

1. For the text below, summarise the problem and suggest a software system architecture solution. The main stakeholders in this case are: - Software system architects - DevOps engineers - Developers

**Problem, context, and related work**

Robot Operating System 2 (ROS) is a popular and widely adopted robotics engineering framework. Its main goal is to provide robotics engineers a platform that allows for the development of robotics applications. Framework's infrastructure, which is open source, is based on numerous distributed packages. ROS supports a wide range of devices, starting from small embedded devices up to autonomous industry robots. ROS was developed with real-time capabilities in mind. Since this technology is in its infancy, the ROS community encourages industry and academic research. Academic research in ROS has covered many aspects of robotics development and has achieved quantifiable results regarding soft and hard real-time constraints, lowering latency, and in general, improving performance. However, because of its complicated vertical stack of technologies, there is still room where significant performance benefits could be extracted. The default way to extract performance with aim to lower execution time is by utilizing multi-cores. However, ROS with its complex infrastructure and requirement from robotics engineers to learn parallelization from C/C++ perspective, make it hard to utilize multi-cores properly. Therefore, utilizing multi-cores to lower execution time is one of the main motivators that guides research in ROS. By lowering execution time we gain performance benefits and allow other parts of ROS infrastructure, which are influenced by latency and real-time constraints, to be positively impacted as well. Robots are very heterogeneous devices. Therefore, robots may come with a vast array of different system requirements and system constraints. Some of the system constraints are critical when it comes to the correct functionality of the robots. These critical constraints deal with the overall performance of the robotics application. In the scope of overall performance of the robotics application, the main goal of robotics engineers is to lower execution time or make a particular function execute faster. To facilitate this, we refer to utilization of multi-core processors. However, required specialized knowledge regarding concurrency and expertise, which robotics engineers do not have, slows them down in building their robotics application and decreases efficiency. Robotics engineers are required to understand the architecture of the CPU and how it interacts with main memory. On top of that robotics engineers are required to understand the intricacies regarding caches and data locality. Complemented with this is a software stack that is utilizing the CPU. They need to learn how to use multi-threading libraries, which is not simple. Functions for creating and managing threads require thinking about multiple data-flows in software stack. As a consequence, robotics engineers are required to understand in low-level detail how multi-threading works and how prioritization, scheduling, and affinity combine to solve the problem of lowering execution time. To conclude, the main gap is that robotics engineers do not have an intuitive way of utilizing multi-cores to be able to lower execution time. They are required to know low-level details regarding parallelization and have to develop multi-threading applications. Additionally, robotics engineers do not have accessible interfaces to control Operating System mechanisms regarding prioritization, scheduler type, and affinity

**Requirements**

- SR.01: Framework to facilitate development of multithreaded applications in ROS2
- SR.02: Framework to assist robotics engineers in developing software that takes advantage of multi-cores in order to reduce average execution time, integrated in ROS2.
- SR.02: Pre-locking heap memory - The framework enables pre-locking of heap memory during initialization of the application in order to avoid failed memory allocations, which can occur if application requires significant amount of memory.
- SR.03: Process priority configuration - The framework provides interface for setting priorities of threads in order to avoid priority inversion.
- SR.04: Measurement of heap allocations - The framework offers a visualization on how many memory allocations happened, and it should be easily accessible to

the robotics engineers • SR.05: Measurement of latency. - The framework has a mechanism that measures latency • SR.06: Measurement of jitter - The framework has a mechanism that measures jitter • SR.07: Measurement of CPU caches usage & The framework has a mechanism that measures CPU cache usage • SR.08: Measurement of memory fetches. - The framework has a mechanism which measures memory fetches. • SR.09: Executor type - Robotics engineers write nodes' configuration as input to the framework. The framework outputs the best performing executor type configuration containing nodes. & ROS executor • SR.10: Node to process assignment - Robotics engineers write nodes' configuration as input to the framework. The framework outputs the best performing configuration. & node assignment, thread, process • SR.11: Allocator type - The framework accepts the message data types that are used throughout the application and it advises robotics engineers on which allocators to use. • SR.12: Easy to use user interface - All information and capabilities of the framework should be gathered in easy-to-use user interface. • SR.13: Guideline - Robotics engineers will be able to express their intentions of parallelizing the code in the framework. The framework will output guidelines for robotics engineers. • SR.14: Parallel Configuration - The framework will accept non parallelized code and will turn it into parallel configuration. & Parallel Configuration Quality requirements • SR.01: Increased CPU Utilization - The framework facilitates the development of multi-threaded applications, and as a result, increased CPU utilization. If the number of cores are  $n$ , (and  $n > 1$ ), then utilization is at least  $0.7 * n$ . • SR.02: Decreased Heap Allocation - The framework facilitates the mechanism with pre-locking memory during the start of the execution of the application. At least 50 percent decrease in heap allocations regarding memory is expected. • SR.03: Increased general performance of the application - The framework facilitates the mechanism for process priority configuration. We expect an increase in general performance in a region of 10 percent. • SR.04: Decreased average execution time - The framework facilitates the mechanism for decreasing average execution time by utilizing multi-threading. • SR.05: Increased CPU Cache utilization - We expect an increase in CPU Cache utilization by 30 percent. • SR.06: Decreased Memory Fetches - We expect to have a decreased memory fetches by 20 percent as a result of increased CPU Cache utilization. • SR.07: Increased productivity of robotics engineers - The facilities of the framework will output parallelized configurations. Compared with by-hand optimization, this process will be expected to take 30 percent less time. & increased productivity • SR.08: Usability - The framework will include an easy to use user interface. We expect robotics engineers to quickly understand the framework and use it efficiently. • SR.09: Modularity - The framework's design will closely follow ROS design philosophy. We expect the framework to be at least modular as ROS and to fit on top of ROS. & modularity Use cases • UC.01: Direct Inverse Kinematics - Sensors, Actuators, Moveable joints, Rotating Joints & Inverse kinematics is a field which concerns itself with correct positioning of the joints in order for end-actuator to reach desired position. As the name indicates, the positioning is worked out from the end-actuator backwards to the first-actuator. • UC.02: Detecting human faces and torso orientation - People in a scene, robot scanning the scene - Image processing is a very important field in robotics engineering. This use case tries to identify the way of doing image processing in robotics, so we can learn how to parallelize common functions. • UC.03: Kalman Filter & A specific robot does sensor fusion and uses kalman filter for prediction - Prediction of the measurements. Kalman filtering is a mathematical approach on how to improve our measures that come from the sensors. • UC.04: Assessing execution time of real-time loop - Real-Time Loops, Function, Time & Since ROS was built with real-time capabilities in mind, measuring real-time loop execution time can give us an insight on performance improvements.

Problem: Robotics engineers using the Robot Operating System 2 (ROS) framework face challenges in utilizing multi-cores to lower execution time due to the complex infrastructure and

requirement for low-level knowledge of parallelization. This slows down application development and decreases efficiency.

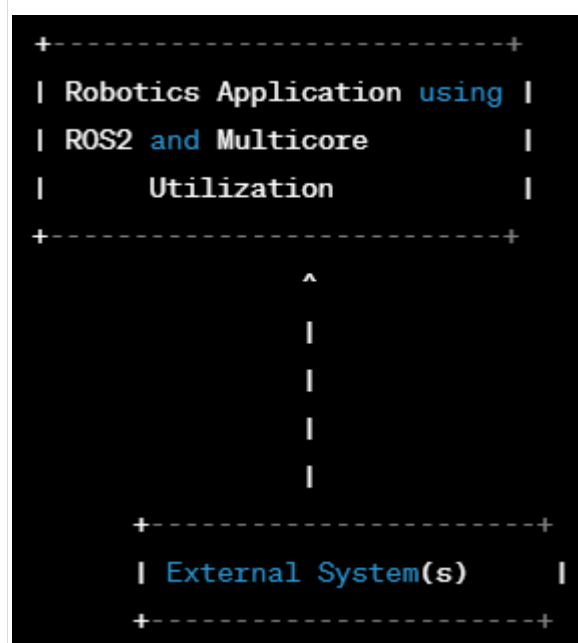
Solution: Develop a software system architecture solution that facilitates the development of multithreaded applications in ROS2, enabling robotics engineers to take advantage of multi-cores to reduce average execution time. The system should have a framework for pre-locking heap memory, process priority configuration, measurement of heap allocations, latency, jitter, CPU cache usage, and memory fetches. It should also offer guidance on executor type, node to process assignment, and allocator type, and provide an easy-to-use user interface. The system should aim for increased CPU utilization, decreased heap allocation, increased general performance of the application, decreased average execution time, increased CPU cache utilization, decreased memory fetches, increased productivity of robotics engineers, usability, and modularity.

Use Cases:

1. Direct Inverse Kinematics
2. Detecting human faces and torso orientation
3. Kalman Filter
4. Assessing execution time of real-time loop.

2. Suggest architecture of a conceptual solution for this problem. We need a C4 system context diagram (<https://c4model.com/#SystemContextDiagram>). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider the following: - Identify relevant external systems (if any) and add them to the diagram. - Identify relevant external stakeholders (if any) and add them to the diagram. - Identify high level abstract data types transferred between components and stakeholders. - Summarise data types transmitted between components. Add this information to the connections and interfaces between components.

Here is a C4 system context diagram for a conceptual solution to the problem:



In this diagram, the central box represents the Robotics Application using ROS2 and Multicore Utilization, which is the system we are designing. There are currently no external systems or stakeholders identified in this diagram.

Here are some possible high level abstract data types that could be transmitted between components and stakeholders:

- Sensor data: Data received from sensors in the robot, which may need to be processed by the application.
- Actuator commands: Data that the application sends to actuators in the robot, to control its movement or actions.
- Configuration data: Data provided by the robotics engineers, describing the nodes and their requirements for execution.
- Performance metrics: Data generated by the application, measuring execution time, latency, jitter, CPU cache usage, and memory fetches.

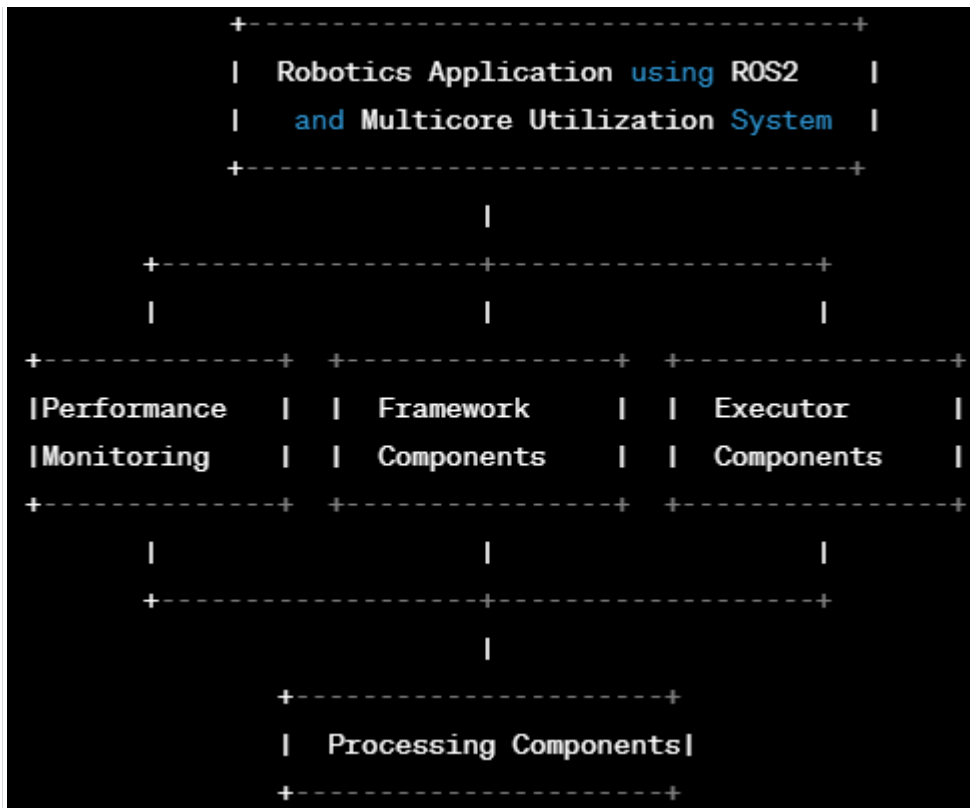
The connections and interfaces between components may involve the transmission of the following data types:

- Sensor data to the processing components
- Configuration data to the framework components
- Performance metrics to the monitoring components
- Actuator commands from the processing components

The system context diagram is intentionally kept simple to illustrate the core system being designed. As the solution is developed, more detailed diagrams can be created to show the internal components and their relationships.

3. Suggest architectural components that we need to solve this problem. We need a C4 container solution diagram (<https://c4model.com/#ContainerDiagram>). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider the following: - Define how the main stakeholder will use the solution. Add it to the diagram.. - Which quality properties are important? Suggest adequate architectural solutions to address them (architectural style, design patterns, design tactics, components and their relationships). Structure the components according to it. - Define data types transferred between components. - Summarise data types transmitted between components. Add this information to the connections and interfaces between components. - Define interfaces between components. Add information about required and provided interfaces to the components, along with the data types. - Document which anti-patterns we need to avoid when implementing this solution.

Here is a C4 container solution diagram for a conceptual solution to the problem:



In this diagram, the Robotics Application using ROS2 and Multicore Utilization System is represented as a container. There are three primary components that make up the system: Performance Monitoring, Framework Components, and Executor Components. The Processing Components represent the code that the robotics engineers are developing.

Here are some additional details on each component:

- **Performance Monitoring:** This component is responsible for measuring performance metrics like latency, jitter, CPU cache usage, and memory fetches. It provides visualizations of these metrics to the robotics engineers.
- **Framework Components:** This component provides a framework that assists robotics engineers in developing software that takes advantage of multicore processors. It includes mechanisms for pre-locking heap memory, process priority configuration, and parallelization guidance. It also advises on allocator type, executor type, and node to process assignment.
- **Executor Components:** This component provides an interface between the robotics engineers' code and the multicore processors. It manages the execution of the code across multiple cores to optimize performance.
- **Processing Components:** This component represents the code that the robotics engineers are developing. It includes the code for direct inverse kinematics, detecting human faces and torso orientation, kalman filtering, and assessing execution time of real-time loops.

The main stakeholder of this system is the robotics engineer, who will be using the system to develop robotics applications that take advantage of multicore processors to optimize performance.

Here are some important quality properties and architectural solutions to address them:

- Performance: To optimize performance, we are utilizing an Executor pattern that distributes the robotics engineers' code across multiple cores. We are also implementing a pre-locking mechanism to reduce memory allocation times, and providing guidance on allocator type, executor type, and node to process assignment.
- Modularity: To ensure modularity, we are following the ROS design philosophy and building our system to fit on top of ROS. The Framework and Executor components are loosely coupled to the Processing Components, allowing for flexibility and maintainability.
- Usability: To ensure usability, we are providing an easy-to-use user interface that displays performance metrics and visualizations. We are also providing guidance on parallelization to assist the robotics engineers in optimizing performance.

Here are some data types that may be transferred between components:

- Sensor data: Data received from sensors in the robot, which may need to be processed by the application.
- Actuator commands: Data that the application sends to actuators in the robot, to control its movement or actions.
- Configuration data: Data provided by the robotics engineers, describing the nodes and their requirements for execution.
- Performance metrics: Data generated by the application, measuring execution time, latency, jitter, CPU cache usage, and memory fetches.

The connections and interfaces between components may involve the transmission of the following data types:

- Sensor data to the Processing Components
- Configuration data to the Framework Components
- Performance metrics to the Performance Monitoring Component
- Actuator commands from the Processing Components

Here are some interfaces between components, along with information about required and provided interfaces and the data types involved:

- Performance Monitoring

4. Suggest architectural components that we need to solve this problem. We need a C4 component solution diagram (<https://c4model.com/#ComponentDiagram>). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider following:

- Define how the main stakeholder will use the solution. Add it to the diagram.
- Which quality properties are important? Suggest adequate architectural solutions to address them (architectural style, design patterns, design tactics, components and their relationships). Structure the components according to it.
- Define data types transferred between components.
- Summarise data types transmitted between components. Add this information to the connections and interfaces between components.
- Define interfaces between components. Add information

about required and provided interfaces to the components, along with the data types. -  
 Document which anti-patterns we need to avoid when implementing this solution.  
 Here is a C4 component solution diagram for a conceptual solution to the problem:



In this diagram, the Robotics Application using ROS2 and Multicore Utilization System is represented as a component. There are four primary components that make up the system: User Interface, Performance Monitoring, Framework Components, and Executor Components. The Processing Components represent the code that the robotics engineers are developing.

Here are some additional details on each component:

- User Interface: This component provides an easy-to-use interface for the robotics engineers to interact with the system. It displays performance metrics and visualizations, and provides guidance on parallelization.

- **Performance Monitoring:** This component measures performance metrics like latency, jitter, CPU cache usage, and memory fetches. It provides visualizations of these metrics to the robotics engineers.
- **Framework Components:** This component provides a framework that assists robotics engineers in developing software that takes advantage of multicore processors. It includes mechanisms for pre-locking heap memory, process priority configuration, and parallelization guidance. It also advises on allocator type, executor type, and node to process assignment.
- **Executor Components:** This component provides an interface between the robotics engineers' code and the multicore processors. It manages the execution of the code across multiple cores to optimize performance.
- **Processing Components:** This component represents the code that the robotics engineers are developing. It includes the code for direct inverse kinematics, detecting human faces and torso orientation, kalman filtering, and assessing execution time of real-time loops.

The main stakeholder of this system is the robotics engineer, who will be using the system to develop robotics applications that take advantage of multicore processors to optimize performance.

Here are some important quality properties and architectural solutions to address them:

- **Performance:** To optimize performance, we are utilizing an Executor pattern that distributes the robotics engineers' code across multiple cores. We are also implementing a pre-locking mechanism to reduce memory allocation times, and providing guidance on allocator type, executor type, and node to process assignment.
- **Modularity:** To ensure modularity, we are following the ROS design philosophy and building our system to fit on top of ROS. The Framework and Executor components are loosely coupled to the Processing Components, allowing for flexibility and maintainability.
- **Usability:** To ensure usability, we are providing an easy-to-use user interface that displays performance metrics and visualizations. We are also providing guidance on parallelization to assist the robotics engineers in optimizing performance.

Here are some data types that may be transferred between components:

- **Sensor data:** Data received from sensors in the robot, which may need to be processed by the application.
- **Actuator commands:** Data that the application sends to actuators in the robot, to control its movement or actions.
- **Configuration data:** Data provided by the robotics engineers, describing the nodes and their requirements for execution.
- **Performance metrics:** Data generated by the application, measuring execution time, latency, jitter, CPU cache usage, and memory fetches.

The connections and interfaces between components may involve the transmission of the following data types:

- Sensor data to the Processing Components
- Configuration data to the Framework Components

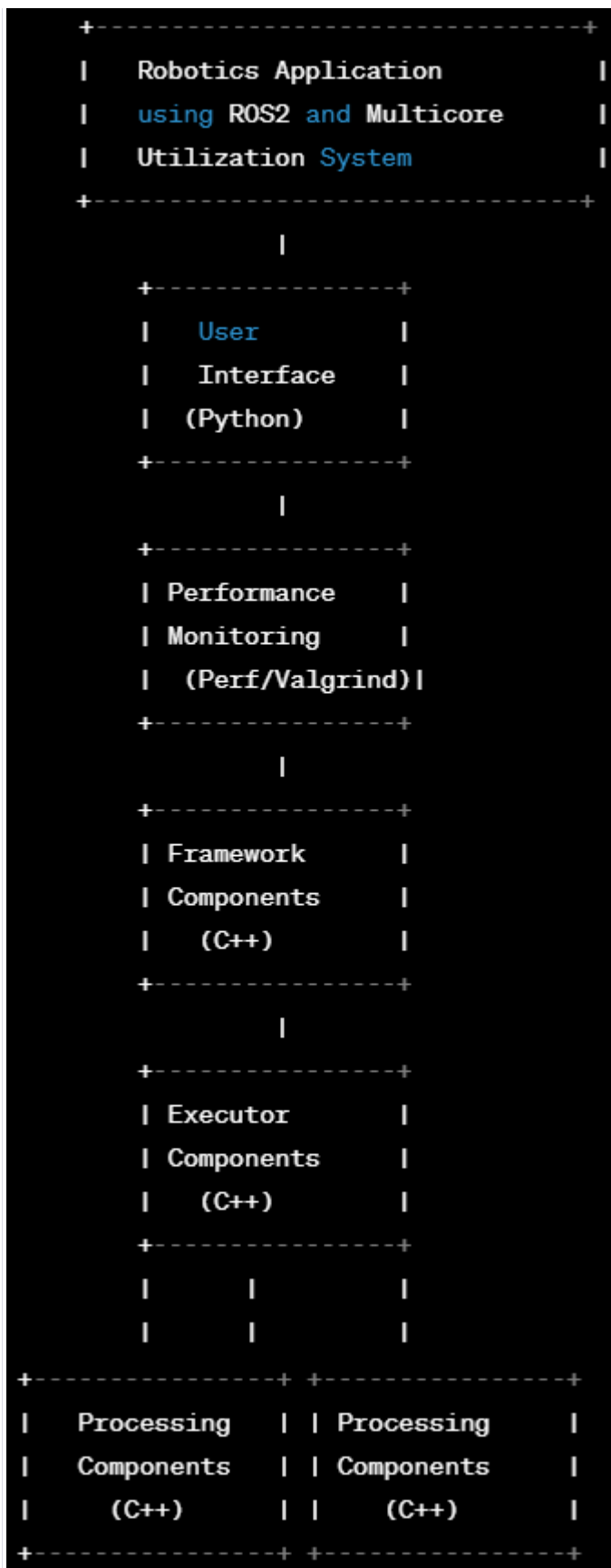


5. Please suggest a toolchain to implement the solution. Consider which tools we can reuse. For the components we need to develop, please suggest a programming language. Please use text (arrows and dots) to visualise the solution. Define data types transferred between components. Summarise data types transmitted between components. Add this information to the connections and interfaces between components.

To implement the solution, we can make use of existing tools and frameworks as well as developing our own components. Here is a suggested toolchain:

- ROS2: We can use ROS2 as the underlying framework for our system, which provides a platform for robotics application development and has been designed to be modular and distributed.
- C++: We can use C++ as the primary programming language for developing our own components, as it is a popular language for robotics applications and provides efficient memory management and performance.
- Python: We can use Python for scripting and for developing the User Interface component, as it is a popular language for scripting and has many libraries for developing user interfaces.
- Performance Monitoring Tools: We can use existing tools like Perf and Valgrind for monitoring performance metrics like CPU utilization, memory allocation, and cache usage.

Here is an updated C4 component solution diagram with the toolchain:



The data types transferred between components may include:

- Sensor data: Represented as sensor\_msgs in ROS, which is a standard message type for sensor data.
- Actuator commands: Represented as control\_msgs in ROS, which is a standard message type for actuator control commands.
- Configuration data: Custom message types that describe the nodes and their requirements for execution.
- Performance metrics: Custom message types that represent performance metrics like latency, jitter, CPU cache usage, and memory fetches.

The connections and interfaces between components may involve the transmission of the following data types:

- Sensor data to the Processing Components: Represented as ROS messages.
- Configuration data to the Framework Components: Custom message types.
- Performance metrics to the Performance Monitoring component: Custom message types.
- Executor commands to the Executor Components: Custom message types.