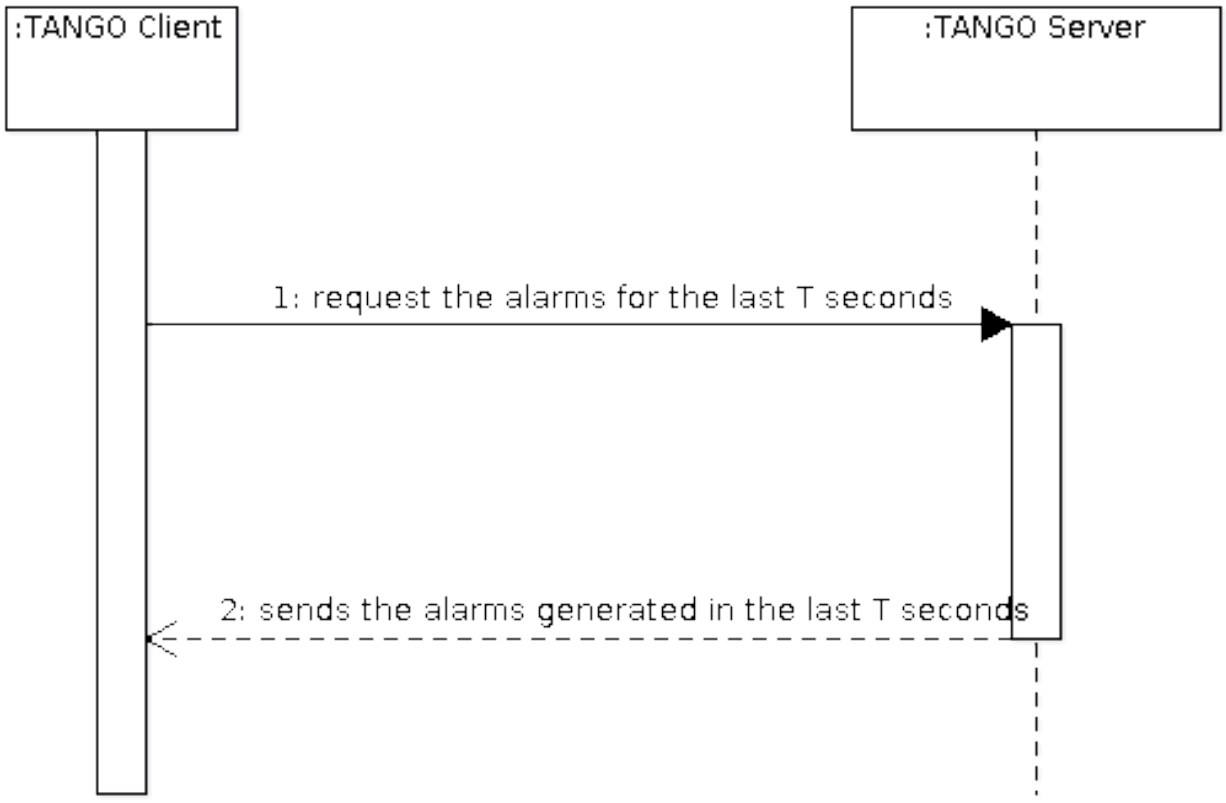
2.6.3.4 Basic Course

1. The Client calls the appropriate method on the Server, passing T as argument.
2. The Server returns the list of alarms raised over the last T seconds.

2.6.3.5 Alternate Course None

2.6.3.6 Exception Course None

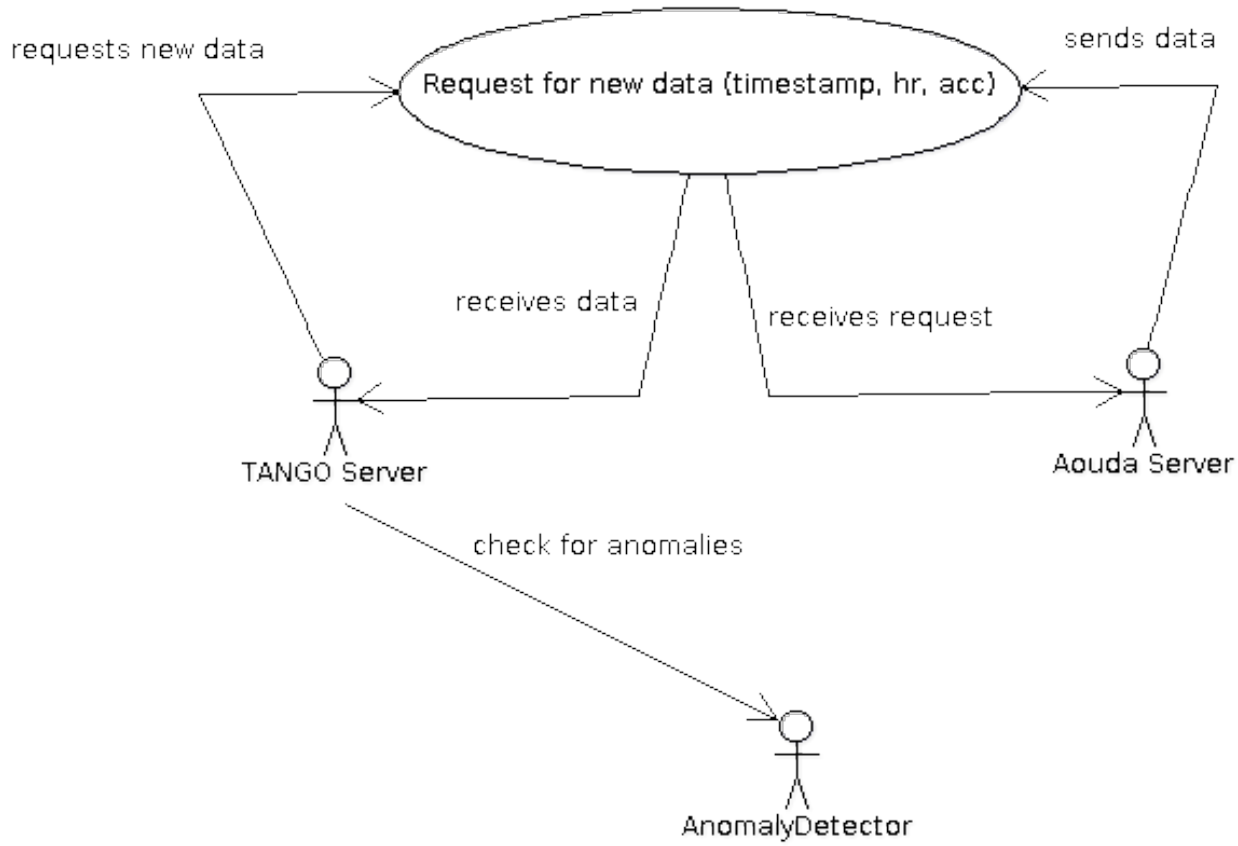
2.6.3.7 Postconditions None



2.6.3.8 Sequence diagram

2.6.4 Server requests new data

The Server reads new data from the Framework Software Bus, ands sends it to the Anomaly Detector for analysis.



2.6.4.1 Actors Server: the Heart Rate Monitor TANGO server. Anomaly Detector: the module in charge of detecting anomalies. Aouda Server: Tango server that provides the Aouda Suit simmulated data.

2.6.4.2 Priority High

2.6.4.3 Preconditions The Server is running and its DevState is ON.

2.6.4.4 Basic Course

1. The Server request new data from the Aouda Server.
2. The Aouda Server returns the data available.
3. The Server sends the new heart rate and accelerometer data to the Anomaly Detector.
4. The Anomaly Detector has enough data to build an anomaly analysis and returns the anomalies scores to the Server.

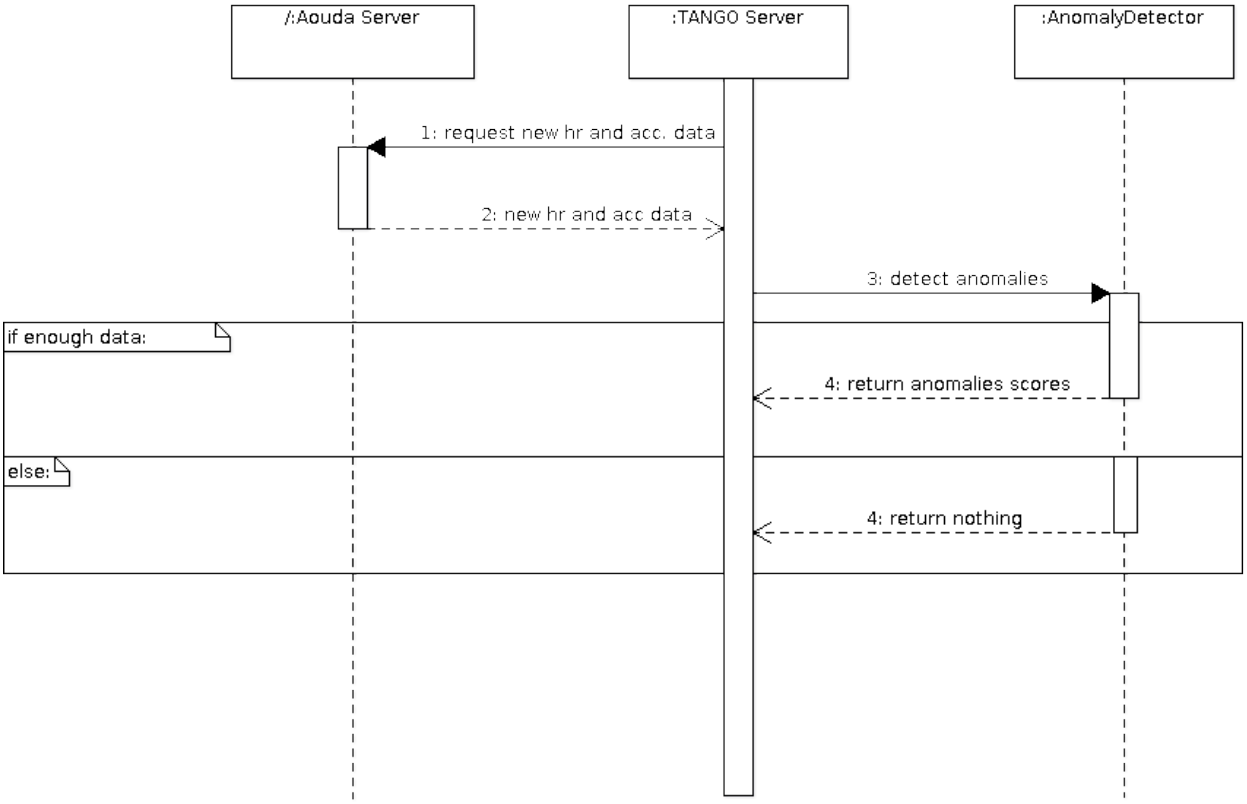
2.6.4.5 Alternate Course None

2.6.4.6 Exception Course

1. The Server request new data from the Aouda Server.
2. The Aouda Server returns the data available.

3. The Server sends the new heart rate and accelerometer data to the Anomaly Detector.
4. The Anomaly Detector has not enough data to build an anomaly analysis and returns nothing to the Server.

2.6.4.7 Postconditions None



2.6.4.8 Sequence diagram

5.6.3 3 Software Design Study for the Heart Rate Monitor

Author Mario Tambos

3.1 Change Record

2013.06.28 - Document created.

2013.09.18 - Updated to reflect current design. Some formatting changes.

3.2 Introduction

3.2.1 Scope

This document describes the top level requirements for the Heart Rate Monitor module, which in turn is part of the Crew Mission Assistant system.

3.2.2 Applicable Documents

- [1] – C3 Prototype document v.4
- [2] – PAMAP2 Physical Activity Monitoring
- [3] – Software Engineering Practices Guidelines for the ERAS Project
- [4] – ERAS 2013 GSoC Strategic Plan
- [5] – Software Requirements Specification for the Heart Rate Monitor
- [6] – TANGO distributed control system
- [7] – PyTANGO - Python bindings for TANGO
- [8] – A. Reiss and D. Stricker. Introducing a New Benchmarked Dataset for Activity Monitoring. The 16th IEEE International Symposium on Wearable Computers (ISWC), 2012.
- [9] – A. Reiss and D. Stricker. Creating and Benchmarking a New Dataset for Physical Activity Monitoring. The 5th Workshop on Affect and Behaviour Related Assistance (ABRA), 2012.
- [10] – Wei, Li, et al. "Assumption-Free Anomaly Detection in Time Series." SSDBM. Vol. 5. 2005. APA

3.2.3 Glossary

AD Anomaly Detection

API Application Programming Interface

ERAS European Mars Analog Station

IMS Italian Mars Society

HRM Heart Rate Monitor

TBC To Be Confirmed

TBD To Be Defined

3.3 Design Considerations

As stated in [5], the approach to the problem of monitoring the crew's heart rate,is to use *AD* techniques. Said techniques are however not unique; nor has a priori search for heart rate *AD* returned any directly applicable results. Therefore several other methods were investigated. After several tests, using data from [2] (see also [8] and [9]), and analysis of their results, the method described in [10] was selected. The objective of the design was nevertheless to encapsulate the anomaly detector itself in a single class, so it can both be used elsewere if necessary and be replaced if some better method is discovered.

3.3.1 Assumptions and dependencies

The *HRM* is to be programmed as a TANGO server, in the Python language. As such its primary dependencies are:

- The Python language.
- The TANGO distributed control system (see [6]).
- The PyTango bindings (see [7]).
- NumPy and Pandas, respectively a scientific computation and a data analysis library for Python, to perform the *AD* on the data.
- SqlAlchemy, a ORM for Python.

3.3.2 General Constraints

- Guidelines defined in [3].
- Requirements described in [5].

3.3.3 Objectives

- Provide some meaningful measure of the anormality level of a sequence of heart rate and acceleration readings, with respect to historic readings.
- Provide some measure of fault tolerance in the face of sensor errors.
- Keep the modules simple and easy to maintain.

3.4 Software Architecture

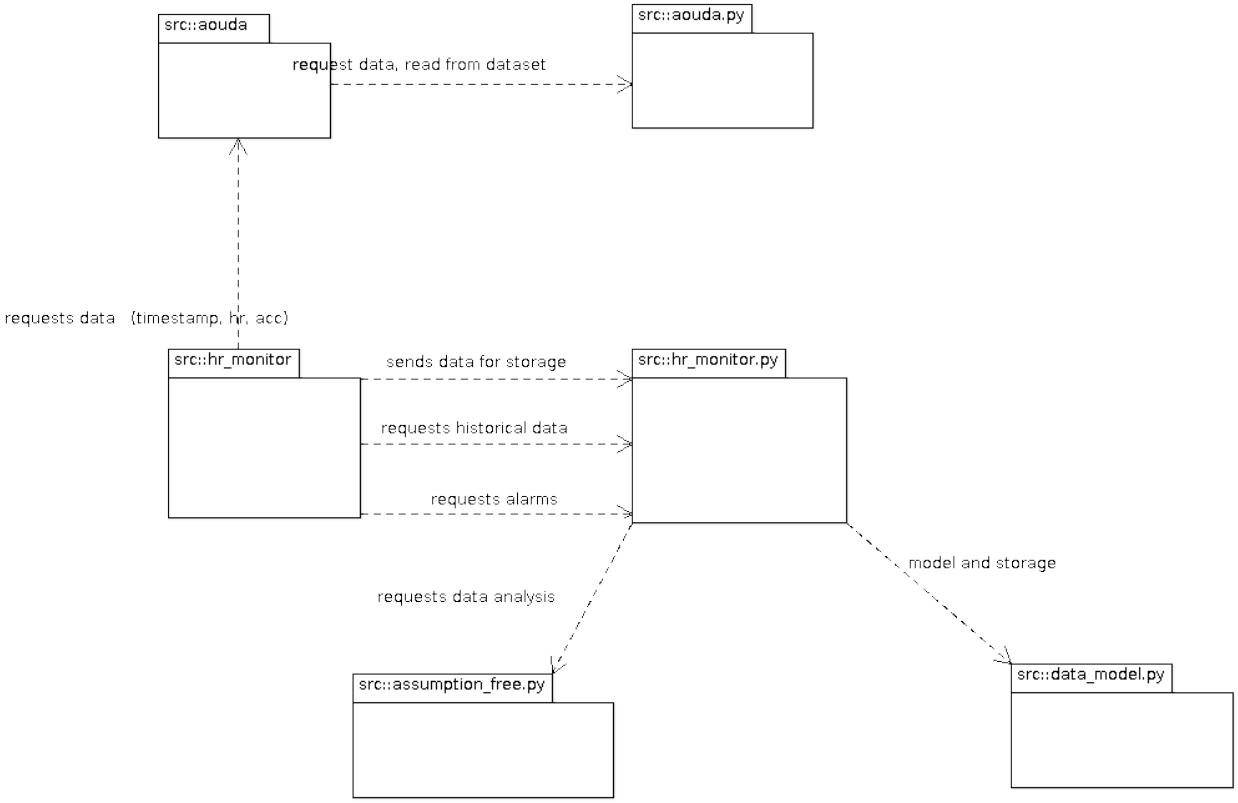
The *HRM* will be divided into two main modules: a TANGO Server, named HRMonitor, and an Anomaly Detector, named AssumptionFreeAA.

In order to test the *HRM* – and avoid the current problems with the Aouda.X suit – an additional TANGO Server will be built, named Aouda, from which the HRMonitor will get the simulated heart rate and accelerometer data (in turn taken from [2] (see also [8] and [9])).

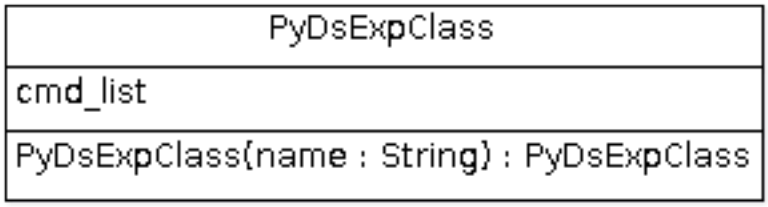
Use case and sequence diagrams showing the hig level interactions between the modules can be seen in section 2.6. of [5].

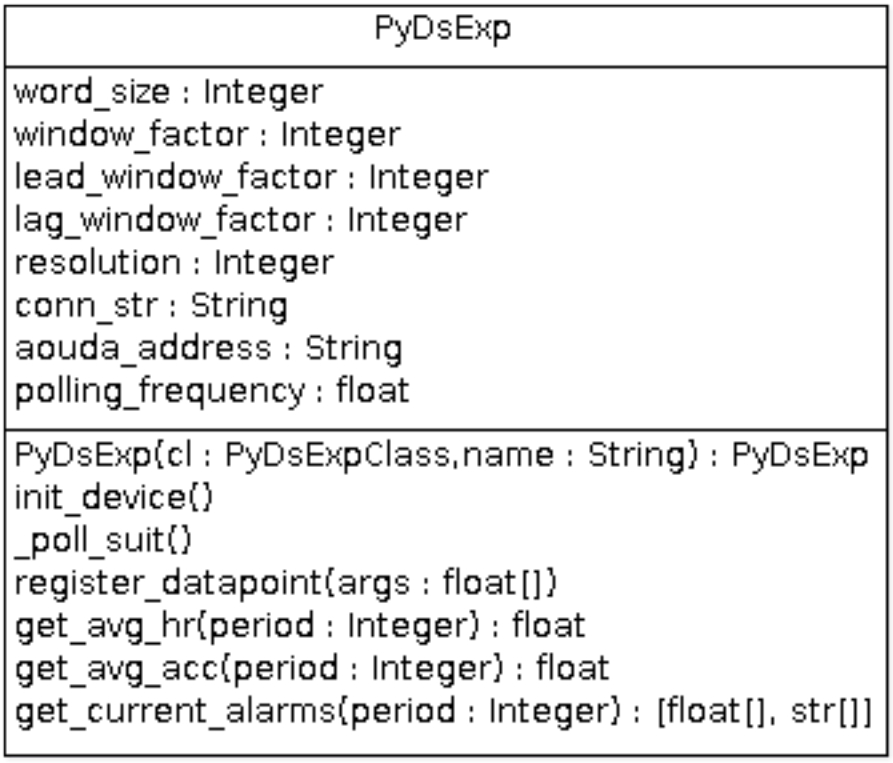
3.5 Software Design

A high level class diagram outlining the components can be seen below.


src::aouda
request data, read from dataset
src::aouda.py
requests data (timestamp, hr, acc)
src::hr_monitor
sends data for storage
src::hr_monitor.py
requests historical data
requests alarms
model and storage
requests data analysis
src::assumption_free.py
src::data_model.py

3.5.1 hr_monitor





3.5.1.1 Classification Package, Class, Tango's DeviceServer implementation.

3.5.1.2 Responsibilities Is in charge of interacting with the Aouda Server, implementing all necessary interfaces to integrate itself with the rest of the C3 Prototype and acquiring the heart rate and accelerometer data. It does not make any computation by itself, it only acts as a proxy between the external world and the *hr_monitor.py*.

3.5.1.3 Constraints Retrieves data from *aouda* with format:

```
[
      [hr1, acc_x1, acc_y1, acc_z1,
       hr2, acc_x2, acc_y2, acc_z2,
       ...
       hrN, acc_xN, acc_yN, acc_zN],
      [timestamp1, timestamp1, timestamp1, timestamp1,
```

```
        timestamp2, timestamp2, timestamp2, timestamp2,
        ...
        timestampN, timestampN, timestampN, timestampN],
]
```

Returns alarms data with format:

```
[[alarm_lvl1, ..., alarm_lvlN], [timestamp1, ..., timestampN]]
```

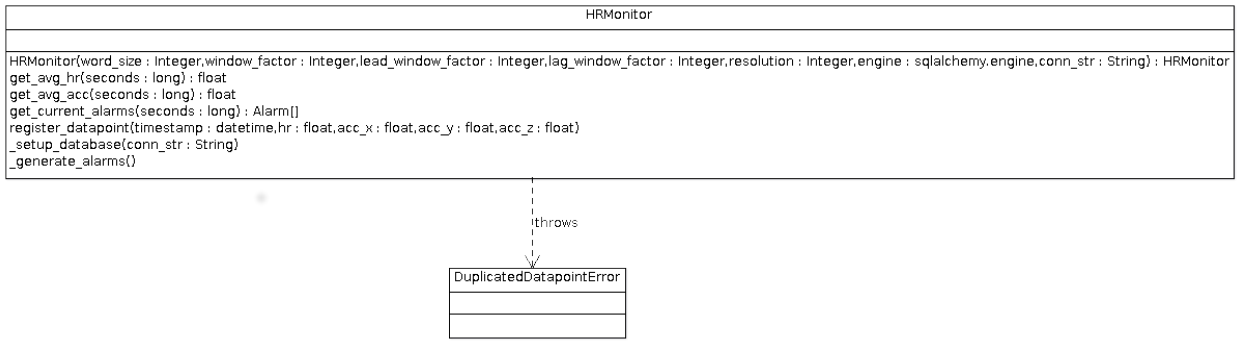
3.5.1.4 Composition The package's subcomponents are described below:

- **PyDsExpClass**
  - Type: class.
  - Function: defines DeviceServer's attributes and commands.
- **PyDsExp**
  - Type: class.
  - Function: implements interface defined in PyDsExpClass.

3.5.1.5 Uses/Interactions

- *hr_monitor.py*: *hr_monitor* forwards its requests to this package for processing.
- *aouda*: *hr_monitor* requests the suits' data from this package.

3.5.2 hr_monitor.py



3.5.2.1 Classification Package.

3.5.2.2 Responsibilities Interacts with data storage and *assumption_free.py*.

3.5.2.3 Constraints The response times should be keept in all cases at a minimum to avoid timeouts. As minimum we will consider here the timeout time of Tango synchronous requests.

3.5.2.4 Composition The package's subcomponents are described below:

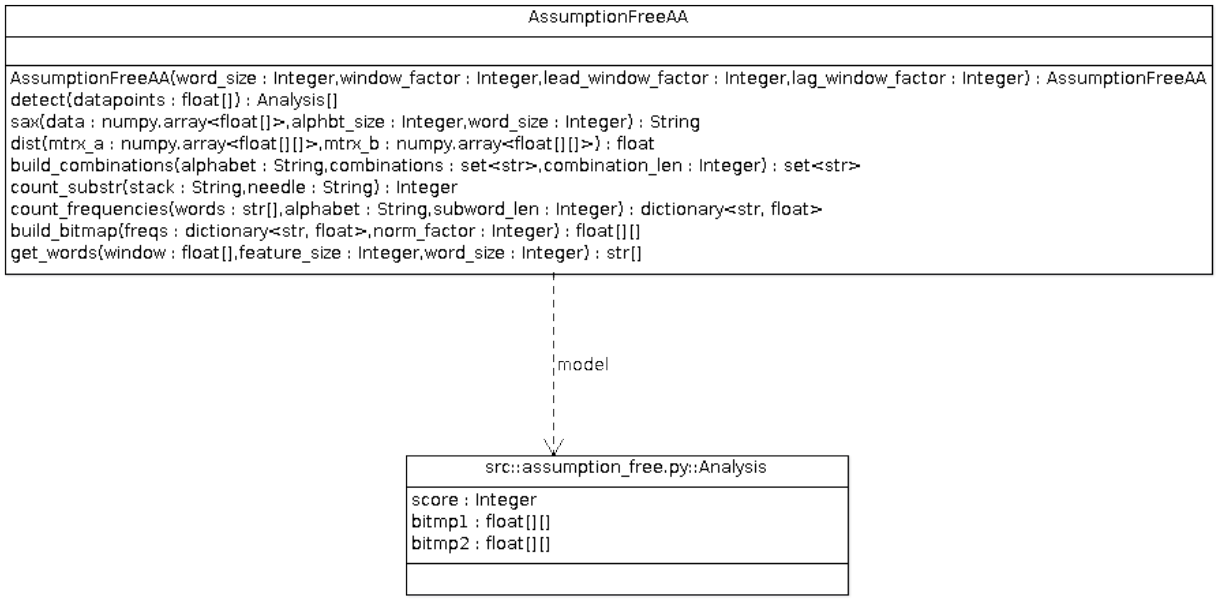
- **HRMonitor**
  - Type: class.

- Function: takes charge of package's responsibilities.

3.5.2.5 Uses/Interactions

- *data_model.py*: *hr_monitor.py* uses it to model the data, both suit's heart rate and acceleration redings and anomaly analysis results, and store it in a database.
- *assumption_free.py*: *hr_monitor.py* sends to it newly acquired data for analysis and receives back the analysis results.
- *hr_monitor*: *hr_monitor.py* receives from it forwarded requests and newly acquired data.

3.5.3 assumption_free.py



3.5.3.1 Classification

Package.

3.5.3.2 Responsibilities

Is in charge of applying the method described in [10] to the data provided by *hr_monitor.py*.

3.5.3.3 Constraints

- Each datapoint is a single dimensional vector.
- The analysis should contain both an anomaly score and its timestamp.

3.5.3.4 Composition

The package's subcomponents are described below:

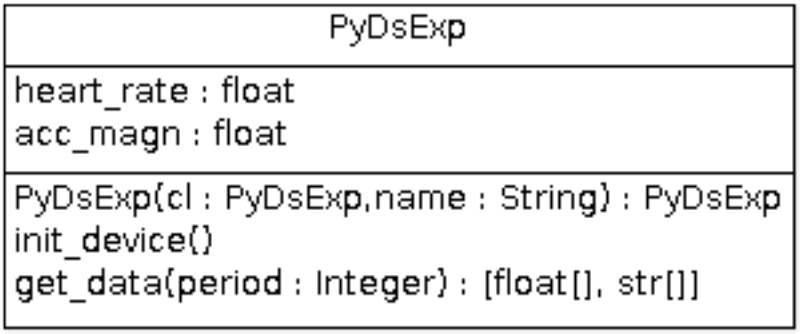
- **AssumptionFreeAA**
  - Type: class.
  - Function: takes charge of package's responsibilities.
- **AssumptionFreeAA.Analysis**
  - Type: named tuple.

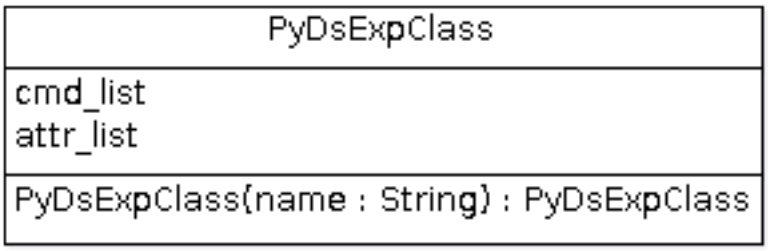
– Function: encapsulates the analysis result.

3.5.3.5 Uses/Interactions

- *hr_monitor.py*: *assumption_free.py* receives from it newly acquired data for analysis and sends back the analysis results.

3.5.4 aouda





3.5.4.1 Classification Package, Class, Tango's DeviceServer implementation.

3.5.4.2 Responsibilities Is in charge of simmulating data generated by the Aouda Suit and implementing all necessary interfaces to integrate itself with the rest of the C3 Prototype. It does not make any computation by itself, it only acts as a proxy between the external world and the *aouda.py*.

3.5.4.3 Constraints Returns heart rate and acceleration data with format:

```
[
      [hr1, acc_x1, acc_y1, acc_z1,
       hr2, acc_x2, acc_y2, acc_z2,
       ...
       hrN, acc_xN, acc_yN, acc_zN],
      [timestamp1, timestamp1, timestamp1, timestamp1,
       timestamp2, timestamp2, timestamp2, timestamp2,
```

```
        ...
        timestampN, timestampN, timestampN, timestampN],
]
```

3.5.4.4 Composition

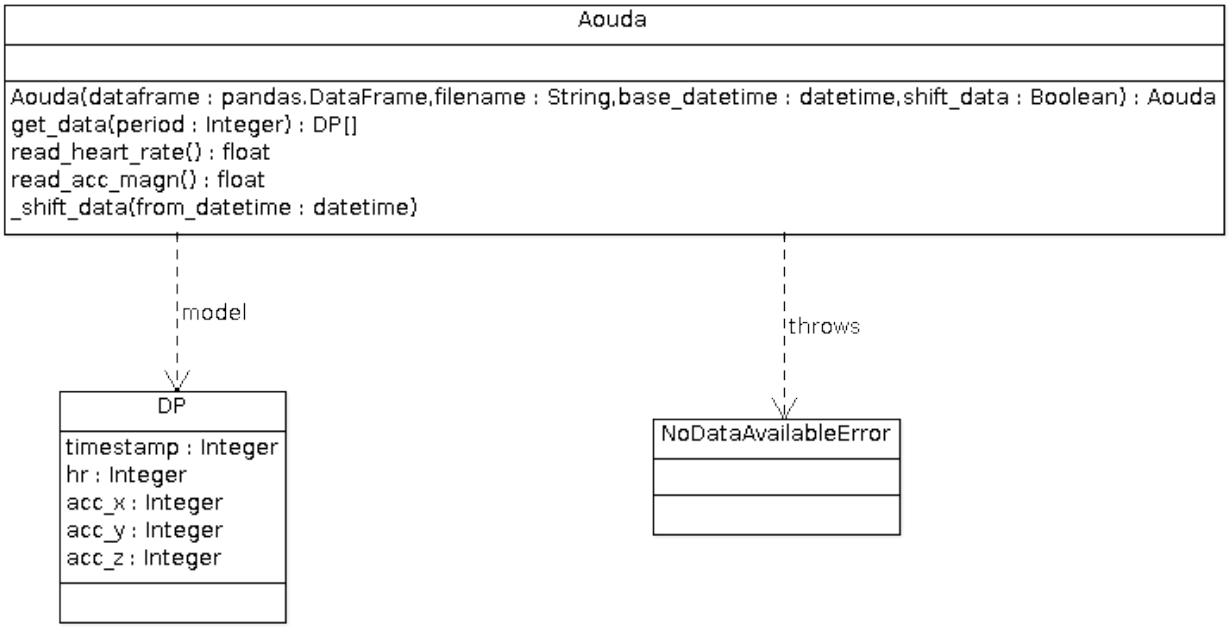
The package's subcomponents are described below:

- **PyDsExpClass**
  - Type: class.
  - Function: defines DeviceServer's attributes and commands.
- **PyDsExp**
  - Type: class.
  - Function: implements interface defined in PyDsExpClass.

3.5.4.5 Uses/Interactions

- *hr_monitor*: *aouda* receives requests for the suits' data from it.
- *aouda.py*: *aouda* forwards its requests to this package for processing.

3.5.5 aouda.py



3.5.5.1 Classification

Package.

3.5.5.2 Responsibilities

Is in charge of simulating the heart rate and accelerometer data generation of the Aouda.X suit.

3.5.5.3 Constraints

The response times should be keept in all cases at a minimum to avoid timeouts. As minimum we will consider here the timeout time of Tango synchronous requests.

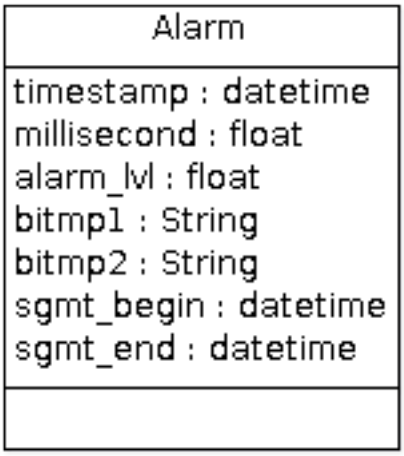
3.5.5.4 Composition The package's subcomponents are described below:

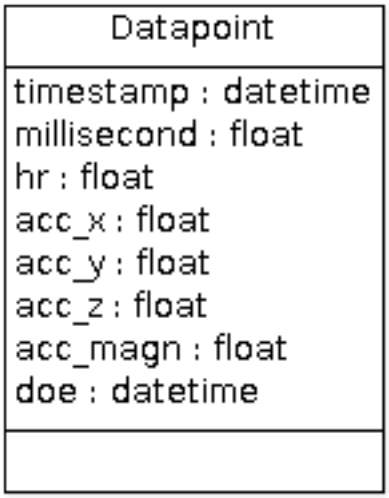
- **Aouda**
  - Type: class.
  - Function: takes charge of package's responsibilities.
- **Aouda.DP**
  - Type: named tuple.
  - Function: encapsulates the heart rate and acceleration data.

3.5.5.5 Uses/Interactions

- *aouda*: *aouda.py* receives from it forwarded requests for simmulated data.

3.5.6 data_model.py





3.5.6.1 Classification Package.

3.5.6.2 Responsibilities Is in charge of modeling the hear rate, acceleration and anomaly analysis data, and of managing its storage.

3.5.6.3 Constraints None.

3.5.6.4 Composition The package's subcomponents are described below:

- **Datapoint**
  - Type: class.
  - Function: models a heart rate-acceleration datapoint.
- **Alarm**

- Type: class.
- Function: models a single anomaly analysis result.

3.5.6.5 Uses/Interactions

- *hr_monitor.py*: *data_model.py* provides to it the classes necessary to model the data used.

5.7 Myro Rover

http://wiki.roboteducation.org/Myro_Reference_Manual

5.7.1 Install Myro

```
sudo apt-get install python-tk
sudo apt-get install idle
sudo apt-get install python-imaging-tk
```

get latest release from http://myro.roboteducation.org/download/:

```
sudo unzip myro-2.9.1.zip
cd myro
sudo python setup.py install
```

5.7.2 Commands

For the moment only the command:

```
move(translate_speed, rotate_speed)
```

has been implemented

- translate_speed: 0 to 1 moves forward; 0 to -1 moves backwards
- rotate_speed: 0 to 1 turns left, 0 to -1 turns right

5.8 Neuro Headset Server

5.8.1 Neuro Headset Server Setup

Dependencies

The following dependencies are required to use the Neuro Headset:

```
sudo apt-get install python-setuptools python-gevent python-dev realpath
wget https://github.com/openyou/emokit/archive/master.zip
unzip master.zip
cd emokit-master/python/
sudo python setup.py install
```

Running the server

To run the stand-alone server use:

```
$ python neurohs.py 400
```

the argument (e.g. 400) is the polling period in milliseconds.

If you want to log the output on a file, you can also use:

```
$ python neurohs.py 400 --log neurohs.log
```

If you want to read the data from a log file (i.e. simulation mode), you can use:

```
$ python neurohs.py 400 --sim neurohs.log
```

To run the server from Tango use:

```
$ python neurohs epoc1
```

The argument (e.g. epoc1) is the server name registered in Jive.

Troubleshooting

If you installed all the dependencies, everything should work.

If you see this error:

```
emokit-master/python$ sudo python setup.py install
Traceback (most recent call last):
  File "setup.py", line 2, in <module>
    from setuptools import setup
ImportError: No module named setuptools
```

you have to `sudo apt-get install python-setuptools`.

If you see this error:

```
emokit-master/python$ sudo python setup.py install
...
config.status: executing src commands
source/_ctypes.c:107:20: fatal error: Python.h: No such file or directory
compilation terminated.
error: Setup script exited with error: command 'i686-linux-gnu-gcc' failed with exit status 1
```

you have to `sudo apt-get install python-dev`.

If you see this error:

```
>>> headset.setup()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/dist-packages/emokit-0.0.1-py2.7.egg/emokit/emotiv.py", line 353, in
    self.setupPosix()
  File "/usr/local/lib/python2.7/dist-packages/emokit-0.0.1-py2.7.egg/emokit/emotiv.py", line 450, in
```

```
    setup = self.getLinuxSetup()
  File "/usr/local/lib/python2.7/dist-packages/emokit-0.0.1-py2.7.egg/emokit/emotiv.py", line 370, in
    realInputPath = check_output(["realpath", "/sys/class/hidraw/" + filename])
  File "/usr/lib/python2.7/subprocess.py", line 568, in check_output
    process = Popen(stdout=PIPE, *popenargs, **kwargs)
  File "/usr/lib/python2.7/subprocess.py", line 711, in __init__
    errread, errwrite)
  File "/usr/lib/python2.7/subprocess.py", line 1308, in _execute_child
    raise child_exception
OSError: [Errno 2] No such file or directory
```

you have to `sudo apt-get install realpath`.

If you see this error:

```
$ python neurohs.py 400
Serial: SNxxx Device: hidraw1
Serial: SNxxx Device: hidraw2 (Active)
Traceback (most recent call last):
  File "/usr/lib/python2.7/dist-packages/gevent/greenlet.py", line 390, in run
    result = self._run(*self.args, **self.kwargs)
  File "/usr/local/lib/python2.7/dist-packages/emokit-0.0.1-py2.7.egg/emokit/emotiv.py", line 353, in
    self.setupPosix()
  File "/usr/local/lib/python2.7/dist-packages/emokit-0.0.1-py2.7.egg/emokit/emotiv.py", line 453, in
    self.hidraw = open("/dev/" + setup[1])
IOError: [Errno 13] Permission denied: '/dev/hidraw2'
```

you need to run `sudo python neurohs.py 400`.

If you see a similar output:

```
$ python neurohs.py 400
Serial: SNxxx Device: hidraw0
Serial: SNxxx Device: hidraw1 (Active)
```

but nothing else is printed, try to turn off the headset, wait a few seconds, and turn it on again *without* restarting the script.

5.8.2 Software Requirements Specification for the Neuro Headset Server

Author Ezio Melotti

Change Record

Introduction

Purpose

This document describes the software requirements specification for the neuro headset server.

Scope

Describes the scope of this requirements specification.

Applicable Documents

Reference Documents

Glossary

Overview

The package will initially provide low-level access to the data sent by the neuro headset. The package will also provide training software and ways to define an higher-level interface used to control several different devices.

General Description

Problem Statement

In a manned mission on Mars, astronauts need to control a number of devices (e.g. a Mars rover) but often have limited mobility. Ideally, the best approach consists in an hand-free input control, such as voice command or brain waves. These input methods would allow astronauts to operate devices without needing specific hardware (e.g. a joystick), and even in situations of limited mobility (e.g. while wearing a space suit). While these input methods clearly have advantages, they might not be as accurate as traditional input methods.

Functional Description

The package will offer an interface between Tango and the EPOC neuro headset. Developers can use then use the data provided by the package to control different kind of devices.

Environment

The neuro headset can be used in the habitat or even during EVAs while wearing a space suit, but will requires a nearby computer that will receive and process the signal.

User objectives

User1 Describe all the users and their expectations for this package

Constraints

Describe any constraints that are placed on this software.

Functional Requirements

This section lists the functional requirements in ranked order. Functional requirements describe the possible effects of a software system, in other words, what the system must accomplish. Other kinds of requirements (such as interface requirements, performance requirements, or reliability requirements) describe how the system accomplishes its functional requirements. Each functional requirement should be specified in a format similar to the following:

Requirement

Description

Criticality

- High | Normal | Low

Dependency Indicate if this requirement is dependant on another.

Interface Requirements

This section describes how the software interfaces with other software products or users for input or output. Examples of such interfaces include library routines, token streams, shared memory, data streams, and so forth.

User Interfaces

Describes how this product interfaces with the user.

GUI (Graphical User Interface)

CLI (Command Line Interface)

API (Application Programming Interface) The Tango server will provide a low-level API to access the raw data for the neuro headset. This low level API, in combination with a training software, will be used to create higher-level APIs, possibly as new servers. The actual APIs are still to be determined.

Diagnostics Describes how to obtain debugging information or other diagnostic data.

Hardware Interfaces

A high level description (from a software point of view) of the hardware interface if one exists. This section can refer to an ICD (Interface Control Document) that will contain the detail description of this interface.

Software Interfaces

A high level description (from a software point of view) of the software interface if one exists. This section can refer to an ICD (Interface Control Document) that will contain the detail description of this interface.

Communication Interfaces

Describe any communication interfaces that will be required.

Performance Requirements

Specifies speed and memory requirements.

Development and Test Factors

Standards Compliance

Mention to what standards this software must adhere to.

Hardware Limitations

Describe any hardware limitations if any exist.

Software validation and verification

Give a detail requirements plan for the how the software will be tested and verified.

Planning

Describe the planning of the whole process mentioning major milestones and deliverables at these milestones.

Use-Case Models

If UML Use-Case notation is used in capturing the requirements, these models can be inserted and described in this section. Also providing references in paragraphs 5, 6 and 7 where applicable.

Notes

Appendix A: Use Case template

Use Case: Controlling a rover with the neuro headset

The user wants to control a rover using the neuro headset.

Actors

User, rover.

Priority

Normal

Preconditions

The user should be wearing a charged neuro headset and be within wireless range of a computer that will receive and process the data, making them available on Tango. The user might be required to do a training before being able to use the neuro headset successfully.

Basic Course

1. If the user didn't do the training yet, he should do it in order to associate specific thoughts to specific movements.
2. After the training, he should be able to just think at the movements the rover should do.
3. The server will process the inputs sent by the headset and convert them to higher level signals, accoding to the data collected during the training.
4. The higher level signals can be accessed by other servers (e.g. the rover server) and used to determine what actions should be taken.

Alternate Course

None

Exception Course

If any of the preconditions are not met, the user should make sure to address the problems before continuing with the basic course.

Postconditions

At the end of the session the user should turn off and remove the neuro headset, and recharge it if needed.

Notes

None

5.8.3 Software Design Study for the Neuro Headset Server

Author Ezio Melotti

Change Record

Introduction

Purpose

This document describes the software design study of the neuro headset server.

Scope

Applicable Documents

Reference Documents

Glossary

Overview

Provides a brief overview of the package defined as a result of the requirements elicitation process.

Design Considerations

This section describes many of the issues, which need to be addressed or resolved before attempting to devise a complete design solution.

Assumptions and dependencies

Describe any assumptions or dependencies regarding the software and its use. These may concern such issues as:

- Related software or hardware
- Operating systems
- End-user characteristics
- Possible and/or probable changes in functionality

General Constraints

Describe any global limitations or constraints that have a significant impact on the design of the system's software (and describe the associated impact). Such constraints may be imposed by any of the following (the list is not exhaustive):

- Hardware or software environment
- End-user environment
- Availability or volatility of resources
- Standards compliance
- Interoperability requirements
- Interface/protocol requirements
- Data repository and distribution requirements
- Security requirements (or other such regulations)
- Memory and other capacity limitations
- Performance requirements
- Network communications
- Verification and validation requirements (testing)
- Other means of addressing quality goals

- Other requirements described in the requirements specification

Objectives

The objective of the server is to provide within the TANGO infrastructure all the data available from the neuro headset. The server (or possibly a different server) will also process the data and provide an higher-level result.

These data can be in turn used by other servers.

Software Architecture

The server will provide two different API at two different levels. At the lowest level will have the raw data read by all the EEG sensors, including the signal intensity and signal quality. These data will be processed and the result will be provided on an higher-level API.

In addition to these two APIs, a third API that provides the data from the accelerometers and possibly battery level will be provided.

These APIs might be provided by the same server or by different servers, but this hasn't be defined yet.

Software Design

The class diagrams containing all the classes can be put here.

Unit n

Most components described in the System Architecture section will require a more detailed discussion. Other lower-level components and subcomponents may need to be described as well. Each subsection of this section will refer to or contain a detailed description of a system software component. The discussion provided should cover the following software component attributes:

Classification

The kind of component, such as a subsystem, module, class, package, function, file, etc.

Definition

The kind of component, such as a subsystem, module, class, package, function, file, etc.

Responsibilities

The primary responsibilities and/or behavior of this component. What does this component accomplish? What roles does it play? What kinds of services does it provide to its clients? For some components, this may need to refer back to the requirements specification.

Constraints

Any relevant assumptions, limitations, or constraints for this component. This should include constraints on timing, storage, or component state, and might include rules for interacting with this component (encompassing preconditions, postconditions, invariants, other constraints on input or output values and local or global values, data formats and data access, synchronization, exceptions, etc.)

Composition

A description of the use and meaning of the subcomponents that are a part of this component.

Uses/Interactions

A description of this components collaborations with other components. What other components is this entity used by? What other components does this entity use (this would include any side-effects this entity might have on other parts of the system)? This concerns the method of interaction as well as the interaction itself. Object-oriented designs should include a description of any known or anticipated subclasses, super-classes, and meta-classes.

Unit n+1

5.9 ERAS Planning

5.10 Solar Storm Forecasting Server

5.10.1 Software Requirements Specification for the Solar Storm Forecasting

Author Simar Preet Singh

Change Record

2013.06.23 - Document created

Introduction

Purpose

This document describes the software requirements specification for the Solar Storm forecasting.

Scope

Describes the scope of this requirements specification.

Reference Documents

- [1] – GOES Data warehouse.
- [2] – SXI Solar X-ray imager.
- [3] – Software Engineering Practices Guidelines.
- [4] – ERAS 2013 GSoC Strategic Plan.
- [5] – Solar Dynamics Observatory Data.
- [6] – Example DSD.

Glossary

ERAS European Mars Analog Station

IMS Italian Mars Society

GOES Geostationary Operational Environmental Satellite

NGDC National Geophysical Data Center

SDO Solar Dynamics Observatory

DSD Daily Solar Data

SESC Space Environment Services Center

Overview

General Description

Problem Statement

The magnetosphere around the Earth protects us to certain extent, from the constant bombardment by charged particles from the sun. But, Much of Mars' atmosphere, on the other hand, is exposed directly to these fast-moving particles from the sun and the effects of solar flares. These storms of solar radiation can disrupt satellite communication, resource information, electrical power, and radar. Energy in the form of hard x-rays can also damage space craft electronics. The module aims to forecst such events and issue relevant warning.

Functional Description

The goal of the module is to provide a Neural Network implementation trained using local database to be constructed. This trained Neural Net can then be interfaced with the Tango to issue a warning based on the type of solar flare.

User objectives

Describe all the users and their expectations for this package

Constraints

Although, the archive data for the training is readily available in [1]. The module is constrained by the continued availability of functional data from the respective satellites.

Functional Requirements

Interface Requirements

User Interfaces

Diagnostics A validation set of data will be maintained for the diagnostic requirements.

Software Interfaces

Communication Interfaces The module is to be implementhed as a Python Tango server, which issues appropriate warnings in case of forecasted Solar storm.

Development and Test Factors

Standards Compliance

The Software Engineering Practices Guidelines for the ERAS Project in [3] to be followed.

Planning

The planned steps for the design and implementation of the model :

1. Variable selection
2. Data collection
3. Data preprocessing
4. Training and validation sets
5. Neural network paradigms
6. Evaluation criteria
7. Neural network training
8. Implementation

This procedure is not a single-pass one, and may require the revisiting of previous steps especially between training and variable selection. Although, the implementation step is listed as last one, it is being given careful consideration prior to collecting data.

Use-Cases

Use Case: Data collection and integration

The main focus is Data colection and preprocessing.

Actors Raw data, local database

Priority High

Preconditions The raw data (txt files) must be downloaded on lacal machine.

Basic Course The raw data from the warehouse in [1] is to be parsed and the data to be stored on local database (preferably using Mysql). The data collected from the txt files will be integrated in database using the date as key. An example of the *DSD* file is in [5]. Using this:

The following feature sets will be extarcted

1. Radio flux
2. *SESC* Sunspot number
3. Sunspot area
4. New regions
5. X-ray background flux
6. C-forecast
7. M-forecast
8. X-forecast

The database will then be seperated into training and validation sets to be used for the neural network training.

Alternate Course Although, it was initially thought of using image data from *SDO* in [4]. But, it is presently generating about 1.5TB of data daily and even the downsampled images would require immense processing power and bandwidth (SDO is receiving about 700Mb every 36 secs). Such processing power is not currently available for this implementation.Still, attempts wil be made to find any source of processed data access points or APIs which may provide us the preprocessed data.

Postconditions The database split into training and validation sets.

Use Case: Neural network training

The focus will be to train the neural network to classify the Solar flares.

Actors Neural network, local database

Priority High

Preconditions The training database must be available.

Basic Course Using the training database, four different Neural networks will be trained where each neural net will be trained to classify the features into a different class (classes to be trained for X, M, C, A&B). Each of these neural nets will be trained for one class only. The extracted feature set in the database will be used to identify the class as the output.

The following Neural Network paradigms will be considered :

1. number of hidden layers
2. number of hidden neurons
3. transfer functions

Additional factors considered for the training :

1. number of training iterations
2. learning rate
3. momentum

After the training, the validation set will be used to verify the performance for the neural network.

Alternate Course As an alternate course, a neural network consisting of multiple outputs to classify the features into the respective classes can be trained. Based on the input feature set, the output will be the corresponding class. The performance of both the implementations can be analysed to identify the most suitable solution.

Postconditions A trained neural network implementation.

Use Case: Solar storm warning

Actors Trained neural network as server, client that responds to warning

Priority Normal

Preconditions The neural network has access to input data feed.

Basic Course The input features would be fed to the trained neural network. As the network has already been trained offline, the implemented neural network should be able to provide fast response. In case of a warning, the relevant warning will be issued specifying the type of forecast.

Alternate Course None

Postconditions None

5.10.2 Software Design Study for the Solar Storm Forecasting

Author Simar Preet Singh

Change Record

2013.07.23 - Document created.

Introduction

Purpose

This document describes the software design study of the solar storm forecasting server module.

Scope

The scope of the Solar storm forecast server is to be able to issue warning in case of an expected solar storm, to prevent disruption of satellite communication and damage to space craft electronics.

Reference Documents

- [1] – PyBrain Library.
- [2] – GOES Data warehouse.
- [3] – Space Weather Prediction Center.
- [4] – Software Engineering Practices Guidelines.
- [5] – PyTANGO - Python bindings for TANGO.

Glossary

ERAS European Mars Analog Station

IMS Italian Mars Society

GOES Geostationary Operational Environmental Satellite

NGDC National Geophysical Data Center

SDO Solar Dynamics Observatory

SWPC Space Weather Prediction Center

Overview

The module attempts to forecast a solar storm event using Neural Network implementation. A neural network architecture is trained offline on the prepared dataset. This trained neural network can then be implemented as a Tango server to issue warnings in case of relevant events.

Design Considerations

Assumptions and dependencies

Some of the dependencies for the module are PyBrain in [1] and PyTango in [5]. The PyTango library is used for the Python bindings of the Tango distributed control system. The PyBrain library is being used for the construction of different possible neural network architectures. Please note that the documentation of the PyBrain library has not been updated for a long time. So there seem to be some differences in the documentation and the actual code. Thus, it is highly recommended that the PyBrain code be studied instead of just relying only on the documentation.

General Constraints

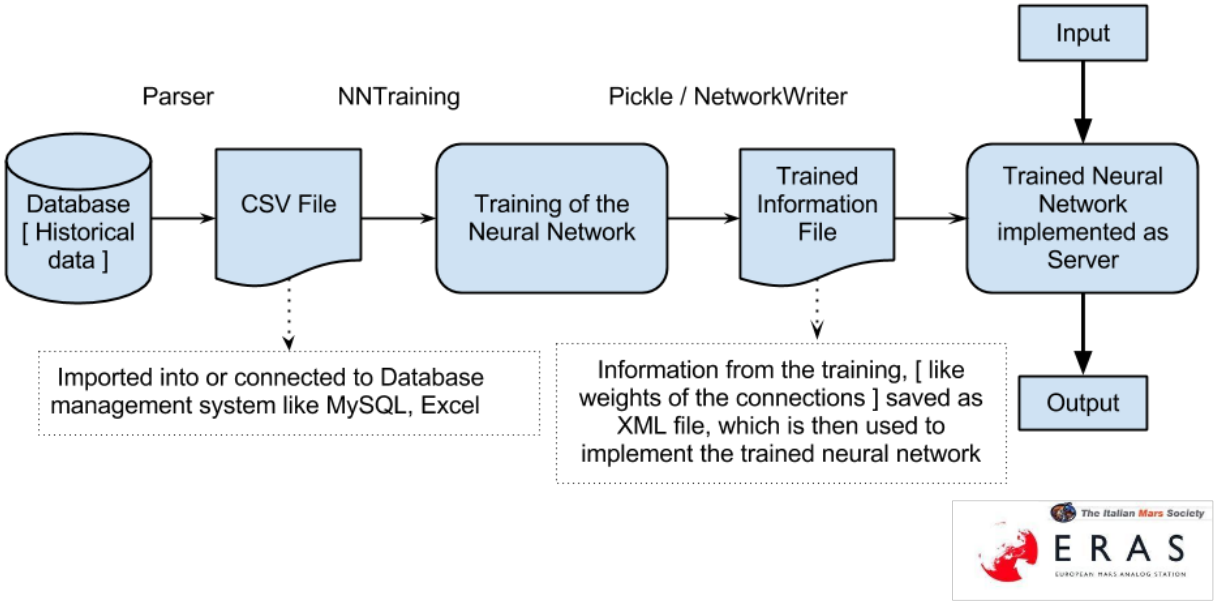
A general constraint for the module is continued availability of the satellite data. For instance the NASA's Solar Dynamics Observatory is planned as 5 - 10 years mission of observing the sun. As the module relies on the data from the satellites. So, the availability of such satellite data feed acts as a constraint for the module.

Software Architecture

This section should provide a high-level overview of how the functionality and responsibilities of the system were partitioned and then assigned to subsystems or components. Don't go into too much detail about the individual components themselves (there is a subsequent section for detailed component descriptions). The main purpose here is to gain a general understanding of how and why the system was decomposed, and how the individual parts work together to provide the desired functionality.

At the top-most level, describe the major responsibilities that the software must undertake and the various roles that the system (or portions of the system) must play. Describe how the system was broken down into its components/subsystems (identifying each top-level component/subsystem and the roles/responsibilities assigned to it). Describe how the higher-level components collaborate with each other in order to achieve the required results. Don't forget to provide some sort of rationale for choosing this particular decomposition of the system (perhaps discussing other proposed decompositions and why they were rejected).

A brief software architecture diagram is presented here :



Software Design

Unit 1 : Local Database Creation

Definition

This unit is a data retriever module that is used for the creation of local database.

Responsibilities

The responsibility of this module is to retrieve files and data from possibly different sources to create a common local database. The reason for creating a local offline database is that, such local database can be better managed and processed without any possible interruptions.

Constraints

The module is constrained on the correctness of the data retrieved.

Composition

The unit is composed of a Data retriever module that can be used to retrieve data files from possibly different sources, and add those files to the local database.

Uses/Interactions

This is meant to be an automated module which needs to be provided the local directory address. Additional methods can be implemented for downloading the data files from different sources.

Unit 2 : Parser Unit

Definition

This a data parser unit for parsing the various data files and creating a common database [CSV file].

Responsibilities

The responsibility of this module is to parse the formatting of the various data files in the local database. It is responsible for creating a common database in the form of a single CSV file. The advantage of having a single CSV file is that it can be easily connected or imported to different database management systems like MySQL, Excel.

Constraints

The correctness of the data files and their formatting is major constraint. As an instance such data file from the GOES warehouse seems to be highly inconsistent. Almost all of the entries in such file have 0 as flare values. Such inconsistencies can harm the training of the neural network.

Composition

The unit is composed of a Parser module that writes the CSV file based on the formatting of the raw input data files.

Uses/Interactions

This is meant to be a module which writes a CSV file based from the input data files. It needs to be provided with the source directory address of the local database. Additional methods can be implemented depending on the formatting of the raw files.

Unit 3 : Neural Network training unit

Definition

This is the unit that trains the Neural Network.

Responsibilities

The primary responsibility of this unit is to train the neural network based on the CSV file generated from the Parser unit. The unit aims to train the neural network on the target values, for the different input features.

Constraints

The training of the neural network is constrained on the quality of the the input dataset. AI techniques perform only as good as the quality of the training dataset.

Composition

The unit is composed of the Neural Network architecture. Various network architectures including Multiple output, or different Single output networks can be attempted.

Uses/Interactions

Various training techniques like backpropagation with different learning rates, momentum can be used to achieve optimal results for the training. Other classification techniques can also be used.

Unit 4 : Trained information file

Definition

This file contains the information retrieved [i.e. results] from the training of the neural network.

Responsibilities

This file is responsible for providing the weights and other neural features for the final trained network implementation.

Constraints

This is a static file which contains only the neural network information from the training performed locally.

Composition

Two different methods can be used to create this file and the resulting formats will depend on the choice. The use of NetworkWriter produces an XML file. Also, the native Python Pickle can be used.

Uses/Interactions

This file contains the results from the neural network training unit. These results can then be used to implement the final trained neural network. So, this will avoid the need to retrain the neural network every time, as the training results are made available in this file.

Unit 4 : Trained neural network implementation

Definition

This is the final trained neural network implemented as a Tango server.

Responsibilities

This unit is responsible for the forecast and the issuing of relevant warning in case of an identified storm.

Constraints

This unit is constrained on the availability of the data feed for the input of the network.

Composition

This is composed of the trained neural network architecture. It also contains the relevant implementations for the Tango integration.

Uses/Interactions

This unit will interact with the Tango control system and provide the relevant warning system.

5.10.3 Readme

This draft document provides instructions for installation of the dependencies, and also running the code.

Setup Git

You need to have Git installed on your system for cloning PyBrain:

```
sudo apt-get install git
```

For systems other than Debian/Ubuntu, you can use the corresponding package managers. Additional Git installation instructions can be found at Git-scm.

Setup SciPy

For Ubuntu:

```
sudo apt-get install python-numpy python-scipy python-matplotlib ipython ipython-notebook python-pand
```

For other systems, the installation instructions are available at SciPy.

Setup PyBrain

Make sure you have SciPy installed, then Clone the PyBrain library from Github using this command:

```
git clone git://github.com/pybrain/pybrain.git
```

From the cloned directory, install PyBrain using:

```
python setup.py install
```

Running the Code

Data Retriever unit:

```
python dataretriever.py /home/Repos/eras/servers/solarstorm/database/data
```

Provide the destination directory as argument. The raw data files will be downloaded to the provided directory.

Parser unit:

```
python parser.py /home/Repos/eras/servers/solarstorm/database/data
```

Provide the source directory, where you downloaded the data files in the previous step. This unit will parse the data files in the source directory and generate the CSV file.

Neural Network Training:

```
python neuraltraining.py
```

This should be in same directory as the parser.py It will automatically take the CSV file as input, train the neural network and generate the XML file as output.

5.11 Rover Vision

5.11.1 Computer Vision on the Trevor Rover

Author Mathew Kallada

Change Record

2014.06.24 - Document updated.
2014.05.21 - Document created.

Introduction

Purpose

Computer vision capabilities are crucial in a rover helping it to analyze and understand its enviroment. This document outlines the key features of the computer vision features of the Italian Mars Society's Trevor Rover.

Applicable Documents

- [1] – C3 Prototype document v.4
- [2] – OpenCV
- [3] – Software Engineering Practices Guidelines for the ERAS Project
- [4] – ERAS 2013 GSoC Strategic Plan
- [5] – Marscape Scenario
- [6] – TANGO distributed control system
- [7] – **'PyTANGO - Python bindings for TANGO'_**
- [8] – Minoru 3D Webcam
- [9] – V-ERAS
- [10] – EUROPA Planning Software
- [11] – Histogram of Oriented Gradients
- [12] – Principal Component Analysis
- [13] – Denisty-based scan

Glossary

CV Computer Vision

API Application Programming Interface

ERAS European Mars Analog Station

IMS Italian Mars Society

Overview

This module provides a series of computer vision operations designed for navigation and exploration with the Trevor Rover. These operations include target recognition and hazard detection ultimately satisfing the requirements for the Marscape scenario described in [5].

Design Considerations

Hardware Requirements

We will be using the Minoru3D Webcam and the RaspberryPi. In case of fast moving objects, we will need to optimize the speed of the Minoru+RPi.

Software Requirements

Object Recognition and Tracking

We will use scikit-learn for machine learning (predicting which objects it has previously seen), and OpenCV2 for image analysis (creating disparity fields, locating images).

Interface Requirements

Hardware Interfaces

We will be using the Minoru 3D webcam and the RaspberryPi for computer vision processing and AI planning and scheduling.

User Interfaces

To add human reasoning into the rover's decision making abilities, there will be an interface to allow operators to specify properties of previously seen objects.

Software Interfaces

An inputted image is sent to several tasks for processing. These tasks include object recognition and depth detection. Once we retrieve this information, we can infer conclusions such as hazards nearby, and finally send this data to the EUROPA system ([10]).

Performance Requirements

Ideally, the rover will want to interact and respond to it's enviroment in real time.

Software Design

High-level view of Object Recognition

This module takes a HOG representation ([11]) of each object on screen. Below, I have collected a series of objects and have shown ([12]) the dataset to two-dimensions (with PCA).

Each color represents a different cluster (found by DBSCAN as described in [13]). Each cluster represents an object on screen. This way, we can recognize objects we have seen earlier (the triangle is an object we are trying to predict).

Development and Progression

Standards Compliance

The guidelines defined in [3] should be followed.

Planning

A high level schedule is shown below.

- Milestone I: Finish Object Recognition & Target Tracking
- Milestone II: Enviroment Analysis

[Midterm Evaluation]

- Milestone II: Integrate with pyEUROPA
- Milestone IV: Integrate with the Waldo interface

5.12 Telerobotics

5.12.1 Software Architecture Document for Telerobotics

Author Siddhant Shrivastava

Change Record

25th May, 2015 - Document created.

26th May, 2015 - First Draft open for review.

14th June, 2015 - Added overall architecture diagrams and fixed links in the *Documents* section

18th June, 2015 - Added ROS-Tango Distributed System interaction diagram to the SAD.

21st July, 2015 - Updated with high-level diagrams, Limitations, and Interface descriptions.

Introduction

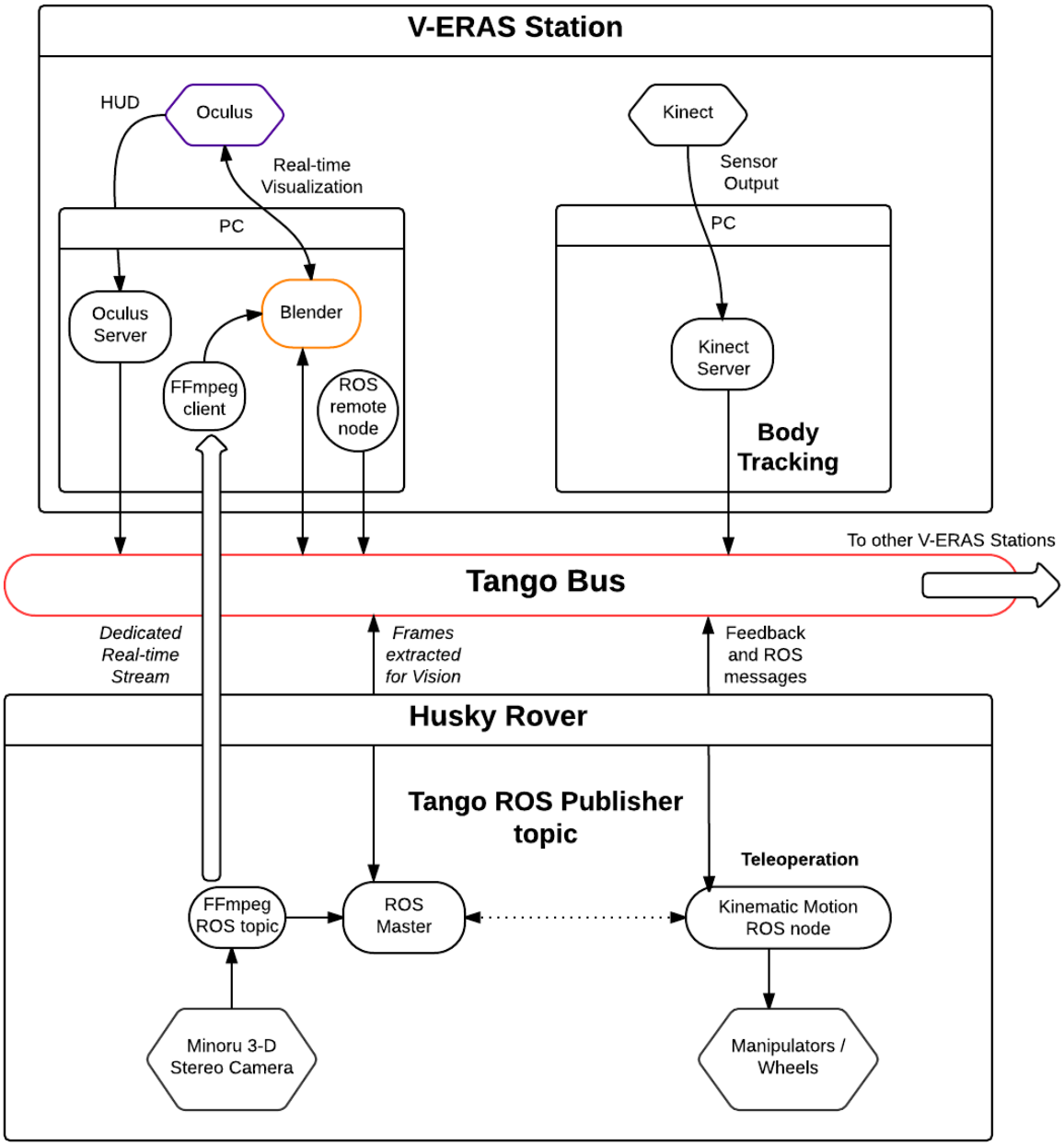
Purpose

The current system describes the architecture for teleoperative control of a Mars Robot rover's motion via human body-tracking. The robot dealt with in the current phase of the document is *Husky*, which is a rugged UGV (unmanned ground).

The purpose of this package is to provide a high-level Telerobotics interface for -

- Mapping Human body-tracking information to rover motion instructions
- Allowing real-time streaming of the rover's stereo camera-feed to the ERAS application
- Augmented reality interface obtained from the processing the rover sensor data

The following diagram describes the logical architecture of the Telerobotics system -



Scope

It is aimed that this requirement specification presents a **correct, understandable, efficient, and maintanable** Telerobotics interface for ERAS. The **scope** of this *requirements specification* spans all the robots that may be used in the future for **Telerobotics applications** *i.e.* **"all applications involving remote control, command and communication** (*C3*) **with a robot and a human"**

Applicable Documents

- [1] – C3 Prototype document

- [2] – Software Engineering Practices Guidelines for the ERAS Project
- [3] – Networking Subsystem of V-ERAS

Reference Documents

- [1] – ERAS 2015 GSoC Telerobotics Proposal
- [2] – TANGO distributed control system
- [3] – PyTANGO - Python bindings for TANGO
- [4] – Tango Setup
- [5] – Adding a new Server in Tango
- [6] – Robot Operating System
- [7] – Husky Unmanned Ground Vehicle

Glossary

AR Augmented Reality

C3 Command, Control, Communication

ERAS European MaRs Analogue Station for Advanced Technologies Integration

HUD **Heads-Up Display** – A display within the viewer's line of sight which indicates information. The Head-Mounted VR device acts as the HUD medium for this project

Husky Husky is a rugged, outdoor-ready **unmanned ground vehicle** (UGV), suitable for research and rapid prototyping applications. Husky fully supports ROS

IMS Italian Mars Society

Kinect Motion sensing input devices for movement, voice, and gesture recognition

ROS **Robot Operating System** - A collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

RTSP **Real-time Streaming Protocol**

TBD To be defined

TBC To be confirmed

Telerobotics Control of semi-autonomous robots from a distance

UGV Unmanned Ground Vehicle

V-ERAS Virtual European Mars Analog Station

VR Virtual Reality

Overview

Functional Requirements are of interest to **software testers**, **astronauts** and **users** planning to *interact with a robot* using this Telerobotics application of ERAS and add value to the **Telerobotics application** by giving feedback for individual features.

Non-functional Requirements are of interest to **Robotics researchers** and **Network Communication engineers** to improve the performance capabilities of the Telerobotics application.

User Interfaces are of interest to **UI/UX designers and users** of the **Telerobotics application**

Make an overview in which you describe the rest of this document the and which chapter is primarily of interest for which reader.

Architectural Requirements

This section describes the requirements which are important for developing the software architecture.

Functional requirements (Use Case View)

The **Telerobotics application** has the following requirements and use-cases -

- **Low-latency** transfer of information from the **Body-Tracking application** to the **Telerobotics application**
- **Fast and collision-free mapping** of bodytracking information to **rover's motion commands**
- **Account for variability and unreliability** in astronaut's body movements.
- **Outlier** body movements in a real-time stream of movements must be neglected.
- **Feedback** from the robot rover
- **Real-time** Streaming of rover information
- **Support for distributed and parallel architectures**
- **Semi-autonomous Teleoperation** - The rover navigates with a certain degree of automation which overrides manual commands in certain inconvenient circumstances.

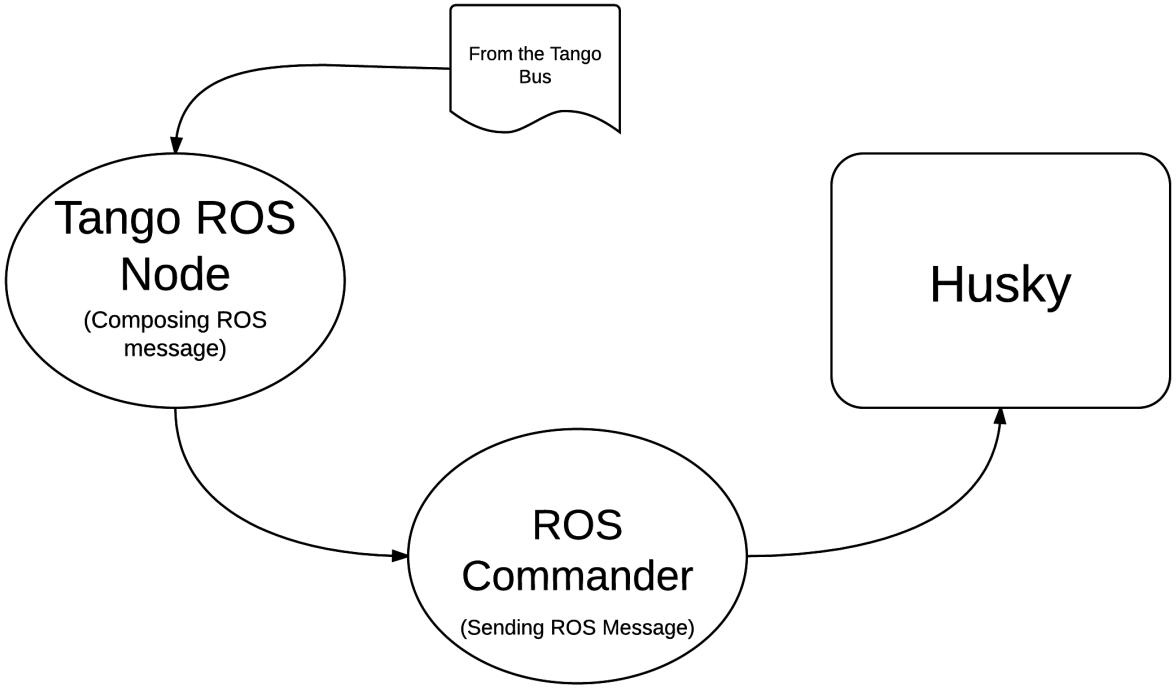
Non-functional requirements

The following are the non-functional requirements -

- **Wireless Communication via WiFi**
- **Real-time Video Streaming protocols**
- **Robotics Operating System**
- **Tango distributed controls system**
- Blender Game Engine
- Clearpath Husky UGV
- Microsoft Kinect

These non-functional requirements are already in place.

The underlying architecture is realized using these requirements:



Interface Requirements

User Interfaces

GUI (Graphical User Interface) The real-time video streams are displayed in the Blender Game Engine application. The GUI therefore is inherited from the V-ERAS Application. There is no separate GUI for telerobotic control. The interface with the Oculus Rift device is minimal and displayed on the twin Oculus screens. This augmented reality interface is essential for blending in the rover's world with the astronaut's world.

While controlling the navigation of a robot, the ROS Visualizer could be used to control the goal point and pose of a robot to override the Scheduler/Planner's output plans.

CLI (Command Line Interface)

- Telerobotics entails deployment of multiple servers from the command-line. The CLI for interactively creating and deleting servers looks like

```
Register format:  python setup-device.py <add,setup,register>

Unregister format:  python setup-device.py <del,delete,unregister>
```

- The command line interface is for accessing the ROS routines and override default application behaviour when needed.
- Command line interfaces are used to access the values of the individual Tango attributes of the Telerobotics server.

```
clk = DeviceProxy("C3/Robot/Diagnostics")

clk.read_attribute("battery").value
```

- The Fallback Keyboard Teleoperation interface has the following command line interface -

‘‘ Teleoperate Husky using Keyboard

Moving around:

u i o

j k l

m , .

q/z : increase/decrease max speeds by 10%

w/x : increase/decrease only linear speed by 10%

e/c : increase/decrease only angular speed by 10%

anything else : stop

CTRL-C to quit ‘‘

The Teleoperation Commands have been kept similiar to the ROS interface to minimize learning curve for ROS users transferring from other robots.

Future plans include a choice of left-handed controls for left-handed users. This is essential because Teleoperation should entail Human Computer Interaction principles of ease-of-use and discoverability.

API (Application Programming Interface) The Telerobotics Server is the organizer of all the Robotics information for the ERAS mission. Curently, it has the following interfaces -

- EUROPA-ROS Interface
- Bodytracking-ROS Interface
- Robot Diagnostics Server

  The diagnostic server provides all the essential information about the ROS-based robot. Any device in the Tango database can use this data in the following way -

  ```
  clk = DeviceProxy("C3/Robot/Diagnostics")
  clk.read_attribute("battery").value
  ```

  The Robot attributes currently supported are -

  1. Battery Capacity
  2. Battery Percentage
  3. Robot Uptime
  4. Current Drawn
  5. Voltage Drawn
  6. Robot Interior Temperature
  7. Error Interrupts

- Robot Information Collector Server

  The Collector subscribes to the various sensor outputs of the Robot -

  1. Robot Twist Information
  2. Robot Pose Information
  3. Robot Laser Data

4. IMU Calculations
5. Left Camera image
6. Right Camera Image

- Fall-back keyboard mode for Teleoperation

Hardware Interfaces

The Telerobotics hardware interfaces include -

- Husky robot
- Microsoft Kinect
- Minoru-3D Webcam
- Ubuntu Linux machine
- Microsoft Windows machine
- Graphical Processing Units

The robot can be extended to have a mobile manipulator arm for interacting with the environment.

Software Interfaces

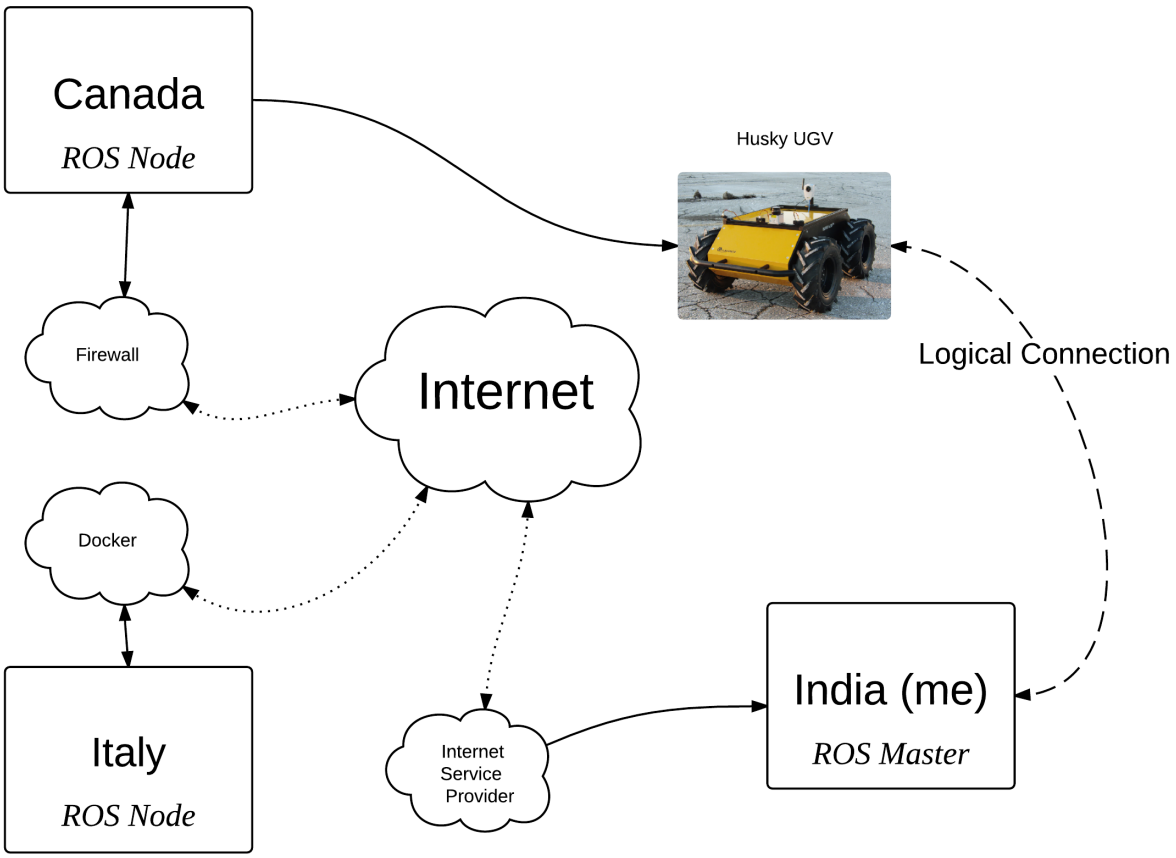
The Software Interface comprises of -

- Dictionary structures for mapping bodytracking information to robot motion
- Buffer structures for video streaming
- **FFmpeg** for high performance streaming and transcoding of data
- Basic Video Processing algorithms for the 3D stream
- Control structures for managing Tango and ROS messages
- Navigation structures for semi-autonomy for the rover
- Basic ROS structures (nodes, topics) for robot description and localization

Communication Interfaces

Communication forms a major part of this **command-control-and-communication** application.

- The **Telerobotics** application communicates with the body-tracking application over a shared Tango data bus.
- The communication with the robot is **wireless communication** from the ERAS station.
- Real-time stereo video stream is wireless communication over a **dedicated physical channel**
- The communication with the Oculus VR device is *wired communication* from the ERAS station.
- The software structures communicate via the Tango bus.
- Flow control among different software interfaces is realized by *buffer control structures*
- Being a collaborative effort from all over the world, it is necessary to create a **Virtual Private Network (VPN)**. The requirement can be summarized in the following diagram -



Performance Requirements

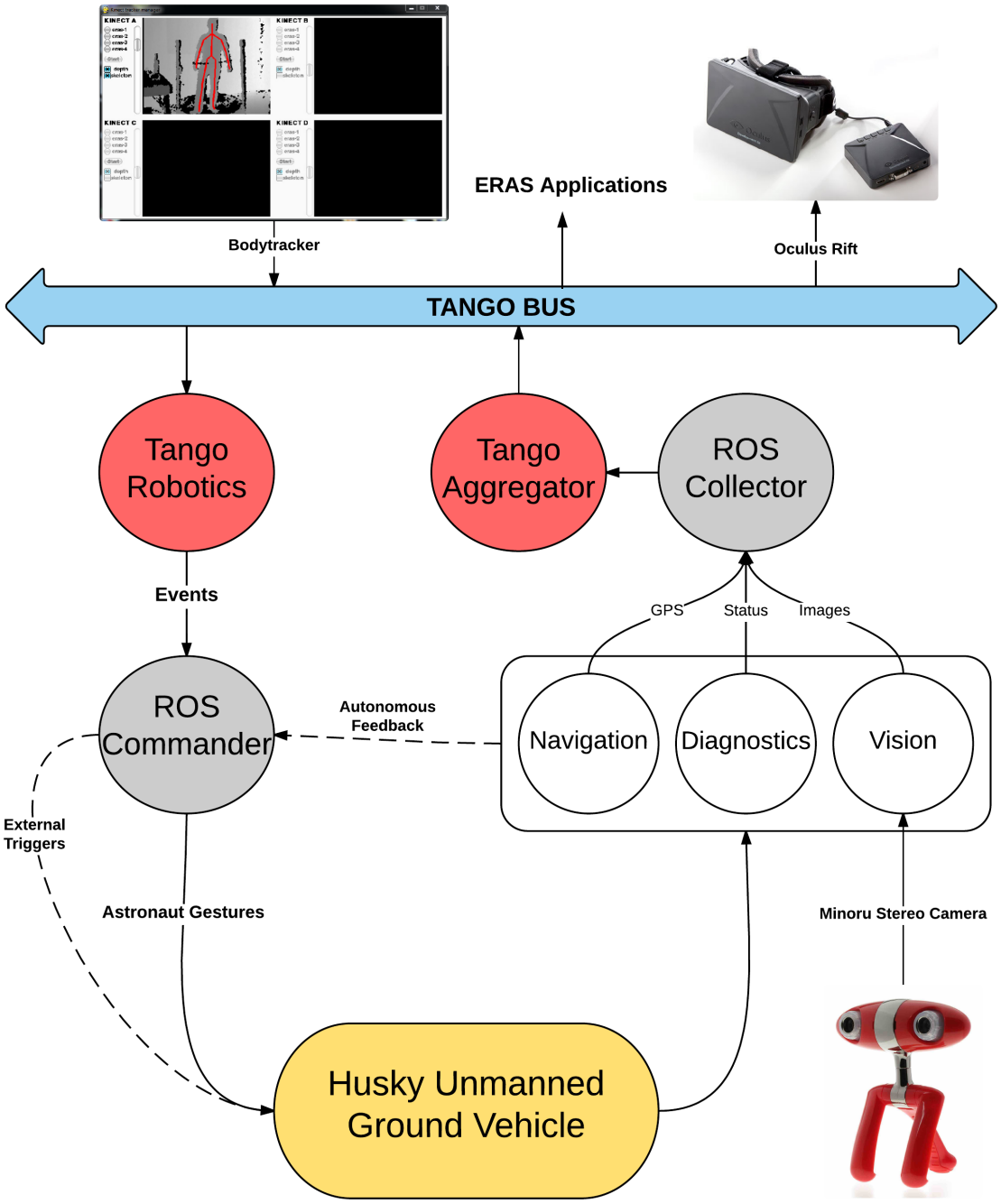
Telerobotics is a high-performance requirement application.

Real-time requirements need **at least soft-realtime guarantees* with **jitter** of the order of 100 microseconds. The 3D video streaming and AR applications are expected to be **hard-realtime** applications.

Logical View

Diagram

The entire ERAS project currently under development can be summarized logically in the following diagram -



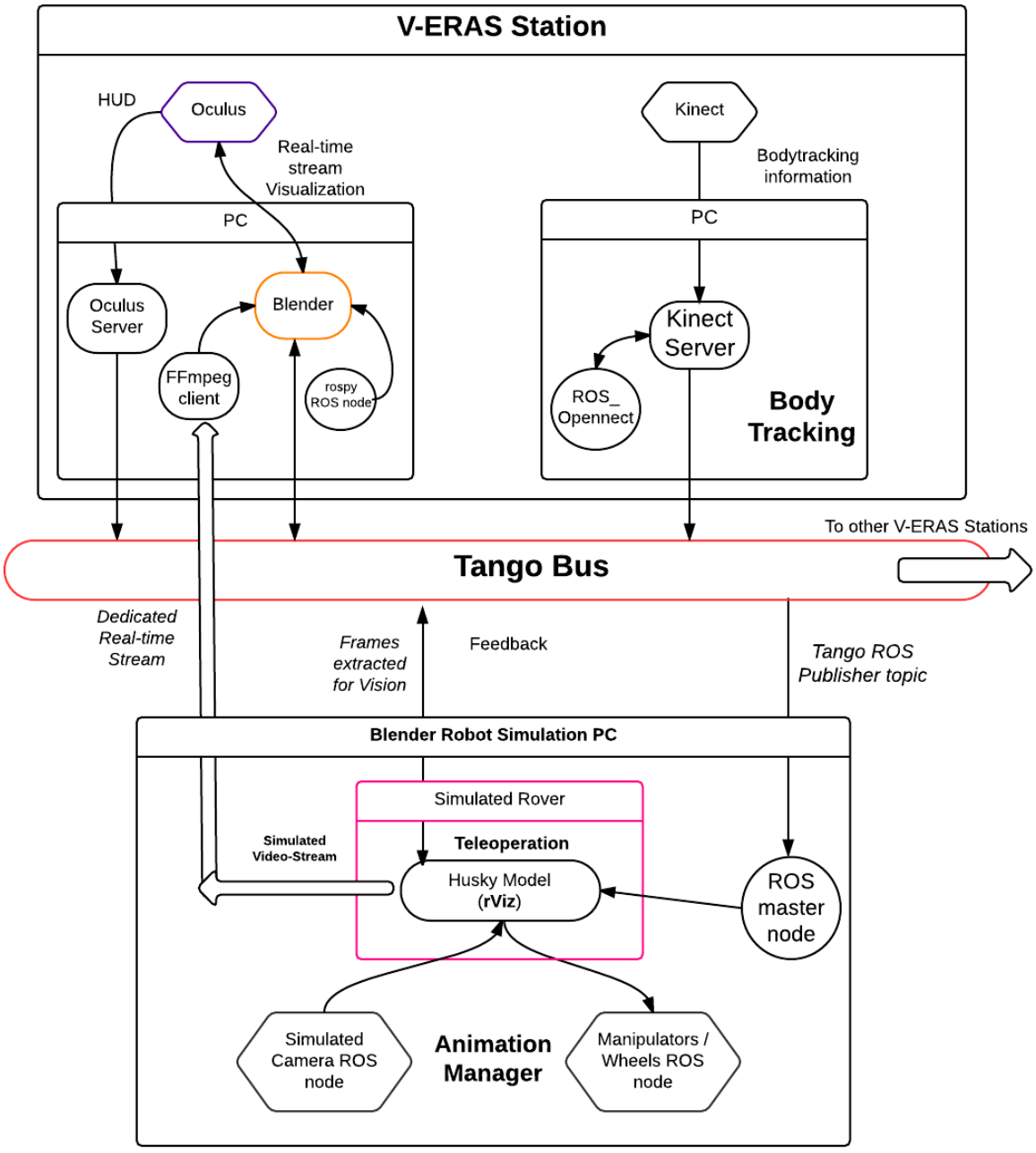
Layers

The ERAS software applications belong to the heterogeneous Distributed Control System (DCS) domain which can be represented as a layered architecture. This is a common design pattern used when developing systems that consist of many components across multiple levels of abstraction as in ERAS case. Normally, you should be developing components that belong to the Application layer

A simulated robot model is used in the initial parts of the project to ensure correctness and provide transparency in

results.

With a simulated rover, the architecture takes the form -



Development and Test Factors

Hardware Limitations

- The hardware of desktop machines are unsuitable for Stereoscopic Streaming applications.
- The Kernel is not configured to use multiple webcams. This results in memory-related errors.

- Currently, the Minoru 3D stereo camera is able to provide a **live stereoscopic feed** at 24 frames per second at a scale of 320x240.
- Telerobotics is provided as a Docker image. Docker uses the same process space as the host computer. Thus it is not suited for high-performance ROS calculations. Hardware is a limitation in using Virtual Environments.

Software validation and verification

- Exceptions are used wherever possible to check for all circumstances.
- Python's inbuilt *Profiler* will be used for estimating the areas which need optimization.
- Thorough Integration testing is planned since *Telerobotics* is a multi-component application. Current Integration supports Bodytracking Integration and EUROPA Integration.
- `ffprobe` is used to analyze the encoding and streaming performance of the Stereoscopic camera feed.
- `roswtf` is used to verify proper ROS behaviour
- The *unittest* library for **Python** will be used for all software testing.

Planning

12-week timeline (May 25 - Aug 24)

Week 1-2 (May 25 - June 8)

Teleoperative control of simulated Husky rover model

- **Week 1 (25 May - 1 June)** Create basic interface for mapping Kinect bodytracking information to teleoperation commands
- **Week 2 (1 June - 8 June)** Write a Tango device server to act as a Publisher/Subscriber ROS node in order to communicate to the Husky rover model. Unit Tests. First bi-weekly report.

Week 3-4 (8 June - 22 June)

Drive a real Husky model with the Telerobotics module

- **Week 3 (8 June - 15 June)** Employ the parallelly developed generic gesture control interface in another project for the telerobotics module
- **Week 4 (15 June - 22 June)** Extend the Teleoperative control to a real Husky mobile robot. Second bi-weekly report

Week 5 (22 June - 29 June)

Real-time Streaming from stereo camera to the V-ERAS application and Oculus

- Field Tests for the developed modules
- Integrate 3-D camera stream from the Minoru 3-D webcam with Blender and the Oculus VR Device
- Configure high-performance ffmpeg servers to communicate video streams for different Quality of Service (QoS) requirements

Week 6 (29 June - 6 July)

- Buffer Week
- Visualize the stereo camera streams in the V-ERAS application.
- Field tests continued for the developed modules

- Unit Tests for body-tracking Husky rover
- Performance evaluation of Minoru 3-D camera, ROS, BGE, and Oculus working together in V-ERAS
- Commit changes to V-ERAS
- Third bi-weekly report + Midsem Evaluations

Week 7 (6 July - 13 July)

- Oculus Integration with the stereo camera stream
- Extend the existing Oculus-Blender interface to display and update the incoming stereo video stream

Week 8-10 (13 July - 3 August)

Augmented Reality experience through a Heads-Up Display(HUD) for Oculus Rift using the Blender Game Engine

- **Week 8 (13 July - 20 July)** Use the positional-tracking feature of Oculus VR DK2 to set rover camera angle. Complete any remaining part of teleoperative control of Husky Fourth bi-weekly report.
- **Week 9 (20 July - 27 July)** Integrate Augmented Reality with the work done in week 1-6. Commit changes to V-ERAS /HUD
- **Week 10 (27 July - 3 August)** Develop a generic HUD API for any future application to use. Fifth Bi-weekly report.

Week 11-12 (3 August - 17 August)

Code cleaning, Testing, Documentation, Analysis, Commit, Polish existing functionalities

- Week 11 (3 August - 10 August): Network Performance Analysis, PEP8 compliance
- Week 12 (10 August - 17 August): Integration Tests, Documentation. Final commits and merging. Final report

5.12.2 Guide to ERAS Telerobotics

Author Siddhant Shrivastava

Change Record

20th August, 2015 - Document created.

Introduction

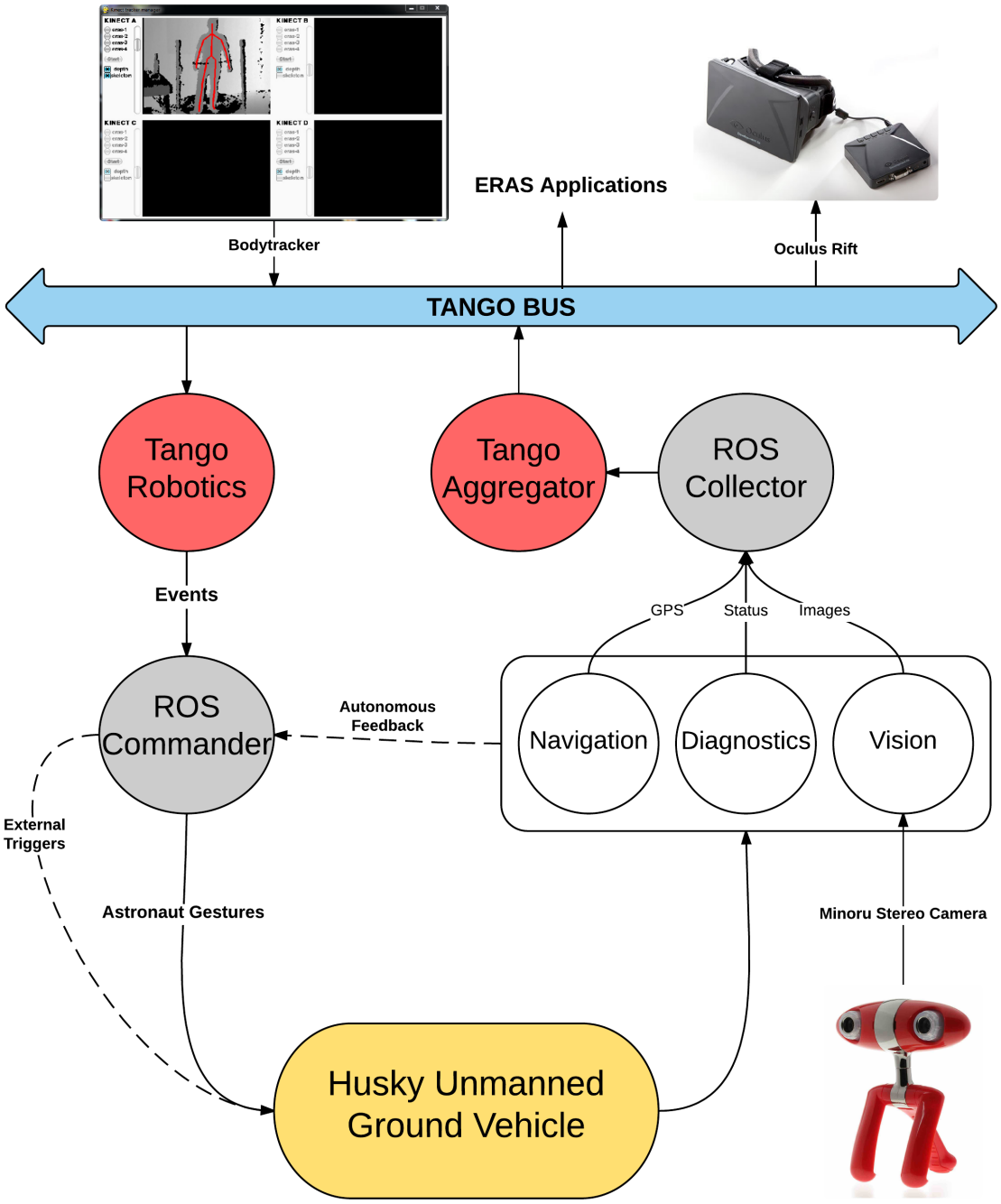
ERAS Telerobotics is a feature introduced in 2015 as part of the *Google Summer of Code Program* with the *Italian Mars Society* to support Telerobotics in Mars Missions. **Telerobotics** is a specialized field of **space robotics** that entails teleoperative control of Robots by Human beings *vis-a-vis* various channels -

1. Joystick/Keyboard Control
2. Gesture based shadowing of Astronauts
3. Another Robot's movements

ERAS Telerobotics uses *all three of the above* channels for the **Husky Unmanned Ground Vehicle** to test its Telerobotics platform. It is based on the Robot Operating System (ROS) for its high-performance, ease-of-use, distributed nature and open community.

This document is intended to be a walkthrough of the current state of ERAS Telerobotics.

On a high level, Telerobotics is supposed to look like this -



Setup of ERAS Telerobotics

The best way to get started is to use the existing Docker image

Understanding ERAS Telerobotics

Bulk of the Telerobotics platform makes use of the **Python** programming language. The three major libraries that this project is based on are -

- Robot Operating System
- Tango-Controls
- FFmpeg

Interfaces with other ERAS components

Telerobotics is a highly integrated application and seamlessly integrates with other important ERAS components such as -

- Bodytracking with Microsoft Kinect

  This is important because the aim of Telerobotics is to provide a way for astronauts to drive around the UGV on the Martian terrain. The **step estimation** data from the ERAS Bodytracking server is used to drive around the UGV in real-time. It is implemented in `telerobotics-bodytracking.py`.

- EUROPA Scheduler and Planner

  The scheduler is intended to be an astronaut's mission field guide. The rover is also a part of the crew and needs to plan its trajectory around the Martian terrain for scientific experiments. This requires Telerobotics to interface with the existing ERAS Planner and navigate the Robot safely to the desired destination. The implementation is provided in `europa-navigation.py`

The functional requirement of ERAS is `control-command-communicate`. Telerobotics realizes this in all aspects.

Diagnostics server

It is important for **Mission Control** to realize the real-time conditions of the Husky Robot. It is also necessary to use the Robot's diagnostic information for various tasks such as resource planning and path optimization. To facilitate this, the Robot Diagnostic server is created. Implementations are provided in `robot-diagnostics-server.py` and `robot-info-collector.py`.

Fallback Interfaces

Telerobotics is a network-intensive high-performance application. In the unintended case of breakdown of the Bodytracking server or the Tango-Controls system, it is necessary to keep controlling the Husky rover. Keyboard Teleoperation is suggested to help in this case. An implementation is provided for right-handed astronauts in the file `teleoperation-keyboard.py`.

Streaming

Streaming is a necessary requirement for ERAS. The video feed from the stereoscopic camera mounted on the Husky rover is streamed in real-time to **Mission Control** which is then processed and sent to the astronaut who can decide the future actions of the rover.

The implementation is provided in the `eras/servers/telerobotics/streams/` directory.

5.12.3 Machine Configuration for Telerobotics-Bodytracking Interface

Author Siddhant Shrivastava

Change Record

13th August, 2015 - Document Created

14th August, 2015 - Replication Details added

15th August, 2015 - First Draft open for review

Machine Setup

Successfully tested on a **single** laptop with the following configuration -

- **Ubuntu 14.04 amd64** *host OS*
- **Windows 7 Ultimate 64-bit** *Virtual Machine*
- **VMware Workstation 11**
- *Hardware Specifications* - 8GB RAM, intel Core i7 x64
- *Networking Specifications*: `NAT-Only` mode between host OS and VM

Windows Configuration

The following instructions are based and were tested on Microsoft Windows 7 64-bit. After having installed the operating system in VMware, setup all drivers needed to use the machine in the right way.

Installing Python

To support PyKinect, you must install *Python 32-bit 2.7.10*. To install this version of Python, use this link.

It is recommended to install this version of Python in `C:\Python27_32bit\`.

To be able to excute Python from a command line, you must add the installation folder path to the `Path` variabe in Windows. In order to do this, follow these steps -

- Open **My Computer**
- Right-click and select *Properties*
- Choose *Advanced system settings* from the options on the left panel
- A menu should appear. Click on the *Environment Variables* button
- Add the Python path to the Path variable

You will also need to set the `PYTHONPATH` variable likewise. Create a new environment variable with the name **PYTHONPATH** and the value of `C:\Python27_32bit\`

In order to use Python 2.7 on Windows, Visual C++ compiler (*VCforPython27*) must also be installed. It can be downloaded from this link.

Setting up Kinect Tools

In order to use Kinect on Windows, Kinect Developer Kit and the SDK need to be installed

The installation process will be quite long, and it will probably require some reboots. After that, you have to install (in this order):

- Kinect SDK
- Kinect Developer Kit

Installing PyKinect

By following the above instructions for installing a Python package from Visual Studio, or simply using `pip` on a command line terminal, install the package `pykinect`.

Installing additional Python packages

Before continuing, you need also to install the following Python packages:

- *numpy*: required to install PyTango; it can be installed with `pip install numpy`
- *PyTango*: download the last 32-bit version for Python 2.7, available from here
- *pgu*: download from here and install it with `pip` (follow the above instructions, as if the package you download is a .whl file)
- *VPython*: download the automatic installer for Visual Python from here (chose the x86 version, not the x64 one!)

Note: As a source to find a lot of Python libraries, packed as Windows installers or as `.whl` files, you can refer here

Installing Tango

Go to this page and select the binary distribution for Windows x64. Download and install it.

After the installation, you will be able to access to a lot of utility and tools to get information about Tango and the device servers (e.g. *Jive*). To use them, you must install *Java for Windows*; you can get it from here

Configure Tango Host

To be able to get all Tango informations, you need to specify the address of the Tango host. Assuming that it is 198.168.1.100:10000, open the command line and type:

```
set TANGO_HOST=192.168.1.100:10000
```

Installing Java

Dowload the latest Java runtime from the Oracle website. The specific page is located here.

Ubuntu configuration

Install Ubuntu 14.04.2 on the computer.

VMware Setup

Install **VMware Workstation 11**. Follow the Ubuntu instructions as shown on this page. Set up Windows on this VMware setup and follow the instructions for Windows.

ROS Setup

Install **ROS Indigo** on Ubuntu using the following instructions -

- Setup the Sources list

  ```
  sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release
  -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
  ```

- Set up keys

  ```
  sudo apt-key adv --keyserver hkp://pool.sks-keyservers.net --recv-key
  0xB01FA116
  ```

- Installation

  ```
  sudo apt-get update
  ```

  ```
  sudo apt-get install ros-indigo-desktop-full
  ```

  ```
  sudo apt-get install synaptic
  ```

- Initialize ROS

  ```
  sudo rosdep init
  ```

  ```
  rosdep update
  ```

  ```
  echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
  ```

  ```
  source ~/.bashrc
  ```

- From the Dash, open `Synaptic Package Manager`
- Search for `husky indigo` and install all the packages prefixed by `ros-indigo-husky-`

Tango Setup

Follow the instructions in the ERAS documentation to set up Tango.

This should set up everything on the Ubuntu side.

Networking Setup

Tango should be appropriately configured on both sides. In the working setup, the Tango Master is configured to be the Ubuntu machine.

Set the **Networking Mode** of the Virtual Machine to be `NAT-Only`. Observe the output of `ifconfig` in the host OS. The Virtual Machine should have created two additional interfaces - `vmnet1` and `vmnet8`. Use the interface whose subnet matches the interface in the output of `ipconfig` in the Windows Virtual machine.

Configure Tango Host to the IP Address corresponding to the `vmnet` interface which matches the subnet information. Try running `jive` on both the Operating Systems to check for consistency.

Replication instructions

Once the setup and configuration is complete (as discussed in the previous sections), run the following commands on the host OS -

First configure the Tango Database server to use the Bodytracking device with the following attributes -

- **Device Name** - `eras-1`
- **Device Class** - `PyTracker`
- **Canonical Name** - `c3/mac/eras-1`

On the Bodytracking (Windows) machine -

- Open a Command Prompt window and execute -

  ```
  python tracker.py eras-1 --sim <json_file_location>
  ```

- Open another Command Prompt window and execute -

  ```
  python visualTracker.py eras-1
  ```

This should bring up the skeleton model which is updated in real-time.

On the Telerobotics (Ubuntu) machine -

`cd` to the `src` directory of the `Telerobotics` ERAS server.

Open **three** terminals -

- In the first terminal, run

  ```
  roslaunch husky_gazebo husky_empty_world.launch
  ```

- In the second terminal, run

  ```
  roslaunch husky_viz view_robot.launch
  ```

- In the third terminal, run

  `python telerobotics-bodytracking.py`

5.12.4 Instructions for setting up a Real-Time Streaming Server

Author Siddhant Shrivastava

Change Record

21st June, 2015 - Document created

26th June, 2015 - Added single camera streaming Instructions

10th July, 2015 - Added **'Explanations'_** section

15th July, 2015 - Added FFmpeg command explanation for single camera

19th July, 2015 - Added *Debugging* section

20th August, 2015 - Added details and links.

FFmpeg Setup

Download from source Generally, `FFmpeg` comes with `v4l2-utils` support built-in. This is important for controlling video devices on Linux-based Operating Systems.

Steps to get the setup working

1. Configure a ffmpeg server on the remote machine.
2. For this, copy the `ffserver.conf` file from `eras/servers/telerobotics/streams/single_camera` into the `/etc/` directory of the remote machine.
3. Start the server via the ffmpeg command specified in webcam.sh
4. The server reads from `/dev/video0` and serves it on the listening port.
5. The client machine has the Blender scene and script running.

Testing and Explanation

Single Camera

'' ffmpeg -v verbose -r 30 -s 640x480 -f video4linux2 -i /dev/video0 http://localhost:8190/webcam.ffm ''

- This command ensures a **single camera stream** explained by the various options of FFmpeg. The *-r* option specifies a framerate of 30fps (frames per second from the input device) and *-s* option specifies an input resolution of 640x480 from the camera.
- The input device is `/dev/video0` which is a `video4linux2` supported device.
- The output device is the **Feed** which is identified by the `ffserver.conf` configuration.

Debugging

In case of dropped frames or memory-related issues, it is quite possible that the problem is hardware related. In this case, the `v4l2-ctl` tool is quite useful. Some important commands while troubleshooting such issues are -

- `v4l2-ctl --list-devices`
- `v4l2-ctl --verbose <any_command>`
- `v4l2-ctl --list-ctrls <device name>`

Knowing the capability of the camera in hand goes a long way in identifying the usage profile of the camera as a video device. Cameras have a fixed set of frame rates supported by the hardware. These must be kept in mind before streaming at an intended frame rate.

`ffmpeg` has its own levels of verbosity which can be changed while streaming. `ffprobe` can be used for further performance analysis. `ffplay` is a good tool to check the output quality of a stream.

5.12.5 Setting up and Callibrating the Minoru 3D Camera

Author Siddhant Shrivastava

This manual has been adopted partially from here.

Change Record

20th July, 2015 - Document Created

20th August, 2015 - Draft Open for review

The Minoru camera is recognized as a `v4l2` or `Video4Linux2` device.

The Minoru is UVC compilant, and therefore very easy to use on a GNU/Linux operating system. Plug in the camera, then open a command shell and type:

```
ls /dev/video*
```

This should display two extra video devices.

After setting up `v4l2stereo` from the `streams/Minoru3D` directory, run the following command -

```
v4l2stereo -0 /dev/video1 -1 /dev/video0 --features
```

So, once you have established that the cameras are working the first thing to do is calibrate them using the –calibrate option. This uses the OpenCV stereo camera calibration routines in order to obtain the optical parameters. First, print out a calibration pattern, which consists of a checkerboard pattern, and mount it on a rigid backing such as cardboard or wood. Then type:

```
v4l2stereo --dev0 /dev/video1 --dev1 /dev/video0 --calibrate "6 9 24"
```

The first number of the calibrate option is the number of squares across, the second is the number of squares down, and the third is the size of each square in millimetres. The order of the dimensions should correspond to how the calibration pattern is presented to the cameras. The video below shows the procedure.

Optionally the number of calibration images which are gathered can be set. By default this is 20, but higher numbers should give a more accurate result.

```
v4l2stereo --dev0 /dev/video1 --dev1 /dev/video0 --calibrate "6 9 24"
--calibrationimages 60
```

Once camera calibration is complete the parameters are automatically saved to a file called calibration.txt. Normally when running v4l2stereo the program will search for this file in the current directory, but optionally you can also specify it as follows:

```
v4l2stereo -0 /dev/video1 -1 /dev/video0 --calibrationfile calibration.txt
```

To test the image rectification:

```
v4l2stereo -0 /dev/video1 -1 /dev/video0
```

If the rectification is good you should notice that when the left and right images are placed side by side the rows of both images correspond. If there is any vertical displacement it is possible to manually alter this, either by editing the vshift parameter within calibration.txt or by using the –offsety option.

5.12.6 Docker Container Instructions for Telerobotics

Author Siddhant Shrivastava

Change Record

25th May, 2015 - Document created.

Introduction

From the Docker website -

> Docker is an open platform for building, shipping and running distributed applications. It gives programmers, development teams and operations engineers the common toolbox they need to take advantage of the distributed and networked nature of modern applications.

The Telerobotics Docker Image is a Ubuntu 14.04 image configured with Robot Operating System (ROS) packages, Tango-Controls, and Gazebo models to use Telerobotics out-of-the-box. ERAS is also provided as part of the image.

Docker Installation

Install Docker for your Platform by following instructions from this site

Setting up the Telerobotics Image

Pull the `sidcode/ros-eras` image the Docker Hub by running the following command -

```
docker pull sidcode/ros-eras
```

Note that a high-speed Internet connection is required as the image is around 4 GB in size.

Using the Telerobotics Image

Most of the features of the image can be harnessed using the following command -

```
docker run -i -t sidcode/ros-eras bash
```

In order to run `Gtk-based` applications, one needs to share the display with the host machine. To achieve this, start the image with the following command -

```
docker run -i -t -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix
sidcode/ros-eras /bin/bash
```

This should provide access as a root user to the Telerobotics image.

Start the ssh-server which is to be used for ROS GUI applications -

```
./etc/init.d/ssh start
```

A Linux group called `eras` and a user called `eras` is configured in the image. The details are as follows -

Username - `eras` Password - `eras`

View the Network interface information by running `ifconfig`. The IPv4 Address corresponding to the `eth0` interface is the one that is to be used to connect to the Docker image.

From the Physical machine, start a terminal and `ssh` to the Docker image as follows -

```
ssh -Y <docker-IP> -l eras
```

Now we are all set to use the Telerobotics Docker Image.

Follow the instructions in the rest of the manuals for the next steps.

5.12.7 Primer to understand the Robot Operating System (ROS) for ERAS Robotics

Author Siddhant Shrivastava

Change Record

20th August, 2015 - Document created.

5.13 Web Plotter

5.13.1 Web Plotter Setup

The Web Plotter reads data from any Tango server and plots on a graph on a web page the values of all the attributes provided by the server.

Running the server

To run the web server use:

```
$ python webserver.py
```

This will start running a web server at <http://localhost:8080>.

From the web page you can access any tango device by adding its name at the end of the URL, e.g. <http://localhost:8080/C3/neurohs/epoc1>.

If the Tango device is up and running, a list of values and a graph should be displayed.

Usage

The Web Plotter will keep reading data from the Tango server and plot them in real time. You can select which attributes are plotted by checking/unchecking them in the list of attributes.

It is also possible to select how often the graph is updated and how many values are displayed by selecting the desired values under the graph.

Troubleshooting

If the list of attributes and the graph are empty, make sure that the name in the URL is correct and that the server is up and running. After you started the server you will have to refresh the page.

If the values stop being updated, verify that the server is still running and then try to refresh the page.

If the plotting is slowing down the browser/pc, try to select a longer update interval or a smaller number of displayed values.

Developers' docs:

CHAPTER 6

Tango Setup

This page documents how to install Tango on Ubuntu. These instructions have been tested on Ubuntu 13.10.

6.1 MySQL

First you have to install MySQL:

```
sudo apt-get install mysql-server mysql-client
```

During the installation of **mysql**, you can leave the root password empty. If you specify a password you will have to enter it later, while installing **tango-db**.

6.2 Tango

After installing MySQL you can install tango:

```
sudo apt-get install tango-common tango-db tango-starter tango-test python-pytango libtango8-doc libt
```

If you are still using Ubuntu 13.04 you should use `libtango7-*` instead of `libtango8-*`.

During the installation of **tango-common**, enter `pcname:10000` as TANGO host, where *pcname* is the name of your machine.

During the installation **tango-db** select:

- Configure database for tango-db with dbconfig-common? [yes]
- Password of the database's administrative user: [leave empty]
- MySQL application password for tango-db: [leave empty]

Next you have to install **libtango-java**. If this is not available in the repositories, you can add the launchpad repository manually:

```
sudo add-apt-repository 'deb http://ppa.launchpad.net/tango-controls/core/ubuntu precise main'
sudo apt-get update
sudo apt-get install libtango-java
```

If you get this warning during the `sudo apt-get update`:

```
W: GPG error: http://ppa.launchpad.net precise Release: The following signatures couldn't be verifie
```

and/or this warning during the `sudo apt-get install libtango-java`:

```
WARNING: The following packages cannot be authenticated!
    libtango-java
Install these packages without verification [y/N]?
```

you can fix it by doing:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys A8780D2D6B2E9D50
sudo apt-key update
sudo apt-get update
```

Note: Currently the repository doesn't have packages for *quantal/raring/saucy*, so it is necessary to specify *precise* even if you are running a more recent version. If you use `sudo add-apt-repository ppa:tango-controls/core`, your version of Ubuntu will be selected, and you will have to change it to *precise* manually.

You should now be able to lauch **astor** and **jive** from the terminal.

For more information see the Troubleshooting section below and this link: http://www.tango-controls.org/howtos/binary_deb

6.3 Troubleshooting

At the end of the installation, you should have two tango processes running:

```
$ ps aux | grep tango
tango    15451  1.0  0.1  71800  9820 ?        Sl   10:56   0:00 /usr/lib/tango/Starter pcname
tango    21109  0.0  0.1  94396 10752 ?        Sl   May18   0:20 /usr/lib/tango/DataBaseds 2 -ORBendP
```

The first one is from **tango-starter**, the second one from **tango-db**.

If you don't see them, you can try to reinstall these two packages, using:

```
sudo apt-get remove package-name
sudo apt-get install package-name
```

If you are reinstalling **tango-db** and/or if you get this error:

```
An error occurred while installing the database:
ERROR 2002 (HY000): Can't connect to local MySQL server through socket
'/var/run/mysqld/mysqld.sock' (2)
```

you have to select <Yes> when asked "Deconfigure database for tango-db with *dbconfig-common*?".

If you are still having problem you can try the following things:

- check that `/etc/tangorc` contains `TANGO_HOST=pcname:10000`;
- try to set the environment variable `TANGO_HOST`. You can also add the following line to your `~/.bashrc` to make it automatic (you will have to restart bash):

```
export TANGO_HOST=pcname:10000
```

- try to (re)start **tango-db** and/or **tango-starter** manually using:

```
sudo /etc/init.d/tango-db start
sudo /etc/init.d/tango-starter start
```

- if it still doesn't work try completely removing the packages:

```
sudo apt-get purge tango-db tango-starter
```

  - Deconfigure database for tango-db with dbconfig-common? [yes]
  - Do you want to purge the database for tango-db? [yes]
  - Password of the database's administrative user: [leave empty]

  and then reinstalling them:

```
sudo apt-get install tango-db tango-starter
```

  - Configure database for tango-db with dbconfig-common? [yes]
  - Password of the database's administrative user: [leave empty]
  - MySQL application password for tango-db: [leave empty]

6.4 Fix for Ubuntu 13.04

If you are running on Ubuntu 13.04 you have to install this patch in order to avoid a Segmentation fault (core dumped) at every python server run. Install first libboost-python-dev:

```
sudo apt-get install libboost-python-dev
```

Download the patch from:

https://pypi.python.org/packages/source/P/PyTango/PyTango-7.2.4.tar.gz

untar and install it:

- $ tar xzvf PyTango-7.2.4.tar.gz
- $ cd PyTango-7.2.4
- $ python setup.py build
- $ sudo python setup.py install

6.5 Adding a new server in Tango

To register a new server run **jive**, select `Edit -> Create Server` and provide:

- the executable name and the instance name (ex: legorcx/c1b8)
- the Class name
- the device name in the format: `C3/subsystem/device`

Then start the java/python/C++ application always providing the instance name, example:

```
python legorcx c1b8
```

and the Class properties will be automatically filled in the database

CHAPTER 7

Software Engineering Practices Guidelines for the ERAS Project

This document intends to provide an overview of the Software Engineering (SE) practices for the ERAS Project Software. The main areas covered are: Coding Standards, Version Control, Change Management, Static and Dynamic Verification and Documentation.

If an answer cannot be found here, use the existing code as an example or ask on the mailing list: eraspro-ject@googlegroups.com

7.1 Reference Documents

- [1] PEP 8 – Style Guide for Python Code
- [2] Mercurial – Mercurial Home Page
- [3] Hg Init – A Mercurial Tutorial
- [4] pep8 – Python style guide checker
- [5] radon – Python static code analysis tool
- [6] unittest – Python Unit Testing Framework
- [7] coverage – Code coverage measurement for Python

7.2 Coding Standards

ERAS software must be readable, easy to maintain and as least error prone as possible. It will have to use official standards as much as possible. To this purpose, coding standards should be applied systematically. Coding standards range from proper code documentation to file naming conventions and in general help in preventing a certain category of software bugs. They do not address specific implementation details like algorithms or programming methodologies. The application of coding standards is deemed crucial to any modern software undertaking.

The main coding standards reference for Python can be found in [1]. Here follow the most important recommendations:

- don't write more than 80 chars per line
- use spaces around operators like =, +, ==
- don't put spaces between the function name and the (
- don't write more statements on the same line.
- use `CamelCase` for classes names, `lowercase` or `lower_with_underscores` for the rest

- avoid `mixedCase`.
- use clear names for variables, functions, and classes.

7.3 Version Control

Software developers use a Version Control tool to baseline configurations, prepare releases and to deal with synchronous update of items. For release preparation in particular, code freezing or "tagging" is required i.e. a simple notification by the package responsible, that a certain version of the package has achieved its planned objectives and may be taken from the repository for the release. The Version Control tool must enable:

- identification of items
- traceability of changes (who did what, when)
- accessibility of previous configurations for any item

Both source code and documentation must be subject to strict version control. The tool chosen to support ERAS Software Version Control is Mercurial (HG). Please refer to [2] and [3] for a more exhaustive description of the tool.

7.3.1 Configuration Management Item: Software Package

A software package is a piece of software (code and documentation) able to perform functions and having an interface available to an external user to access the functions provided. Technically a package is a way to organize functions in homogeneous groups. The interface hides the implementation and system dependencies from the user. Managerially the package is the basic unit for planning, software project control and configuration control. There is no rule to define how big a package shall be. Common sense and programming experience should be enough to identify what can be gathered and treated as a unique item. Each package shall have one responsible person, who will be entitled to delegate activities on the package but shall retain responsibility at all times.

7.3.2 Package identification

A package is identified by its name and its version mnemonics. The version identification model is defined with a mnemonic comprising at least two numbers for the major (v) id and the minor (r) id (a third number can be present representing a patch (p) id). The increase of these two numbers during a development has to follow these rules:

- if the changes in the libraries of package A are not backward compatible (it is necessary to change something in packages which are using A) then the major id number must be increased on one unit.
- if the change does not imply any modifications in the other packages, the minor id number must be increase by one unit.

A patch version is sometimes useful to isolate a change in the code during a temporary test for instance. The package version is then labeled `vXrYpZ` and is not meant to be included into an ERAS software release. This versioning identification model is very similar to the Semantic Versioning Specification (SemVer). While dealing whit global software releases which bundle the whole set of ERAS Software packages, the Semantic Versioning Specification (X.Y.Z) will be followed.

7.3.3 Package structure

Each package will correspond to a TANGO server or client and should contain the following:

- an `xxx.py` file that implements the functionality of the server, but it's independent from TANGO.

- an `xxx` file that implements the actual TANGO server/client by importing `xxx.py` and defining TANGO-specific classes. This file should not have an extension and should be set as executable, otherwise TANGO won't be able to use it.
- a `test_xxx.py` that includes unittests for `xxx.py`.
- **a `doc` directory containing:**
  - `index.rst` that contains the toctree for the server docs.
  - `xxx.rst` that includes generic documentation for `xxx.py`.
  - `sad.rst` Software Architecture Document for `xxx.py`.
  - `swum.rst` User and Maintenance Manual for `xxx.py`.
- possibly additional files required by the server/client.

7.3.4 HG Repository structure

The `servers` dir will contain all the TANGO servers/clients. Every server/client has its own directory. The `doc` directory contains the general documentations, such as instructions about TANGO installation and setup, templates, etc.

7.3.5 Branches and heads

Don't push new branches or heads on the repository. Before committing make sure that there are no incoming changesets (`hg incoming`), and if there are use `hg pull` to pull them. If you accidentally commit before pulling and create a new head (you can check with `hg heads .`), you will have to use `hg merge` and `hg commit` to merge the heads before being able to push.

7.3.6 Commits, commit messages and tags

One commit per issue. Adding a new class with tests and documentation is OK. Fixing a bug and adding a new feature in the same changeset is *not* OK. Fixing two unrelated bugs or adding two unrelated features in the same changeset is *not* OK. "Work in progress" changesets should be avoided – the code should work at every changeset (it's OK to make a commit for a basic but still incomplete class that works, and add more features afterwards).

Before committing use `hg diff` and `hg status` to make sure that what you are committing is OK and that all the files are included and that there are no unrelated changes. If necessary you can update the `.hgignore` file.

Descriptive, non-empty comments are required for each commit. They must be complete and readable, making reference to issue entries when applicable and explaining briefly what the changeset does in the present tense. `"Implement new feature XXX."`, `"Fix bug XXX by using YYY."`, `"Add tests for the XXX class."`, `"Improve documentation for XXX."` are *good* commit messages. `Fix a bug.`, `fix a bug`, `improve the code` are *bad* commit messages.

Before pushing into the central repository your changeset must be tagged using the version identification model (vXrYpZ) previusly mentioned.

7.4 Change Management

In ERAS we will be using the Issue Tracker embedded into Bitbucket as Change Management tool. The tool will allow internal or external users of the ERAS Software to report problems/errors, submit change requests or to require clarification on software, hardware or documentation.

Here we briefly summarize the basic workflow of the system:

- Issue submitted and all relevant people add themselves as monitoring users
- Notes added by any user
- A Responsible Person is assigned for the issue
- Responsible works on issue
- Responsible add a final remark on the issue and software manager close it.

7.5 Static and Dynamic Verification

7.5.1 Code Inspections (Static Verification)

Adoption of approved coding standards must be periodically monitored and this can be achieved by inspections of the code. Both manual (human) and automatic inspections are possible. Source code will be subject to scrutiny (at package level) by suitable software tools which will rate the code according to compliance to predefined guidelines. Human inspections will be done for certain packages of special relevance or for those code segments which exhibit a remarkably high algorithmic complexity. Tools measuring standards metrics (like McCabe cyclomatic complexity) will be used to identify which software packages are more prone to exhibit faulty behavior, and should therefore be tested more thoroughly.

More specifically for the Python language, developers could use the tool `pep8` [4] in order to check compliance with the PEP 8 standards before pushing on the main repository. Once notified of a package release, Software Mentors will make use of `pep8` [4] and `radon` [5] to identify the code segments to be reviewed, review them and provide feedback to developers. Developers will then commit required modifications.

7.5.2 Testing (Dynamic Verification)

The amount of software faults or incorrect behaviors in the ERAS software must be kept to a minimum and the system must be validated, i.e. it must be guaranteed that it is working according to its specifications. The application of a consistent testing scheme and the diffusion of a "testing culture" will help to achieve this goal. Although the developer is encouraged to delegate test code writing to someone else, it is his/her final responsibility to make sure that his/her package has achieved a sufficient degree of testing. A formal testing scheme will be adopted to ensure developers push only packages, which have been previously tested. During integration software packages versions may be rejected if they do not provide sufficient testing certification. Developers are required to start working on their test suites as a result of design, prior to implementation (i.e. use test-driven development). The responsible for each software subsystem will make sure that two types of regression tests are performed:

- Unit tests: the smallest unit is tested under isolation. If needed, the behaviour of other code units interacting with the unit under test will be mimicked by building stubs.
- System tests: the system (or subsystem) as a whole is tested against its functional specifications

Tests should be defined for each release and based on the Use Cases which have been implemented. This will permits to trace the requirements through the whole process. All test procedures must be fully automatic or, when this is not possible, based on a detailed checklist.

For development in Python:

- unit tests should be developed using unittest.
- In order to determine the amount of code coverage of each test suite and thus its sensibility the use of the `coverage` [7] tool is mandatory. At each package release the obtained coverage report must be provided.

7.6 Documentation

The appropriate documentation has to be written together with the code. We can individuate those levels and types of documentation:

1. Comments inside the code
2. Release Notes
3. Manuals

7.6.1 Release Notes

For every major release of the ERAS Software, the Release Notes for all the ERAS applications and programs has to be produced. It is up to the ERAS software manager to organize the Release Notes, their delivery with the ERAS distribution and publishing on the web.

7.6.2 Manuals

Software documentation must cover the entire software process, from the Software Architecture definition phase (Software Architecture Document) to the User documentation (Software User Manual and Software Maintenance Manual). The documents should go under configuration control in the software repository within the software package. All the documentation is written in reStructuredText. Before committing it should be checked that the documentation builds without errors or warnings, by running `make html`. After building the documentation it should be open with a browser and check that it looks OK.

CHAPTER 8

Templates

8.1 Software Architecture Document for the <XXX application>

Author Name Surname

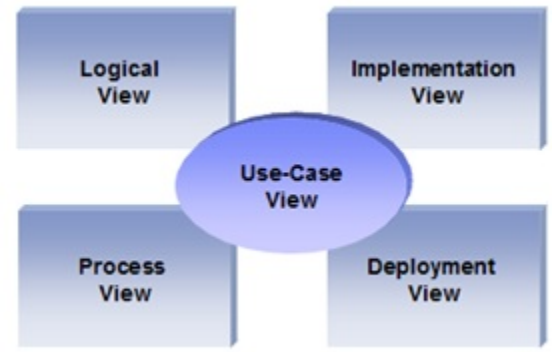
8.1.1 Change Record

8.1.2 Introduction

Purpose

The Software Architecture Document (SAD) contains the description of the system in terms of its various architectural views, in order to highlight the different aspects of it.

The description makes use of the well-known 4+1 view model.



Scope

Describes the scope of this requirements specification.

Applicable Documents

Reference Documents

Glossary

Overview

Make an overview in which you describe the rest of this document the and which chapter is primarily of interest for which reader.

8.1.3 Architectural Requirements

This section describes the requirements which are important for developing the software architecture.

Non-functional requirements

Describe the architecturally relevant non-functional requirements, i.e. those which are important for developing the software architecture. Think of security, third-party products, system dependencies, distribution and reuse. Also environmental factors such as context, design, implementation strategy, team composition, development tools, use of legacy code may be addressed. Usually, the non-functional requirements are already in place and can be referenced here. Provide a reference per requirement, and where the requirement is addressed.

Use Case View (functional requirements)

Refer to Use Cases or Use Case scenarios which are relevant with respect to the software architecture. The Use Cases referred to should contain central functionality, many architectural elements or specific delicate parts of the architecture. A Use Case template is available in Appendix A. If UML Use-Case notation is used in capturing the requirements, these models can be inserted and described in this section

8.1.4 Interface Requirements

This section describes how the software interfaces with other software products or users for input or output. Examples of such interfaces include library routines, token streams, shared memory, data streams, and so forth.

User Interfaces

Describes how this product interfaces with the user.

GUI (Graphical User Interface)

Describes the graphical user interface if present. This section should include a set of screen dumps or mockups to illustrate user interface features. If the system is menu-driven, a description of all menus and their components should be provided.

CLI (Command Line Interface)

Describes the command-line interface if present. For each command, a description of all arguments and example values and invocations should be provided.

API (Application Programming Interface)

Describes the application programming interface, if present. Foreach public interface function, the name, arguments, return values, examples of invocation, and interactions with other functions should be provided. If this package is a library, the functions that the library provides should be described here together with the parameters.

Hardware Interfaces

A high level description (from a software point of view) of the hardware interface if one exists. This section can refer to an ICD (Interface Control Document) that will contain the detail description of this interface.

Software Interfaces

A high level description (from a software point of view) of the software interface if one exists. This section can refer to an ICD (Interface Control Document) that will contain the detail description of this interface.

Communication Interfaces

Describe any communication interfaces that will be required.

8.1.5 Performance Requirements

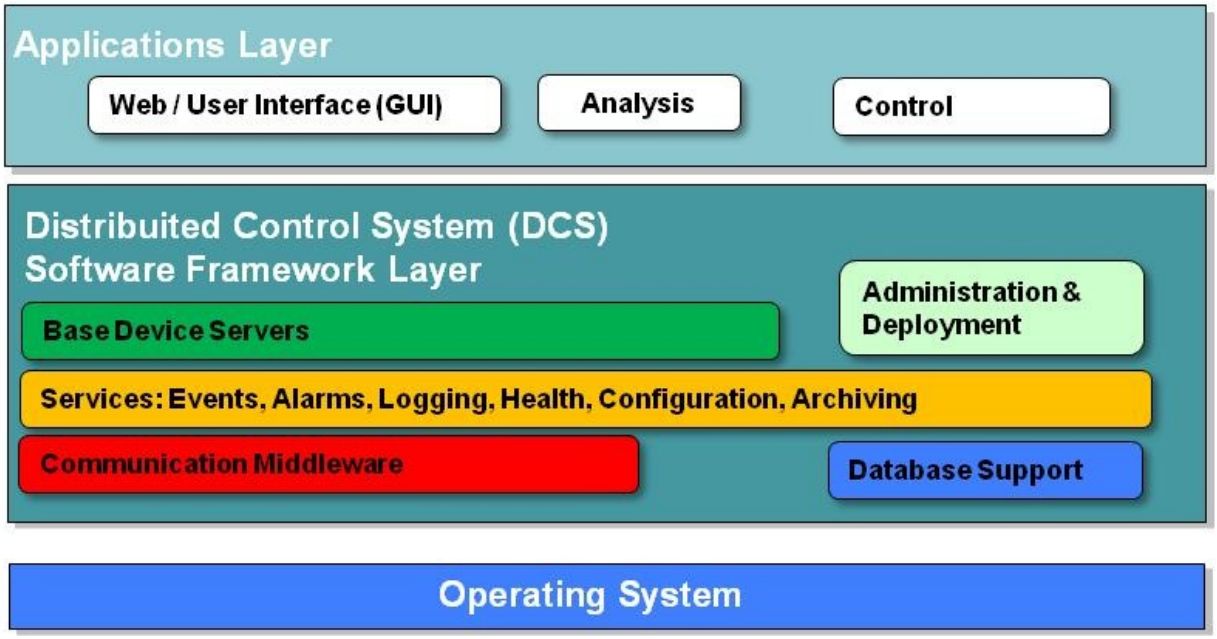
Specifies speed and memory requirements.

8.1.6 Logical View

Describe the architecturally significant logical structure of the system. Think of decomposition in terms of layers and subsystems. Also describe the way in which, in view of the decomposition, Use Cases are technically translated into Use Case Realizations

Layers

The ERAS software applicationg belong to the heterogeneous Distributed Control System (DCS) domain which can be represented as a layered architecture. This is a very common design pattern used when developing systems that consist of many components across multiple levels of abstraction as in ERAS case. Normally, you should be developing components that belong to the Application layer



Subsystems

Describe the decomposition of the system in subsystems and show their relation.

Use Case Realizations

Give examples of the way in which the Use Case Specifications are technically translated into Use Case Realizations, for example, by providing a sequence-diagram.

8.1.7 Implementation View

This section describes the technical implementation of the logical view.

8.1.8 Deployment View

Describe the physical network and hardware configurations on which the software will be deployed. This includes at least the various physical nodes (computers, CPUs), the interaction between (sub)systems and the connections between these nodes (bus, LAN, point-to-point, messaging, etc.). Use a deployment diagram.

8.1.9 Development and Test Factors

Hardware Limitations

Describe any hardware limitations if any exist.

Software validation and verification

Give a detail requirements plan for the how the software will be tested and verified.

Planning

Describe the planning of the whole process mentioning major milestones and deliverables at these milestones.

8.1.10 Notes

8.1.11 Appendix A: Use Case template

Use Cases drive the whole software process and bind together all the phases from requirements capture to final delivery of the system and maintenance. They are a very effective way of communicating with customers and among team members. Before every discussion always provide the partners with a set of relevant Use Cases.

During meetings, they stimulate focused discussions and help identifying important details. It is important to keep in mind that Use Cases have to describe WHAT the system has to do in response to certain external stimuli and NOT HOW it will do it. The HOW is part of the architecture and of the design.

What follows is the empty template:

8.1.12 Use Case: <Name>

<Short description>

Actors

<List of Actors>

Priority

<Low, Normal, Critical>

Preconditions

<List of preconditions that must be fulfilled>

Basic Course

<Step-by-step description of the basic course>

Alternate Course

<Step-by-step description of the alternate course>

Exception Course

<Step-by-step description of the exception course>

Postconditions

<List of postconditions (if apply)>

Notes

8.2 <Application> Software User and Maintenance Manual

Author Name Surname

8.2.1 Change Record

8.2.2 Introduction

Purpose

Describes the purpose of the document, and the intended audience.

Scope

Describes the scope of this requirements specification.

Applicable Documents

Reference Documents

Glossary

8.2.3 Overview

Hardware Architecture

This section should give an overview of the corresponding HW environment that is related to the application, referring to the involved devices, their deployment, etc.

Devices and Components

Involved HW devices (if applies) should be mentioned here, technical details should be mentioned at an appendix.

Hardware Layout Diagram

If applies, a general HW layout diagram could be shown here. This diagram should focus on the devices mentioned before, plus WSs, Ethernet backbones, gateways, subsystems, etc.

Software Architecture

In this section a general overview of the software architecture should be given.

Interaction Diagram

A general system interaction diagram should go here.

Deployment Diagram

This diagram shows the deployment of the application over the mentioned HW layout.

8.2.4 User Manual

This section contains general descriptions, screenshots, etc., that explain how to use the application. Some examples are:

Application Start-up

Application Shut-down

Alarms

GUIs

8.2.5 Maintenance Manual

This section should contain all the information necessary to maintain the application. For example, it should describe the SW structure, how to perform configuration changes, software package names, basic package requirements, dependencies, etc.

Application Documentation

If applies, you could mention here any other relevant application documentation like README files, etc.

Class Diagram

If not automatically generated, this section could contain a class diagram.

Application Program Interface (API)

If not automatically generated, this section could contain the application's API.

Package Diagram

Gives the name(s) of the involved packages and shows the package diagram.

Error Definitions

If applies, you could mention here the error definitions that apply for the application.

8.2.6 FAQ and Troubleshooting

Here you could write down FAQs and any relevant hint for the application's troubleshooting.

8.2.7 Installation Guide

HW & SW Precondition

Hardware:

- TBD

Operating system:

- TBD

Configuration:

- TBD

Software packages:

- TBD

Step-by-step Procedure

Hardware Preparation

Application Installation

8.2.8 Appendix A

CHAPTER 9

Glossary

ERAS European Mars Analog Station

IMS Italian Mars Society

CHAPTER 10

Indices and tables

- genindex
- search