# ERICA

Empathic Robotic Interactive Companion and Assistant – QC Enterprises
FT_7: Floris, Jordy and Olav

## 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'll assume this platform to a) exist and b) be sufficient for our international ambitions.

## Stakeholders & 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

## Reasoning and architectural decisions

For being able to update the system over the Internet, we decided to implement a service stack that manages a connection to the Internet and is capable of updating the system. To make sure the service stack can also add complete new modules to the system, we decided to keep the system modular, with modules that represent groups of functionalities. The service stack should also be able of sending data over the Internet when requested to do so by other parts of the software, as well as receiving data when it is sent to the system, to pass it on for further processing.

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

Based on the architecture of HERB[1], we implemented the possibility to offshore computationally heavy tasks to an external computer (ERICA base) in the service stack, to make sure the system can be

[1] https://www.ri.cmu.edu/pub_files/2010/1/HERB09.pdf

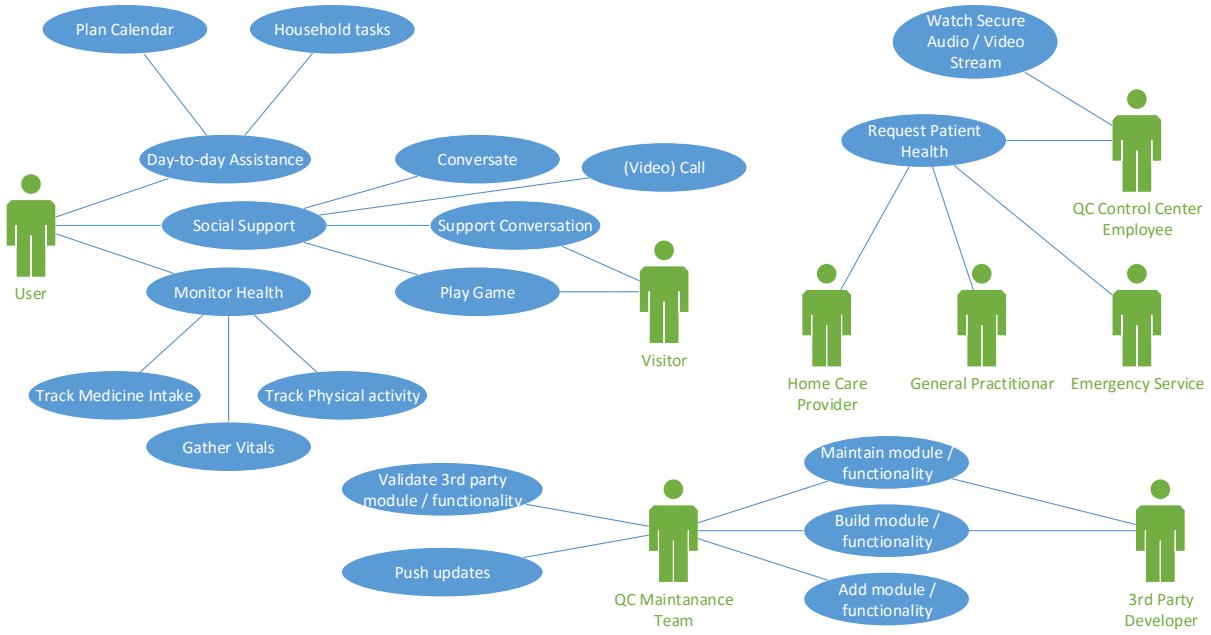
reasonably fast despite it is likely much computation much will need to be done. This also ensures the system will not need so much power it's battery life will be too short or its batteries will take too long to charge. To enable the system to express personality and prioritize different tasks, we added 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[2]. For implementing health monitoring and communication with healthcare providers, implementation can be based on a third system: GiraffPlus[3].

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.

## 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



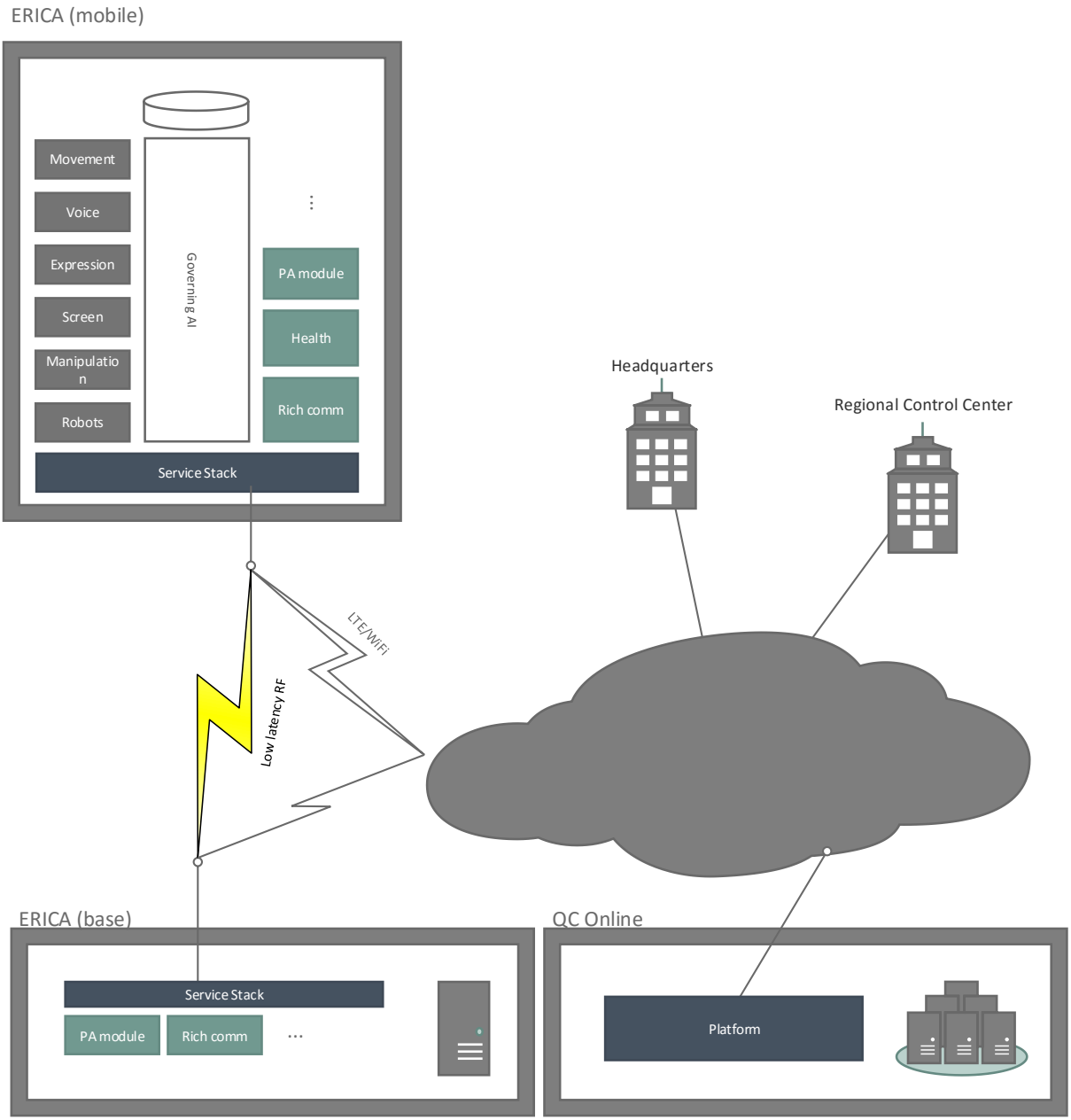

The above diagram documents the actors in the software for the ERICA system and their desired functionality. From the user's perspective ERICA should be a highly functional robot, capable of socializing and easing day to day tasks, as well as functioning as a health monitor. Communication of these medical 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.

[2] http://www.parorobots.com
[3] http://www.giraffplus.eu

## Logical and deployment view

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


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.

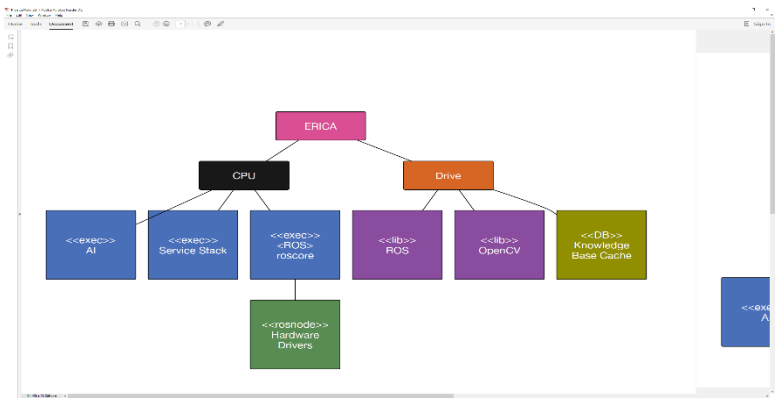


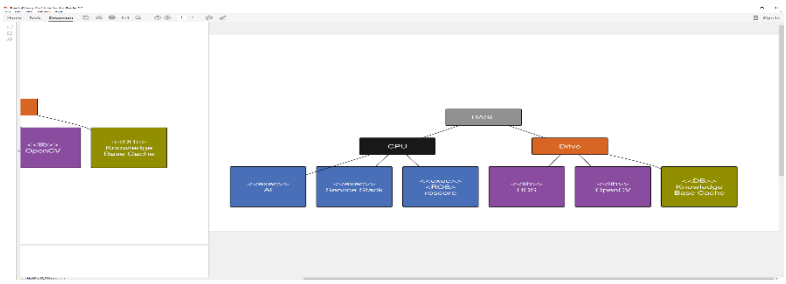
Presented above is the general logical view for ERICA. Components in the architecture are:

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;
5. **Platform**, delivering data services, backup functionality and intelligence.

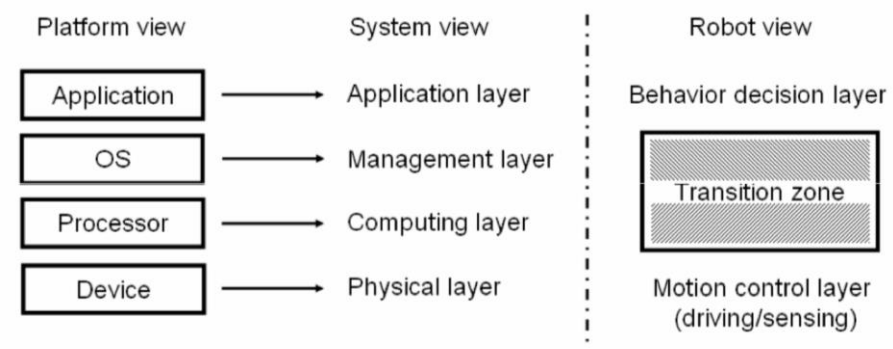
## Hardware requirements

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[4]. 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.

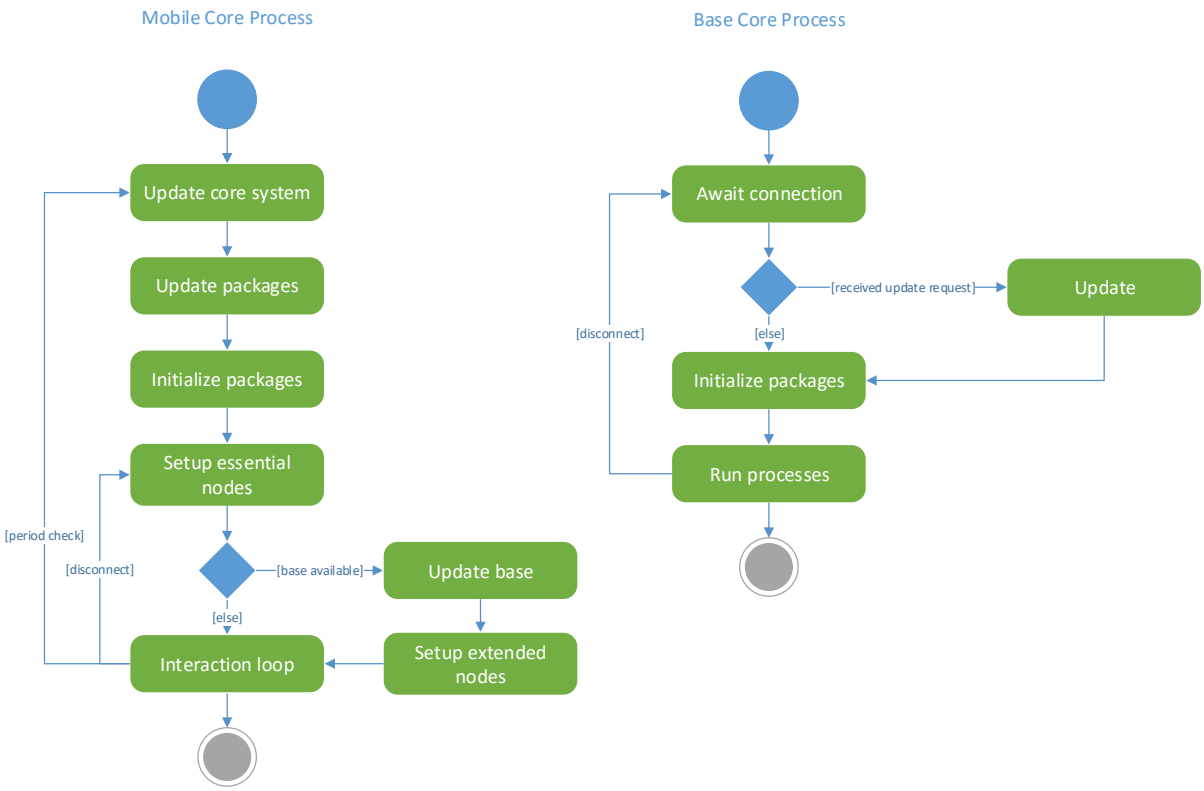
Software in ROS is organized in packages. A package might contain nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages it to provide this useful functionality in an easy-to-consume 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.

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[5]. 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.

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

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




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.

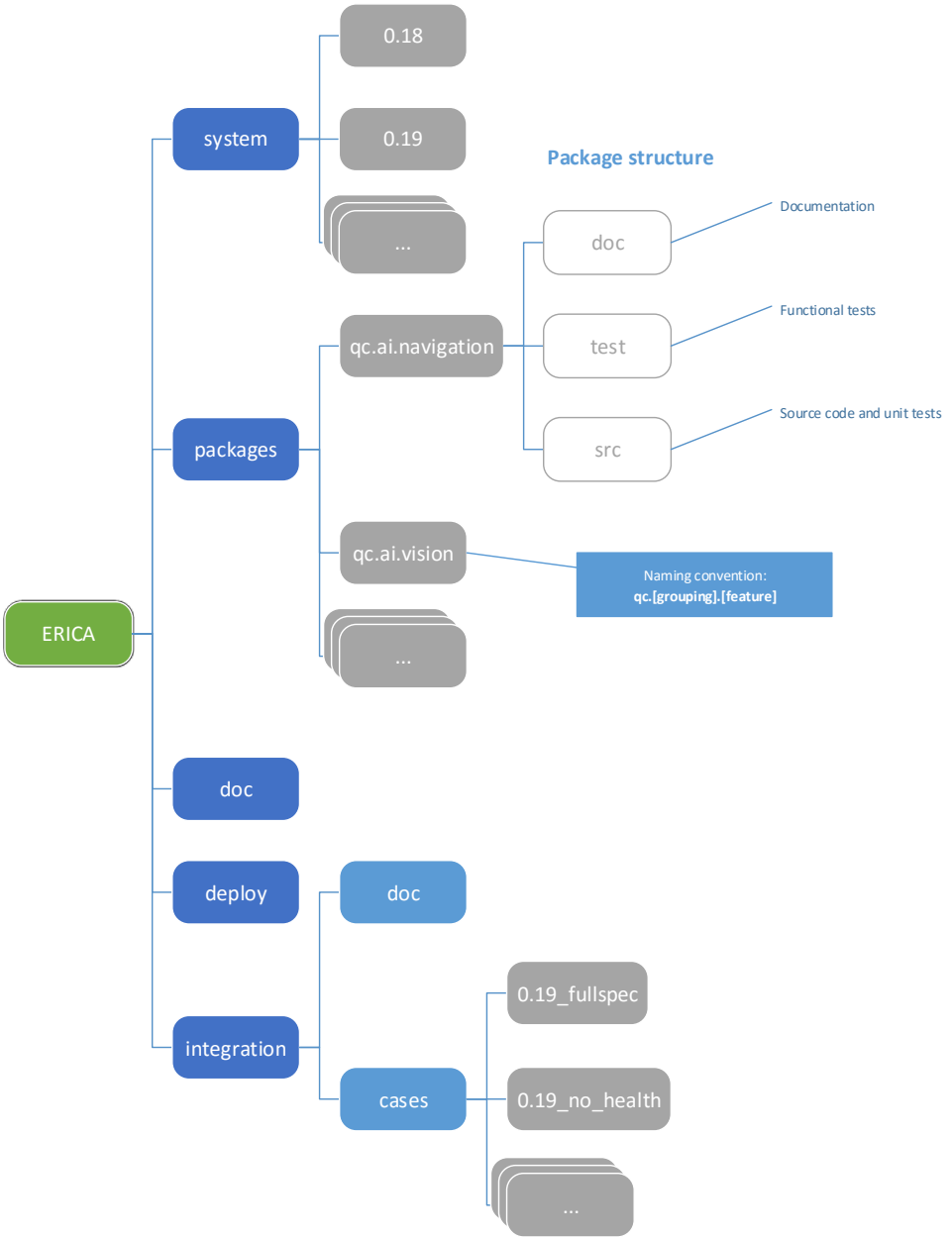
Under 'core' system, everything down from the management layer shown on the previous page is understood. This includes Unix OS updates and firmware upgrades crucial to the functioning of the core process. Subsystems are updated through packages, as they likely require highly specific or vendor dependent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



## 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*[6]) or must produce one on build

## Integration test development & automated testing

- For each deliverable product, an environment with specific test cases must be conditioned
    - Deliverable here means 'ERICA + feature set'
- Packages may at any point be pushed to the build server, which will run a full or partial test suite
- Once tagged, a package will automatically be pushed to the build server
- Before pushing to QA, a package must be tagged and build without warning

## Examples of packages

- Service
    - Updater, integrates package management and system updates with QC online
    - Sync, synchronizes local data storage with QC online
- Hardware interfaces
    - Vision, hearing, speech, movement, manipulation, direct-rf, WiFi, battery life sensor, display and monitoring devices.
- AI components: task scheduling, navigator, situational awareness
- Functionality modules: personal assistant, health, social

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

# Architectural patterns and styles

This chapter 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 don't 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.[7]

### 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.[8] 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.[9]

## Irrelevant patterns and styles

### Monolithic

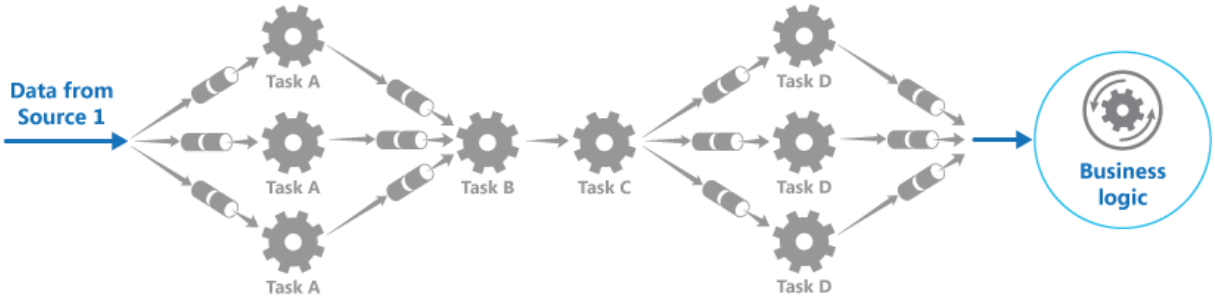
The monolithic pattern describes a system in which the UI and data-access code are combined in a single application from a single platform. This is not desirable for ERICA as we want it to be modular. This would also implicate that if a third party developer wanted to develop a piece of software for 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.

---

[7] https://en.wikipedia.org/wiki/Component-based_software_engineering
[8] https://en.wikipedia.org/wiki/Publish–subscribe_pattern
[9] http://wiki.ros.org/ROS/Tutorials/MultipleMachines

## Appendix A: Implementation reflection

Implementing the two required cases didn't go so well initially, as our new pipeline framework couldn't 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:




If you follow this approach literally, and see data as elements rather than streams, you can't 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're always pushing all data through all streams, which just doesn't 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 wasn't really possible unfortunately.

So what went wrong? Clearly the creators didn't 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'll 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 isn't their fault, it just shows that unless all architects interpret the concepts the same, you'll 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 doesn't *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's only the internal wiring. So as an architect, you'd 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'd skip the threading; web services, focus on retry and timeouts. If you really want to do it properly, you'd focus on messaging technology and resource allocation.

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