ERICA

Empathic Robotic Interactive Companion and Assistant

QC Enterprises, FT_7

Floris den Heijer, Jordy Heemskerk & Olav Trauschke

Contents

Business Case

At the heart of the QC product line lies ERICA, a general purpose robot for the elderly, capable of making coffee and monitoring client's health. A social and lovable robot, who is customizable in appearance and extendible in capacity. As the first entry in the QC Robotic Universe, she is a jack of all trades, but master of none and will thrive if supplemented by more specialized QC robots. By entertaining clients, who are often lacking (or missing frequent) visits of family or friends, ERICA improves quality of life.

Scope

During the conceptual phase of ERICA's design we touched on the topic of a supporting infrastructure. Any robot is a highly complex device and will likely require frequent updates preferably pushed over the internet. In addition, the modular functionality approach of ERICA demands a marketplace, possibly integrating third-party solutions. While this platform is interesting, for this project we will assume this platform to a) exist and b) be sufficient for our international ambitions.

Stakeholders & Requirements

Below, we describe the requirements that must be fulfilled for given use cases for ERICA of a given stakeholders of this system. A more complete overview of the requirements per use case per stakeholder can be found in Appendix D.

Internal QC stakeholders

For adapting the company's update system to be able to update ERICA, the software architecture team needs the system to be connected to the Internet and to be capable of executing updates. The updates that should be executed will be pushed to the system by the software maintenance team. In addition, 'updates' adding complete new modules to the system will be pushed to individual robots by the customer support department. The system's Internet connection will also be used by stream video and audio and send recorded vitals, medication intake and physical activity to a QC Control Center (QCCC).

External stakeholders

External software developers need to be able to deliver modules for ERICA to QC Enterprises to have them distributed to users of the system. To be able to develop these modules, they need access to a clear description of an interface to communicate with the rest of the system.

The elderly using ERICA need to have a nice and useful system that behaves as expected, safely and within reasonable time after they have instructed it to do something with as little effort as possible. Therefore, they need to be able to communicate easily and the system must be able to execute its functions safely and within reasonable time. For many of its functions, the system also needs to be able to move around the house of its users and that should be done without causing harm to users, people visiting their houses and their houses or their belongings. The system should at least be able to maintain a calendar, perform household tasks, make (video) calls, support conversations, encourage interaction by interacting with pets and other care-robots, play games, track medication intake and physical activity, gather vitals and report emergencies.

Design considerations and trade-offs

The architecture discussed in this document is based on a several important trade-offs which will briefly be discussed in this section. A key consideration is scalability of the product, in large part due to the vast market size[1], the complexity of robotics and the unique challenges of bringing robotics into the personal space. Scalability has a large impact on the envisioned architecture, areas covered here are maintainability and extensibility. In addition, some of the current technical limitations and solutions will be discussed.

To save cost and improve time-to-market, ERICA will integrate many existing robotic solutions, resulting in a heterogeneous computing environment. Experience and robotic specialists tell us that maintaining compatibility and stability in such an environment is a major concern, which should be reflected in the architecture. In broad terms, we expect two kinds of errors: hardware related (i.e. sensor failure, battery) and software. Hardware errors will be primarily be solved through factory support and will not be covered in this document.

Key to tackling failures is identification of failures in the first place, as with any complex system bugs will often be found ‘in-the-wild’. A system should be in place for users to report obvious and major malfunctions, likely a telephone based and on-site support system is best suited for this job. Alternatives could include a direct ‘call support’ button on the robot connected to a communication system external to ERICA. But if this system should fail, another system is needed regardless and benefits of a direct button versus a telephone system are meager.

In addition to the user operated support system, a support structure permitting automated software updates and error reporting should be in place. Users are not expected to be competent in modern technology and as such will not be able to a) diagnose problems and b) solve any but the most basic failures. Delivering software updates can be achieved through either the internet or a DVD/media delivery scheme. The latter comes with serious disadvantages such as cost (production, delivery) and delay, though it might be useful in remote locations. Advantages of the internet are easy scalability, the ability to gather additional deployment or diagnostics and as little external effort.

Delivering updates is not an easy job, as ERICA spans multiple connected systems at two sites (more on that later). Updates may include firmware for a camera, accessible only through a vendor API or a (software) functionality upgrade. This calls for a service stack which is able to access – at some level – all software in the system and requires additional effort.

Stakeholders require some level of software extensibility through different product lines or purchasable add-ons, the most logical way to represent these are with modules. Managing a calendar and performing household tasks can be implemented in a "personal assistant" (PA) module. All communication tasks can be implemented in their own "rich communication" (rich comm.) module. Finally, tracking and sending information about a person's health can be implemented in another ("health") module. All these modules need to be supported by modules abstracting hardware for vision, hearing, speech, movement, manipulation, peer-to-peer radiofrequency communication, Wi-Fi connection, and display and monitoring

[1] Over 2 million elderly in the Netherlands alone (http://opendata.cbs.nl/)

devices. The locomotion subsystem should make sure the system is mobile enough and can for example ascend and descend stairs.

Certain aspects of robotics require intense computing, primarily vision and recognition algorithms. There is a trade-off between ERICA's mobility and its onboard processing resources, as more processing power means a greater electrical power draw and an increase in size. Putting ERICA on a power cable is not a feasible option from a marketing point of view and the expected increase in size and noise would undercut the 'cuteness' requirement.

An example of this tradeoff is based on the HERB[2] experiment: realistically, video data can be processed by a mobile robot at a rate of 1 or 2 frames per second. Such a low frame rate makes it impossible to detect human expression and eye movement, greatly limiting the social capabilities of ERICA. By delegating computationally heavy tasks to a remote system, around 60 frames per second can be achieved. For this reason, ERICA will span two systems connected through a low-latency, high-bandwidth wireless connection.

This introduces new problems, as a remote connection is by definition orders of magnitude slower than an onboard system. In addition, wireless connections have a higher loss rate as ERICA moves around the room. Algorithms using vision processing must be aware of these constraints and must be able to fallback safely or have enough confidence to perform a task even when 'blind'.

To enable the system to express personality and prioritize different tasks, we added a layer with a complex reasoning system ("Governing AI"), that also directs all communication between other (functional and hardware bound) modules, so that it can prioritize and schedule whatever has to be done. To make the system even more attractive, learning by motivation (by the user) is included in the personality system, as was done in PARO[3]. For implementing health monitoring and communication with healthcare providers, implementation can be based on a third system: GiraffPlus[4].

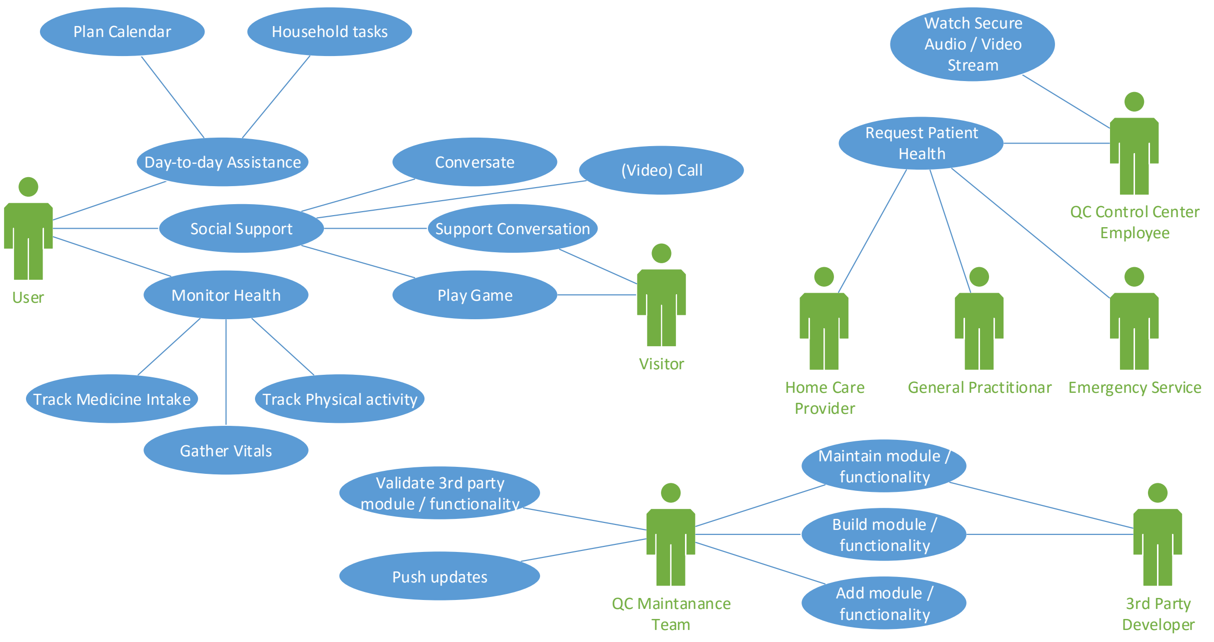
To keep the system affordable and marketable, we decided not to include medical functionalities in the first version of the system, instead allowing them to be inserted as new modules, possibly developed by third parties later on. This decision was made because adding medical functionalities would have introduced new stakeholders such as medical regulatory agencies and likely new requirements. Therefore, it is likely a bigger investment would have been needed and the system would have had to be more expensive, thus less affordable.

[2] https://www.ri.cmu.edu/pub_files/2010/1/HERB09.pdf
[3] http://www.parorobots.com
[4] http://www.giraffplus.eu

Architecture overview

Below, the architecture we derived from the above requirements and decision is described in views based on those used in the 4+1 architectural view model.

Use case view



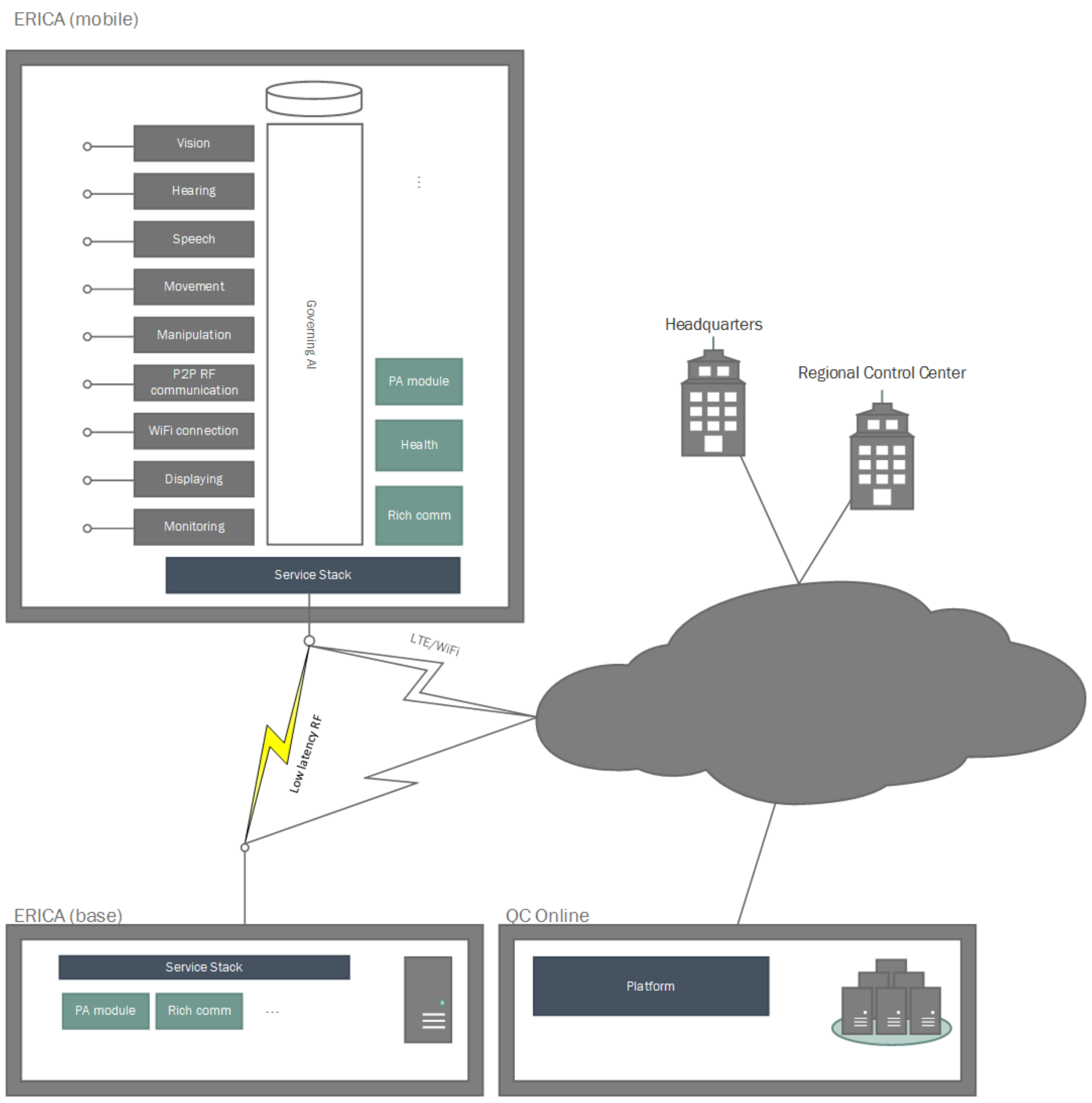
The above diagram documents the actors in the software for the ERICA system and their desired functionality. From the user's perspective ERICA should be a highly functional robot, capable of socializing and easing day to day tasks, as well as functioning as a health monitor. Communication of these medical insights is the primary concern for care providers and GP's, though emergency services should be able to access the data if possible. Development is primarily concerned with modular development and the ability to push updates. Third party modules can be added to ERICA, though these must always to through extensive validation by QC staff.

Logical and deployment view

The ERICA system is the combination of three physically separate systems, namely:

Mobile
Base station
QC Online

The robot is the main actor and holds most hardware capable of interacting with the environment. Due to the energy and space constraints imposed by the robot’s mobility, more powerful computing is available from a local base station. Finally, synchronization, data services and insights are communicated through the QC Online platform.



Presented above is the general logical view for ERICA. Components in the architecture are:

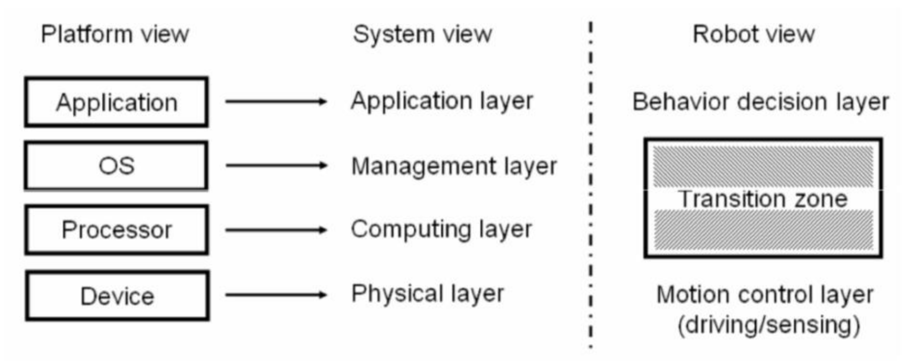
1. **Off-the-shelf hardware subsystems**, responsible for vision, output and environmental actions;
2. **Governing AI**, with persistent personality and capacity to reason about available resources;
3. **Functionality modules**, providing integration services and high-level functionality;
4. **Service stack**, providing a consistent mechanism for updates, communication and diagnostics.

As described in more detail below, these components will all work on top of the platform consisting of the hardware and the Robot Operating System (ROS).

Hardware requirements

ERICA is designed to be used with common, off-the-shelf hardware (COTS). The mobile system will at its core use a multi-processor, multi-core x86 system connected via Ethernet or other high-speed interconnect to other (vendor specific) nodes. The base system must be horizontally scalable through the connection of heterogeneous compute nodes, for instance through a blade system with a mixture of CPU, GPU and IO nodes. The main components of the hardware are described by the images below.

Process view



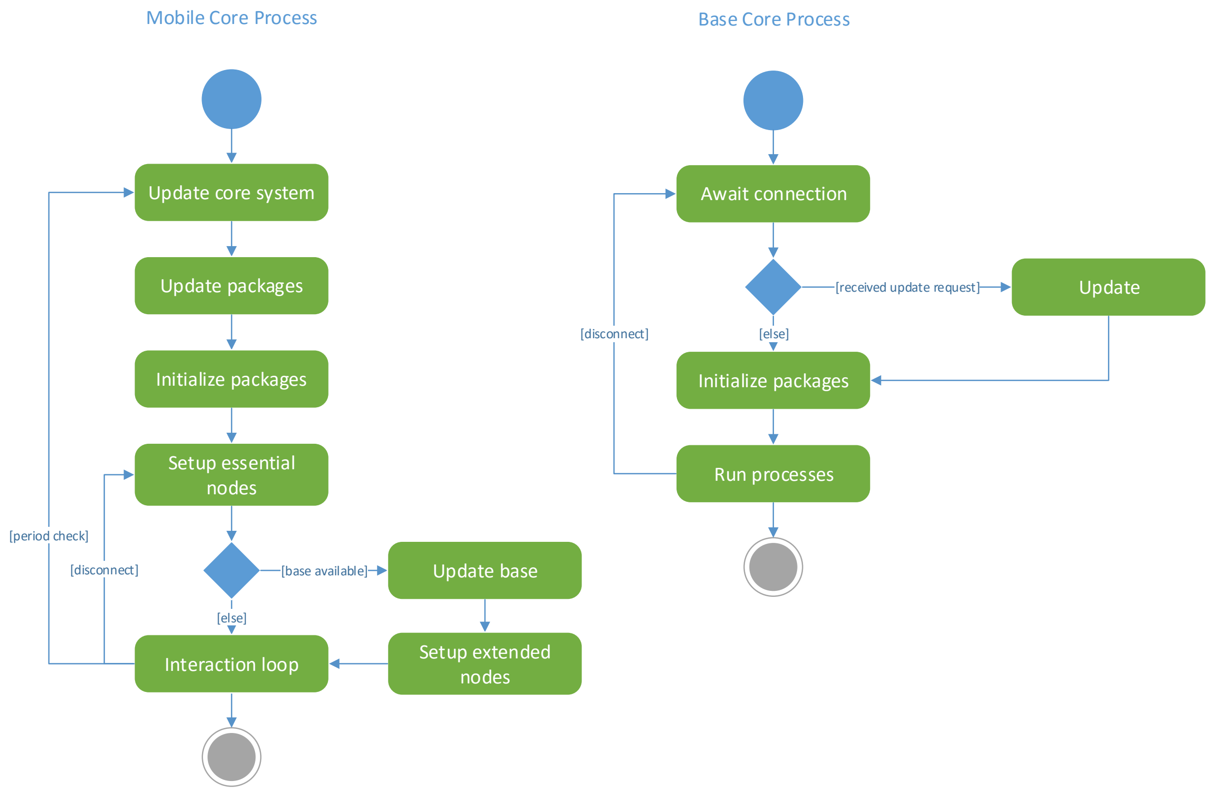
Central to the implementation and process views is a set of middleware called Robot Operating System, or ROS[5]. It offers an integrated approach to robotic development and focuses on peer-to-peer (i.e. distributed) computing, multi-language and reusable components. Due to the *heterogeneous* nature of robotics is key to both ROS and ERICA's architecture. Rarely is a system as easily consolidated as a desktop application. ROS offers a way to spread computing across heterogeneous systems through *nodes*. It also provides standard operating services such as hardware abstraction, low-level device control and package management.

Software in ROS is organized in packages. A package might contain nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages it to provide this useful functionality in an easy-to-consume manner so that software can be easily reused. This reusability means ERICA is able to use off-the-shelf systems for its peripheral features (vision, locomotion) and also provides a way to integrate and deploy custom components.

[5] http://www.generationrobots.com/en/content/55-ros-robot-operating-system

ROS requires a *coordination node*, responsible for the creation and deployment of other logical nodes including the capability to *offshore* computing to a remote system. As the wireless nature of the robot means a connection to the base station cannot be guaranteed, and it cannot be expected of an autonomous robot to be fully dependent on such a system, the mobile system is best suited for this role.

By embracing ROS's organizational structure processes are easily structured as crucial aspects such as inter-process communication and distributed data storage can be tackled with relative ease[6]. In terms of processes, ERICA is modular to a large extend. At the time of writing, no details are available on specific subsystem requirements, though it is known that both mobile and base systems run at least one process responsible for coordinating each system. These processes may spawn or startup new processes depending on hardware and role. On the next page the lifecycle of both processes is shown.



The lifecycles described above are mainly concerned with delivering updates and initializing ROS nodes. These processes are different on mobile and base, as the base system serves as an *extension* of the mobile unit. Modules running on the base are likely highly reliant on those running on the mobile unit, therefore the mobile unit is responsible for determining the software present on the base system.

Under 'core' system, everything down from the management layer shown on the previous page is understood. This includes Unix OS updates and firmware upgrades crucial to the functioning of the core process. Subsystems are updated through packages, as they likely require highly specific or vendor dependent code. All package management is performed by QC Online, responsible for maintaining an

[6] QC online will run from a PaaS solution such as Azure or AWS ensuring scalability, though further details will be omitted.

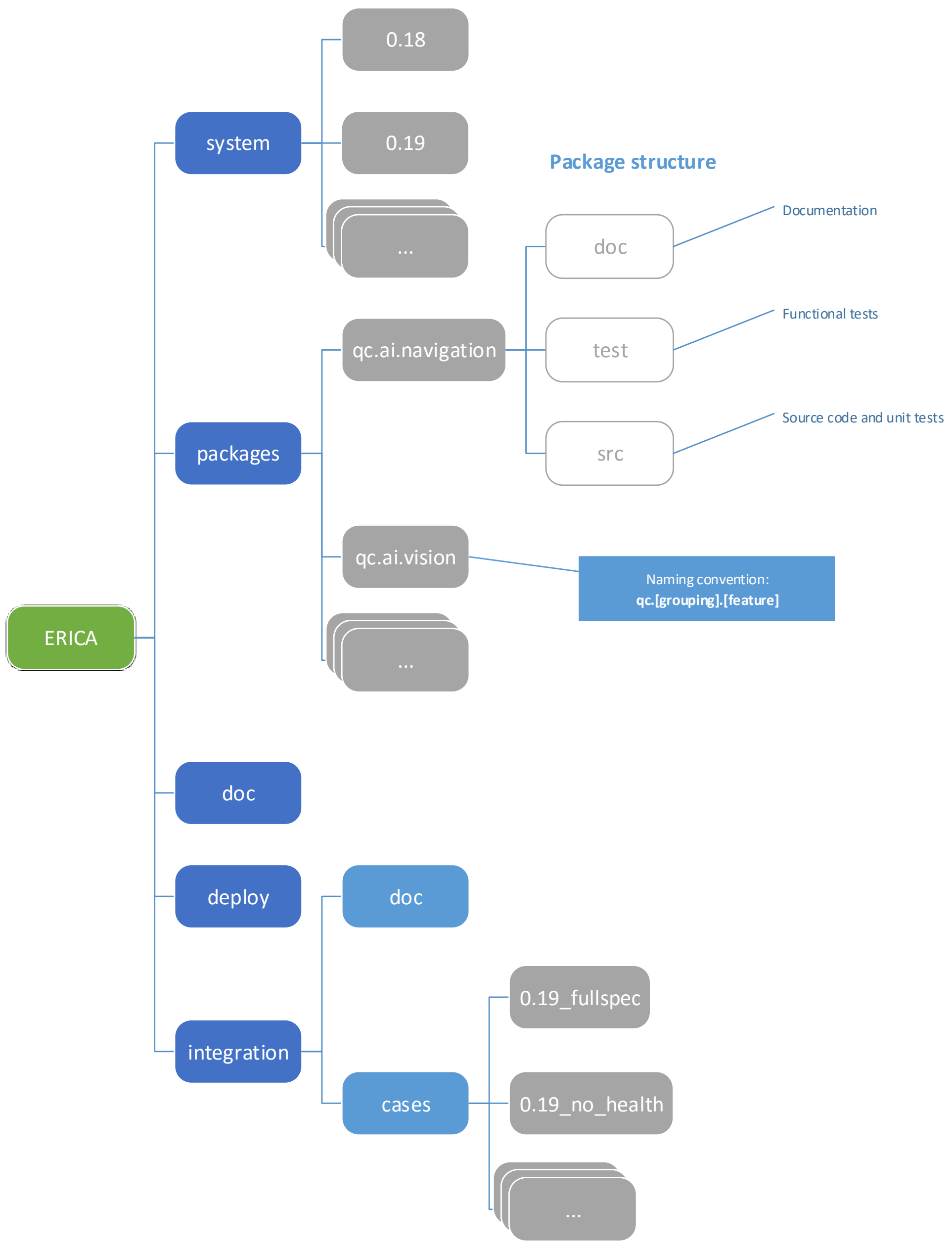
archive of versioned packages and system updates. Should the connection to the base system be lost resulting in another node setup, only packages dependent on the base node should be affected. This means the core AI process will not be killed in such an event.

Nodes are connected through network spanning multiple physical systems. The physical systems are connected through a *to-be-decided* high speed wireless interconnect which delegates connectivity to hardware drivers and the ROS. Internal systems are connected through either a high-speed Ethernet network or a custom interconnect, depending on latency requirements and hardware availability.

Implementation view

Most of ERICA's functionality will be delivered through ROS packages, the remainder being system updates or QC online related development. A ROS package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. They can also be grouped into *meta-packages*, or groups of packages. These packages can define *dependencies*, allowing modular development and testing. Occasionally, system-wide updates may be necessary which may involve kernel updates, key firmware and driver updates and common libraries. These are pushed through the UNIX distribution system.

Most development will focus around the creation, deployment and testing of ROS (meta-) packages which requires a strict source control and naming regime. An outline of the ERICA's source repository is given below.



To work with ROS and to keep different packages developed by different parties consistent, all packages must comply with the regulations described in Appendix E. For each deliverable product, that is, ERICA with a set of features, an environment for integration testing should be provided. When a package is tagged, it will automatically be pushed to the build server, which will run a full or partial test suite. Before pushing to quality assurance, a package must be tagged and built without warning. Examples of packages are the updater, hardware interfaces, components of the governing AI such as a task scheduler and functionality modules.

Component Quality Attributes

Vision module

Availability

The movement and interaction of the robot depend on its vision. Therefore, the availability of this module is crucial for the availability of the whole robot. The robot will be situated in the homes of, mostly non-technical, elderly people, for this reason the robot should not require any technical knowhow to use. This is also the case for crashes, these should not require input, and not happen more than a day per year, which gives an uptime of 99.9%. Because this is the uptime for the entire robot, and it depends on the vision module, this module should also have an uptime of at least 99.9%.

Camera failure

One thread to the availability of the vision module is the possibility of camera failure. Since the cameras will likely be bought as commercial of the shelf components, implementing tactics that require action by the cameras themselves are no option for detecting this kind of faults. Since using multiple cameras at the same time would badly decrease battery life and thus the availability of the system, we also decided against voting, because the availability of the system as a whole is the sole reason for increasing the availability of the vision module.

This leaves us with the following tactics for detecting faults: time stamp, sanity checking and throwing an exception in case of a timeout. These three tactics all detect different kinds of faults, so we decided to use these alongside each other. The **time stamp** tactic will be used to have the camera driver verify that the frames the camera sends to it are up to date. **Sanity checking** will be applied to check if a frame is indeed of the form a frame should be of and throwing an exception when no new information is received for more than the time in which two frames should have been received, a **timeout** exception will be thrown. Off course, timestamping and checking timestamps and sanity and checking for timeout, will all influence performance of the system negatively. This needs to be taken into account when deciding what hardware is needed for the base station to enable the robot to perform good enough.

Since cameras are crucial for safe operation of the robot and only detecting faults does not solve anything, tactics need to be implemented to recover from faults too. As stated before, active redundancy would influence the battery life of the robot too much, and thus we decided to implement **passive redundancy**, with all cameras overlapping half of the area each of its neighboring cameras covers, and only one of each two cameras active in a normal situation. This way, when a camera fails, its two neighbors can be activated to preserve the functionality of the vision module. After a camera has failed, it should be restarted and then be set to **shadow** its neighbors, which have taken over its function. This shadowing can be used to judge whether the camera is functioning normally again. When it does, it can then take over its function again, so that its two neighbors can be switched off again to safe power and thus increase the availability of the system as a whole.

Since commercial of the shelf cameras might be used, exception prevention and increasing the competence set, cannot be used to prevent camera failure. Removal of a camera from service is also not an option, since its function would then have to be taken over by the two neighboring cameras, which would decrease battery life. Therefore, to prevent this kind of faults, a **predictive model** will be used. When the camera driver notices the quality of the images generated by a camera degrade, the nearby QCCC needs to be notified maintenance is necessary.

Driver Failure

Another thread to the availability of the vision module is the possibility of camera driver failure. It is likely hardware drivers will, like the hardware itself, be commercial of the shelf components and it will thus not be possible to implement tactics for which the driver itself would need to be changed. Since timestamping and sanity checking have already been applied in the driver and it seems unlikely the driver affects the sanity or the order of camera frames, we also decided against implementing timestamping and sanity checking again. This leaves us with throwing an exception in case of **timeout** to detect faults. Like when a driver would detect failure of a camera, this can be done when no frame has been received in the time two frames should have been received.

In the case of a driver, redundancy seems hard to realize and we therefore decided against implementing this tactic for recovering from failure of a driver. Tactics like retrying, rollback, ignoring faulty behavior and degradation seem not to be applicable in this case either. This leaves reintroduction methods to recover from this kind of failure. Specifically, we decided to implement **escalating restart** to first remove possible problems with the driver or the camera it drives by restarting them, restarting the whole vision module if this has not solved the problem and finally restarting the complete robot.

When a problem has been detected the vision module should also tell the governing AI to switch to a **safe-mode** until the escalating restart was successful, or maintenance has solved the problem if an escalating restart did not. This means all tasks should be finished gracefully and no new tasks should be started. Implementing this procedure will have a negative influence on the performance of both the vision module as the governing AI, which should have a procedure for gracefully aborting a task ready whenever it starts a task.

To prevent faults in the commercial of the shelf component a camera driver will likely be, faults will have to be reported to the developer of this component to **increase the competence set** of the driver. This is especially the case when a driver failed due to a specific input from a camera.

Vision interpretation software Failure

A final thread to the availability of the vision module is failure of the vision interpretation software. This kind of failure should be detected by the governing AI. We decided to implement the **ping/echo** tactic for this rather than the heartbeat tactic, so that the governing AI is responsible for this process. This is important to keep the responsibility for this failure checking within software developed by QC Enterprises in case a commercial of the shelf component is bought for the vision interpretation. In addition, **timestamping** will be applied so that the governing AI validates the information it receives from the vision module is up to date.

To recover from failure of the vision interpretation software redundancy can be applied. This should again be **passive redundancy** to avoid too much impact on the performance of the system and its availability, due to decrement of the battery lifetime. While switching to the spare software, the earlier described **safe-mode** should be activated to make the system operate as safely as possible without vision. Safe-mode may not be left again until the vision interpretation software is working properly again. To attempt to get this software to work again, the redundant component can be used.

Despite that the redundant component likely allows the system to start operating normally again, the failed component should be **restarted** to attempt to get it to work again. When that component is successfully, it should **shadow** the redundant component that has replaced its position, to test whether it is functioning properly again. When this component is indeed functioning normally again, it can be set as the redundant component to get to work again if the other, now active component fails.

Whether or not the failed vision interpretation software was successfully recovered after failure, its failing should be reported to the QCCC so that the failure can be investigated and an update can increase the competence set of the system so that it will not fail for the same reason again.

Performance

The system has to be able to process the frames received from the camera (driver) within 400ms. This measure is derived from the fact that Erica moves at most 3km/h, while a person walks no more than 6km/h. This makes a total of 9km/h and for the robot to see each meter difference; it should refresh every 400ms.

To reach the required performance level, the **sampling rate can be managed** by making the robot stop moving in case it is not able to perform fast enough while moving. In addition, if there is a frame waiting to be processed while another frame is delivered, the old frame should be replaced by the new one so that no outdated information is processed, which is a way of **prioritizing events**.

Furthermore, to reach the required performance level without decreasing the battery life and thus the availability of the system too much, computationally heavy tasks should be run on the base station as well as on the robot itself so that these tasks can be “offshored” to the base station when the robot cannot perform them itself within the performance limits set for it. During development the **base station’s resources should be increased** until the system is able to perform well enough.

Scalability

When after the release of new functionality modules, robots start to perform worse than the lower bounds given above or have to stop moving to buy themselves extra time to process information, the base stations **resources should be increased**.

Vitals monitoring

Availability

The availability of the process of monitoring vital bodily functions may be low, because these functions will be used for monitoring only and thus will not influence a user’s health. Also, if this process in unavailable for some time, this will not influence the availability of other processes the system is involved in. If monitoring of vitals is critical, ERICA should not be used for gathering this information or at the very least another option to gather the critically required information should be available timely if ERICA fails to gather this information. We assume however, this does not influence the usability of ERICA, because people will likely be admitted to a hospital when gathering information about their vital bodily functions is critical, thus ERICA will not be used in such cases, since the system is meant to be used in the houses of its users only.

Performance

The performance of the measuring vitals has to approximate that of a doctor measuring it. Performance of processing that gathered data is much less relevant, as interaction with the user is no longer necessary and this task can thus be performed in the background. The gathering of vitals may be done at the user’s discretion when asked for, it does not matter if the processing of those vitals takes a couple of minutes longer. As long as the vitals are sent within 10 minutes of gathering them, the medical personal still has time to react to the new vitals.

Usability

The usability of the vitals monitoring has to be high. Taking vitals is a difficult and precise job and can be hard for a user who is not medically trained. Therefore, the robot should guide the user when asking them for their vitals, as if help from a healthcare provider would still be needed, it would defeat the purpose of this function. The guiding can be done by the screen on the robot with a visual representation of the procedure. By using visuals the user can more easily understand and do what is asked from him.

Safety

While monitoring vital bodily functions will not be critical for a user's health when ERICA is used to perform this task, attempting to gather this information in a wrong way could harm users. Therefore, it is important safety is carefully monitored when ERICA is used for this purpose.

Firstly, as with any diagnostic or therapeutic procedure performed by whatever entity, ERICA needs to get informed consent of the person the procedure will be applied to (the user in this case), before attempting to gather vitals (a diagnostic procedure). Therefore, the system should explain what it is planning to do and ask the user whether he or she has understood and agrees with this plan. The system may only proceed with gathering the requested vitals if the answer to this question is positive. If the answer is negative, the system is unable to interpret the answer or no answer is given at all, the procedure may not be started and instead a warning should be sent to the nearby QCCC.

The driver of any device used to gather information about vital bodily functions should report everything it does to the health module's vitals gathering system and may not start a next step until this system has reported the last step was correct. The vitals gathering system should check whether the information reported by the driver **complies with the specification of the procedure** it has started. Only if this is the case, this system should send instructions for the next step to the driver. If this is not the case, an **abortion procedure**, that must be specified for each state in each vitals gathering procedure the system can perform, must be sent to the driver instead of the next step. If this has happened, a **message reporting the problem must be sent to the QCCC** and the system may **not attempt to gather vitals again** until a maintainer has informed it that it is functioning properly again. In addition, if the driver reports any action that does not comply with the abortion procedure, while instructions to perform such a procedure were sent to it, an **emergency message** describing the situation should be sent to the QCCC so that emergency assistance can be sent if necessary.

The safety tactics described above might avoid vitals from be gathered according to schedule, negatively influencing the systems effectivity and ease-of-use. This is, however, to ensure safety when attempting to gathering vitals. Abortion procedures should be executed immediately and without being interrupted. This might influence the performance of the system in other aspects negatively, but when an abortion procedure is activated, executing this procedure correctly and as fast as possible might be necessary to avoid harming the user. For testing the safety of the system, people with medical knowledge should be asked for help.

Appendix A: References architectures

There have been multiple medical and helping robots in the past, this chapter will describe some of their architectural decisions which are of interests of us.

HERB

Figure 1 HERB

HERB is a Home Exploring Robotic Butler built by Intel. The robot, built on top of a Segway base, has a robotic arm and several sensors. The system's architecture was designed around two driving principles: the availability of unlimited computational power and minimal human input into sensing and planning algorithms. The architectural decision of interest for the development of ERICA is to offshore most of the computations. This keeps the power consumption of the robot as low as possible and thus increasing the battery life. This also gives the developers more computing power for things such as object recognition.[7]

PARO

PARO is an advanced therapeutic robot, which allows for animal therapy in environments which do not allow for animals. It improves the relaxation and motivation of patients, helps reduce stress and improves the interaction between patients and caregivers. The patient can also influence the behavior of PARO by stimulating or discouraging the behavior as it happens. This learning and motivating is something which might also be implemented in ERICA, to make the user feel at ease.[8]

Figure 2 PARO

GiraffPlus

GiraffPlus is a complex system which can monitor activities in the home. This is done with sensors in the home and on the body. The center of this system is a telepresence robot, which allows elders to communicate with relatives and health care professionals. This social aspect has been of interest for us from the very beginning. The telepresence robot itself is not capable of much more than just facilitating social interactions, but can be a good addition to the ERICA platform. The monitoring system on the other hand can detect several things in the home. For instance it is able to detect blood pressure and if the elder has fallen, and then contact the appropriate person or professional.[9]

Figure 3 GirfaffPlus

[7] https://www.ri.cmu.edu/pub_files/2010/1/HERB09.pdf
[8] http://www.parorobots.com
[9] http://www.giraffplus.eu

Appendix B: Overview of architectural patterns and styles

This appendix will describe some architectural patterns which are of interest for the development of ERICA.

Relevant patterns and styles

Component based

One of the main ideas we had when we were designing ERICA was that it should be completely modular / component based. This is described in this architecture. Using a component based architecture allows developers to replace components with new ones, as long as they have at least the same functionality as the previous component, and do not require more functionality from other components. But also allows development of new components which use other components without having to know how the other component is implemented.[10]

Publish-Subscribe

The publish-subscribe pattern allows publishers to pass messages to receivers, without knowing who the receivers are. The publisher just writes to a *topic* and every receiver who is interested in that topic, can subscribe to it and will then receive the messages which are sent by the publisher.[11] Using the ROS framework to facilitate this also abstracts from the hardware, eliminating the need to deal with operating systems, transport protocols and other platform dependent issues.

Layers

As can be seen from the logical view diagram, ERICA is structured in layers. For the mobile device (the ERICA robot) for example, the hardware packages are the lowest layers, functioning as services for the layer above. The next layer is the governing AI, which uses the hardware interfaces to accomplish tasks. The layer with the functionality modules can use the interface the AI module to implement concrete functionalities. This layered pattern makes it possible for the components within each layer to be developed or acquired independently from the layers above. It also contributes to the modularity of the system, making it possible for third-parties to add new functionality modules which are only concerned with the AI layer.

Object request broker

One of the designs decisions of ROS is that it should be easy to use with distributed computing. This has been achieved and means that a ROS node does not care where it is computed. This helps us with our design decision to offshore most of the computing to a more powerful computer. ROS can, at runtime, relocate computations to different machines.[12]

[10] https://en.wikipedia.org/wiki/Component-based_software_engineering
[11] https://en.wikipedia.org/wiki/Publish–subscribe_pattern
[12] http://wiki.ros.org/ROS/Tutorials/MultipleMachines

Irrelevant patterns and styles

Monolithic

The monolithic pattern describes a system in which the UI and data-access code are combined in a single application from a single platform. This is not desirable for ERICA as we want it to be modular. This would also implicate that if a third party developer wanted to develop a piece of software for ERICA, the whole application would have to be supplied to them. Besides that this also has negative implications to the maintainability of the system, as you have to maintain the whole application as a whole.

Appendix C: Implementation reflection

Implementing the two required cases did not go so well initially, as our new pipeline framework could not really filter. They probably used the same MSDN article as we did, as they focused heavily on threaded operation. From that article comes this diagram:

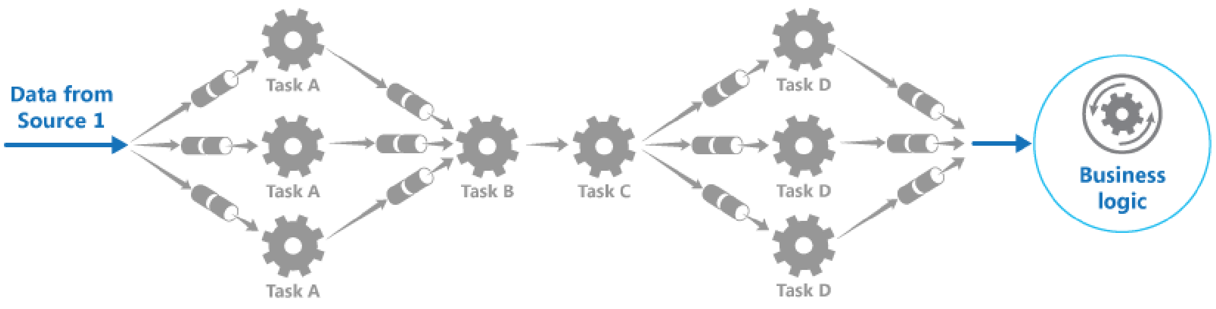


Figure 4 Pipes and Filters Pattern

If you follow this approach literally, and see data as elements rather than streams, you cannot ever imitate Linux pipes unless you allow some data to be discarded. The threading approach caused some confusion for the group, as they used a *null* return value to signal the pipeline should be closed. In essence, you can go from A to B, but not from A to nothing.

Now without going into too much technical details, you could solve this by building a 'meta'-pipeline which wraps A into a container with a flag if the element should be processed. But now you are always pushing all data through all streams, which just does not make sense. In the end some modifications were made which adhered to their interfaces, and the two basic examples were solved. Creating a more interesting network was not really possible unfortunately.

So what went wrong? Clearly the creators did not really think of a useful scenario for the pattern and instead focused on the threading aspect from the above diagram. Multithreaded Java is never easy, and unless you know some of the base libraries you will end up spending most of the time dealing with locks and concurrency. The funny thing is, when we started we took the same route. Which is why we have two implementations… It is not their fault, it just shows that unless all architects interpret the concepts the same, you will end up with a lot of extra work.

This could potentially be solved by creating some functional requirements; in code terms basically writing a couple of generic unit tests before even starting. However this might remove some of the power of architectural patterns and make them too concrete: a filter does not *have* to behave like bash. All this exercise has shown is that architectural patterns to not directly translate into code patterns. They are by definition very generic and should either be made more concrete though a use-case, or be used to explain structure or 'guide' development rather than dictate it.

Any reasonably complex application will likely incorporate many of the patterns we found. Robot OS's internal messaging system itself employs dozens of them, and that is only the internal wiring. So as an architect, you would probably use it to visualize data flow or dependencies, identify weak spots or explain to someone why something is ridiculously hard (or easy). In the pipe-filter pattern, a filter can be anything from a function to a load balanced cloud service depending on interpretation. For functions, you would skip the threading; web services, focus on retry and timeouts. If you really want to do it properly, you would focus on messaging technology and resource allocation.

Politicians view on pipes and filters: https://www.wikiwand.com/en/Series_of_tubes

Appendix D: Overview of stakeholder concerns and use cases and resulting system requirements

Internal QC stakeholders

Entity Shareholders
Concerns Making money
Use case Invest in ERICA
    Requirements Safe, marketable product

Entity Software architecture team
Concerns Being able to develop an architecture for the software and adapt the company's update system to be able to update it
Use case Develop architecture
    Requirement Clear requirements
Use case Adapt company's update system to be able to update software
    Requirement Internet connection, capable of executing updates

Entity Software development team
Concern Being able to develop the software for the system
Use case Develop software
    Requirement Clear requirements

Entity Software maintenance team
Concern Being able to maintain the software for the system and push updates to the system
Use case Maintain software
    Requirement Maintainable software
Use case Push update
    Requirement Internet connection, capable of executing updates

Entity Customer support department
Concern Being able to support users
Use case Support user
    Requirements Easy to use software
Use case Add module to ERICA for a customer
    Requirement Service stack with Internet connection, capable of adding modules

Entity Marketing department
Concern Being able to sell the system
Use case Sell
    Requirements Safe, marketable, easy to use, affordable product

Entity QC Control Center Employee
Concern Being able to check on elderly from a remote location
Use case Watch stream from QC Control Center
    Requirement System can stream audio and video to a server over the Internet
Use case Request patient health
    Requirement System can send vitals, medication intake and physical activity to the QC Control Center over the Internet when request

External stakeholders

Entity Elderly
Concerns Have a nice and useful system that behaves as expected, safely and within reasonable time after telling it what to do with as little effort as possible
General requirements Safe, easy to use, reasonably fast, mobile (moving, including ascending and descending stairs), affordable and attractive system that can prioritize different tasks and displays a personality, is available for at least a full day without having to stop functioning to charge and charges fully within a night
Use case Plan activity
    Specific requirement System can maintain a calendar
Use case Perform household task
    Specific requirement System can perform household tasks
Use case Converse
    Specific requirement System can understand spoken input other than commands to perform tasks and react to them in a way that makes an elder feel comfortable
Use case Make (video) call
    Specific requirements System can stream audio and video to a server over the Internet and sound audio and display video it receives over the Internet
Use case Support conversation
    Specific requirements System can repeat spoken input pronounced clearly or in sign language
Use case Play game
    Specific requirement System and can decide what to do in any situation that can appear in games elderly generally like to play
Use case Track medication intake
    Specific requirement System notices when an elder takes medication and records this event

Use case Track physical activity

Specific requirements System notices when an elder performs physical activity and records this event

Use case Gather vitals

Specific requirements System can announce it has to gather an elder's vitals and then gather and record them

Use case Report emergency

Specific requirements System can recognize and report emergencies

Use case Encourage interaction

Specific requirement System interacts with care-robots and pets of visitors

Entity Visitors of people using ERICA

Use case Visit

Requirement The system does not disturb visitors

Entity Family and friends of people using ERICA

Concern ERICA users should be helped by the system as good as possible

Use case Customize

Requirement Service stack with Internet connection, capable of adding modules available to customer support

Entity Healthcare providers

Concern ERICA users should be helped by the system as good as possible, but the system should also allow and help healthcare providers to help ERICA users when necessary

Use case Request patient health

Requirement System can send vitals, medication intake and physical activity to the QC Control Center over the Internet when request

Entity External software developer

Concern Being able to program modules for ERICA with reasonable effort

Requirements Being able to deliver modules to QC Enterprises to have them distributed to users of the system and have a clear interface to use while programming software for it

Appendix E: Regulations for package development

- Packages may be developed in any language suitable for their purpose (i.e. C for drivers, Python for AI interaction, Java for web integration).
- Packages must have an *identifier*, following the naming convention *qc.[grouping].[feature].*
- If the package uses a language with namespace support, this identifier must be the namespace of the program.
- Packages **must** contain three directories:
    - /doc/ for documentation
    - /test/ for functional tests
    - /src/ for source code and unit tests
- Packages are versioned *in whole integer increments* (1, 2, 3, …) and must be tagged in Git.
- The *src* directory of each package must contain an XML file (*package.xml*[13]) or must produce one when it is build.

Appendix F: Figures

Figure 1: HERB, http://www.robotshop.com/blog/en/herb-intels-butler-robot-762

Figure 2: PARO, http://www.news-medical.net/news/20130405/Animal-robots-for-dementia-treatment-an-interview-with-Professor-Wendy-Moyle-Griffith-University.aspx

Figure 3: GiraffPlus, http://www.wallstreetdaily.com/2014/08/19/giraffplus-robot/

Figure 4: Pipes and Filters Pattern, https://msdn.microsoft.com/en-us/library/dn568100.aspx

Appendix G: Review

Review goal

We would like the review of this document to focus on answering the following question: Does the reasoning sufficiently describe the relation between the use case view and the logical view?

[13] See http://wiki.ros.org/catkin/package.xml for specifications