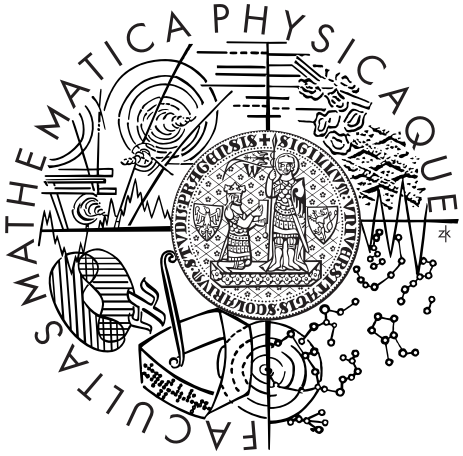
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Vladimír Matěna

Implementation of the DEECo component framework for embedded systems

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Tomáš Bureš

Study programme: Informatics

Specialization: Software Systems

Prague 2014

Thanks to:

Supervisor of this thesis who helped me with the design of the framework and review of the thesis text.

Department of Distributed and Dependable Systems for lending me STM32F4 boards and peripherals.

My grandfather who helped me with language corrections of the text.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date 30.7.2014 signature of the author

Název práce: Implementation of the DEECo component framework for embedded systems

Autor: Vladimír Matěna

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Doc. RNDr. Tomáš Bureš, PhD., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Současný vývoj distribuovaných a decentralizovaných cyber-fyzikálních systémů vedl ke vzniku modelu DEECo. Protože mnohá použití DEECo jsou vestavěné aplikace je zajímavé zvážit jeho použití DEECo na vestavěném hardware. V současné době existuje jen referenční implementace, která je napsaná v Javě a proto nemůže být použita pro vestavěné systémy. Jako součást této práce bylo navrženo mapování DEECo do C++ a zabudovaný framework CDEECo++ používající FreeRTOS pro plánování a synchronizaci. Ukázková aplikace, navržená pro STM32F4, demonstruje použitelnost frameworku. Tato práce obsahuje popis mapování DEECo do jazyka C++, zdrojové kódy frameworku CDEECo++, dokumentaci a ukázkovou aplikaci včetně základních měření jejích real-time vlastností.

Klíčová slova: cyber-fyzikální systémy, komponentový model, vestavěné systémy

Title: Implementation of the DEECo component framework for embedded systems

Author: Vladimír Matěna

Department: Department of Distributed and Dependable Systems

Supervisor: Doc. RNDr. Tomáš Bureš, PhD., Department of Distributed and Dependable Systems

Abstract: Recent development in the field of distributed and decentralized cyber-physical systems led to emerge of DEECo model. As many DEECo use cases are embedded applications it is interesting to evaluate DEECo on embedded hardware. Currently there is only reference DEECo implementation which is written in Java thus cannot be used for embedded applications. As part of this thesis C++ DEECo mapping and embedded CDEECo++ framework were designed using FreeRTOS operating system for task scheduling and synchronization. An example application designed for the STM32F4 board demonstrates usability of the framework. This thesis contains description of the DEECo mapping into the C++ language, source codes of the CDEECo++ framework, documentation and example application including basic measurement of its real-time properties.

Keywords: cyber-physical systems, component model, embedded systems

Contents

1. Introduction

In todays world controlling of physical entities with computer based systems is gaining larger and larger ground. So called cyber-physical systems are used in final products as well as in manufacturing processes. Area of their usage ranges from agriculture to aerospace and cover most industries. Cyber-physical systems design and implementation vary as their intended area of usage. There are lots of models and frameworks dealing with control of physical processes and entities.

This thesis deals with a specific class of cyber-physical systems. Systems in this class are distributed, dynamic and autonomous, but at the same time keep certain level of consistency. Such systems exist thanks to the recent development in the fields of networking, communication and widespread deployment of computers.

A model called DEECo is an example of this class of cyber-physical systems. It is model of component cyber-physical system. It enables creation of systems which are distributed and dynamic in both deployment and component communication. The DEECo component model is developed since 2012. In order to demonstrate its capabilities and perform large scale testing a Java framework which implements DEECo model has been developed.

DEECo consists of two types of objects called components and ensembles.

The Component consists of state information called knowledge and computation processes. The processes use knowledge data of components as their input and output. Independent nodes containing components are deployed to the environment and periodically broadcast knowledge belonging to local components.

The Ensemble is a dynamic group of components. The group formation is based on the cached knowledge of the components. As the nodes are independent and the whole system is fully dynamic and distributed the decision of the ensemble membership is taken by each node separately. Each node uses cached knowledge or remote components.

A component belonging to an ensemble can be either one of ensemble members or ensemble coordinator. Components inside the ensemble map parts of their knowledge from coordinator to members and vice versa. The knowledge mapping provides effective way of exchanging knowledge between components inside an ensemble. The restriction on knowledge exchange directly between members provides the system with certain level of consistency. The processes of the component can use the exchanged knowledge as their input. Thus the ability to exchange information inside ensemble allows processes hosted in the component to make decisions based on combined information collected by ensemble members.

As stated earlier in the text the only DEECo implementation is the JDEECo framework which is implemented in Java language. This implementation is intended to be deployed on PC style hardware with common networking solution such as IP based network. Primary usage of this framework is large scale laboratory testing of the concepts contained in DEECo and further research in the field of cyber-physical systems.

In order to achieve good code structure and variability, needed for rapid development and large scale testing, JDEECo framework uses various constructs of Java language including annotations and garbage collection. The implementation

also relies on the IP network to ensure communication between components.

The use of such technologies is not a limiting factor when the resulting application is deployed on the PC style hardware with appropriate networking solution. Unfortunately, the real life usage of cyber-physical systems is in some cases very different. There are use cases when it is crucial to deploy the system on the hardware which is restricted by size, price or power consumption. Moreover some use scenarios demand real-time properties of the framework in order to achieve full functionality and reliability.

Even when in many cases the disadvantages of the Java implementation can be removed or at last limited, there are applications which require wholly different design. This is especially true when the application is about to be deployed on embedded hardware where the restrictions of computational power and memory effectively disable usage of Java runtime. The runtime with garbage collection also seriously damages real-time properties of the whole system as most garbage collection implementations freeze the system while managing the heap.

1.1 Thesis objective

The target of this thesis is to provide design and sample implementation of the framework implementing the DEECo model. The framework should enable construction of the DEECo based embedded and real-time systems. The framework should also provide support for decentralized wireless communication between nodes hosting the components.

1.2 Problems with implementation

As the framework is intended to be used on embedded systems it cannot rely on full-feature OS running on the target hardware. Instead the framework should be ready to be deployed on bare metal or with limited support of additional libraries.

Recent embedded systems are becoming faster and faster. But still the computational power of such systems is much lower than the one of the general purpose systems. Moreover many use cases require low power consumption and low price of nodes hosting the DEECo components. This requires implementation to be efficient even at cost of limited variability. Generally the framework should maintain some level of variability, but at the same time it should not waste resources on features that are not essential.

In parallel with limited computational power of embedded systems there is also limited storage available. This relates to both RAM and Flash storage. The framework should not rely on space consuming libraries and should save as much RAM as possible for application and runtime data.

The system should also provide real-time guarantees. This seriously limits the selection of algorithms the framework can use. It complicates especially memory management as most memory allocation algorithms are not safe to be used in real-time applications.

1.3 Goals

- Design mapping of DEECo to language suitable for embedded development (Based on the analysis in the Chapter 3 the C++ was chosen as implementation language).
- Design embedded DEECo framework called C++ DEECo implementation (CDEECo++) using the mapping.
- Providing real-time guarantees to DEECo applications.
- Providing decentralized distributed communication based on MANETs.
- Sample implementation of the framework on modern hardware platform (STM32F4 development board will be used together with MRF24J40 radio, SHT1x sensor and UART GPS).

1.4 Document structure

This thesis describes the process of porting Distributed Emergent Ensembles of Components (DEECo) system to embedded environment. It starts with brief description of the basic DEECo system properties in order to give the reader some insight into the technology. These are covered in Chapter 2.

Brief description of the DEECo system is followed by Chapter 3 which contains analysis and use cases of the intended framework. The analysis also deals with the requirements on the implementation.

Next part of the document formed by Chapter 4 describes mapping of the DEECo structure into the C++ language. This chapter describes mainly how DEECo objects are represented by C++ classes. The trade-offs between full variability and implementation simplicity are also described here.

Mapping to C++ is followed by description of the CDEECo++ framework which implements it. Chapter 5 contains description of the framework internals, public interface and intended usage of the framework.

Chapter 6 contains analysis of the real-time properties of the CDEECo++ framework. It deals with scheduling, interrupts, priorities and also contains some measurements of the real system behavior.

Next Chapter 7 tries give some example of the real world usage of the C-DEECo++ framework. Both the source code and the description of the example system are included.

The last but one Chapter 8 deals other frameworks which are used or can be used to serve for the same or similar purpose as the CDEECo++ framework.

Conclusion is contained in Chapter 9. It summarizes what was done and what are the outputs of this thesis.

2. Background

2.1 DEECo principles

DEECo model is formed by two types of objects called components and ensembles. DEECo based applications are dynamic and distributed. Every node of the application may contain components and ensembles.

A component is set of processes and knowledge. The processes are computational tasks which are executed either periodically or triggered on knowledge change. The processes can read and write knowledge of the component. The knowledge of the components inside the node is distributed to other nodes. Basic knowledge distribution is performed by periodic broadcasts of knowledge data.

A node which hosts components and ensembles holds cached knowledge received from other nodes. The knowledge can be received directly from neighbour nodes or forwarded from distant ones.

An ensemble is a group of components. It is formed dynamically based on the knowledge of the components and membership function provided by ensemble. The decision whether the component is part of the ensemble is taken locally using knowledge of the component and cached knowledge from other components. A component present in the ensemble can be either its member or coordinator. The membership is resolved by periodical task which checks whether the local component is either coordinator for some cached member or member for some cached coordinator. Apart from membership function an ensemble also defines knowledge exchange function which is used to map information contained in knowledge from component to member and vice versa. Mapping from member to member is not allowed as it would cause inconsistencies. Mapping function on single node updates only the local components knowledge as the remote component will be updated by remote ensemble instance. It is not possible to update cached knowledge with ensemble output as there is no way of propagating possible changes back to source component.

The concept of ensembles and knowledge mapping between members and coordinators enables formation of dynamic groups. Members of these groups can share their state and make decisions based on shared state. As the information is shared through the coordinator and not directly between the members the result still keeps some consistency even when the process of mapping knowledge and ensemble membership is decided independently without explicit coordination or synchronization.

One of the DEECo concepts that requires closer look is the knowledge of the component. Basically the knowledge is state information of the component used by processes and subject of knowledge mapping performed by ensembles. For practical reasons the knowledge can be defined as tree structure where every node can either carry leaf data or contain another subtree. The tree structure is important as the DEECo processes can be triggered by subtree value change.

2.2 Running example

Search for submerged aircraft

In order to give the reader an example of an embedded DEECo application the following text contains running example which is also referenced in the rest of the thesis. Running example scenario deals with the coordination of robotic submarines on a mission of finding a crashed aircraft in the ocean. The described situation is a model intended to demonstrate DEECo usage, the real wreckage search may be a lot different.

There is an aircraft submerged deep in the ocean. The general location of the wreckage is known, but the accurate location is still unknown. The only clues to accurate location of the wreckage are ultrasound beacons, debris and possible sonar contacts. The search for the wreckage consists of successive improvements of guessed wreckage location which are provided by various sensors and other equipment. First the crash site is identified by debris floating on the ocean surface. Once the crash site is known the search continues with listening to beacons from planes black boxes and triangulation of the location from received signals. The final step is to find the wreckage on the sea floor using active sonar mapping.

In order to find the wreckage a fleet of drones of several types equipped with specialized sensors is deployed to the target location.

Submarines with sonar are submerged in the water and capable of scanning a small portion of the sea floor with active sonar and listening for beacons from black boxes. The communication between submarines and other drones uses sonar pulses as those are the only reliable means of underwater communication.

The buoys are floating on the water surface and listening to possible beacons from black boxes. Their secondary objective is also to relay submarine communication as they are equipped with both sonar and long range radio interfaces.

In order to provide radio communication relay and search for floating debris aerial drones are deployed in the area. Aerial drones are capable of detecting debris and oil spills on the water surface. These mark probable wreckage location. The location of the spills and debris can be used as start location for the search.

2.2.1 State of the art operation control

Deployment of drones in the search efforts for submerged aircraft requires also a deployment of a lot of human resources as the drones needs to be controlled and the obtained data needs to be analyzed. Usually each drone is controlled by a team of experts who decide about drone way-points and analyze data the drone has obtained. In order to coordinate the search efforts there is a central management which decides on asset deployment and analyzes the data on global level.

The central management specifies search location and partition in into several areas. Those are searched by independent teams using the remotely controlled drones. Once some object of interest is found the management moves more drones in the location to further improve the information. The subsequent search in the location may use some more accurate technology. For example when debris is found submarines may be moved to the scene in order to search the area with

sonar.

On one hand the information processing and control by human teams is effective approach to search operation. On the other hand humans limit speed of information processing and are prone to make errors. Furthermore the deployment of a lot of experts may be very expensive.

2.2.2 DEECo approach

Similar search operation can be controlled by DEECo based application. The application can be deployed on embedded hardware directly into the drones, so the drones need no external control. This approach removes need for human personal and speedup information processing. Naturally this comes at cost of limited variability and flexibility.

The first step in the DEECo deployment is identification of the components and ensembles. Fortunately the components are obvious in this example. Naturally, the types of components copy the drone types. Each drone type has its component hosted by the on-board embedded system. The processes of the component are responsible for collecting information and drone control. Knowledge of the component reflects information obtained by drone. A simple component controlling submarine is implemented in Java using code presented in Listing 2.1. The implementation is very basic. Such submarine would just sense for beacons and move to wreckage location set by ensemble. More realistic submarine component would be more complex and DEECo usage would not be so clear. The other components such as airplanes or buoys would be very similar. They just use different processes for different sensors.

Listing 2.1: Example of the submarine component in Java DEECo mapping

```
1  @Component
2  class Submarine {
3      // Knowledge
4      public int id;
5      public Position position;
6      public Position wreckage;
7      public float signalLevel
8
9      @Process
10     @PeriodicScheduling(1000)
11     public static void processPosition(@Out("pos") Pos pos) {
12         // Read the current submarine position
13         pos = Gyroscope.readPosition();
14     }
15
16     @Process
17     @PeriodicScheduling(5000)
18     public static void processBeacons(@Out("sig") float sig) {
19         // Get the strongest signal level
20         for(int f = MIN_FREQ; f < MAX_FREQ; f += FREQ_STEP) {
21             float level = Sensor.senseAtFreq(f);
22             if(sig < level)
23                 sig = level;
24         }
25     }
26
27     @Process
28     @PeriodicScheduling(50)
29     public static void processMotors(@In("wreckpos") Pos wreckage,
30             @In("position") Pos position) {
31         // No wreckage location known yet, stay in position
32         if(wreckage == null)
33             return;
34
35         // Set course to wreckage
36         float distance = calcDistance(wreckage, position);
37         float vector = calcVector(wreckage, position);
38         Motor.setPower(PIDMotorControll(distance));
39         Rudder.setAngle(PIDMotorControll(vector));
40     }
41 }
```

When the submersibles are deployed to the scene one of the challenges is the communication. The water blocks most of the radio signals so the sound, namely sonar, is the only option. Unfortunately the range of the sound communication may be limited. Also the range of radio communication may be limited for small drones floating in the ocean. For example the curvature of the earth may block the microwave signals.

The DEECo solves the problem with simple but effective approach of rebroadcasting the received knowledge. As the rebroadcasting system is not using the information about other nodes in area it is fully distributed and independent. It does not matter what other nodes are in range and what nodes disappeared. DEECo rebroadcasting also solves problems with communication between submarines as the buys will receive knowledge broadcasts using sonar and rebroadcast

the using radio. Other buoys will receive radio broadcasts and rebroadcast using sonar. This effectively allows submarines to talk to each other using radio relay provided by buoys. The same knowledge rebroadcasting can solve communication over long distances as also aerial drones can relay the knowledge broadcast for buoys.

Once the components were defined as natural reflection of drone types it is time to define ensembles. By definition ensembles are dynamic groups of components formed in order to exchange knowledge. In this scenario ensembles will be used to combine knowledge of members into a hint on wreckage location and to organize nearby drones to help in the search.

An example of knowledge combination that can be used as hint on wreckage location is triangulation of wreckage location based on reception of beacon by multiple drones. Such ensemble would consist of submarines in some limited area which have the reception of the signal. The knowledge mapping function is responsible for copying member positions and signal levels from members to coordinator. These will be combined into wreckage location by process of the coordinator. A simplified ensemble of this type may look like the one displayed in Listing 2.2.

Another type of ensemble to be used in this scenario is search coordination. Members of this ensemble are the drones in particular area. Positions of the drones are mapped from members to coordinator and planed member locations are mapped from coordinator to members. Coordinator's schedule process then decides on best member locations.

Listing 2.2: Example of the triangulation ensemble in Java DEECo mapping

```
1  @Ensemble
2  class Triangulation {
3      @Membership
4      public static boolean membership(
5              @In("member.position") Position memberPos,
6              @In("member.signal") float memberSignal,
7              @In("coord.position") Position coordPos) {
8          /**
9           * Member has good beacon reception and
10          * is close to coordinator
11          */
12         return memeberSignal > THRESHOLD &&
13             calcDistance(memberPos, coordPos) > MAX_DISTANCE
14     }
15
16     @KnowledgeExchange
17     public static void map(
18             @In("member.position") Position memberPos,
19             @In("member.signal") float memberSignal,
20             @InOut("coord.posis") Map<float, Pos> positions) {
21         positions.put(mamberSignal, memberPos);
22     }
23 }
```

3. Analysis

Design problems and implementation requirements

This chapter breaks the whole problem of framework design and implementation into subproblems as well as propose solutions applied to overcome them. The problems include direct requirements on the framework identified in the expected usage and some design choices and technical problems related to test environment.

3.1 Embedded system requirements

Purpose of this Section is to identify the the requirement on the DEECo implementation designed for embedded applications. It also proposes some technical solutions intended to satisfy those requirements. The requirements on the implementation are also crucial for selection of the programming language and runtime environment to be used for implementation of the CDEECo++ framework. As a DEECo is model of cyber-physical system it is interesting to evaluate possibility of deployment of DEECo based system directly into environment of interest. Desired solution of this problem is to deploy embedded hardware hosting DEECo components and ensembles as well as sensors and actuators. In order to identify the requirements on the implementation it is important to study some possible use cases of the system. An example described in Section 2.2 was used for this purpose as it contains many features usually found in the target applications.

One of the main requirements on embedded DEECo implementations is to deal with limited resources. Even when embedded hardware is gaining power the low-cost segment of the market has still limited memory and CPU speed. As the DEECo use cases expect deployment of large number of devices the cost per device should be minimized. In order to achieve this purpose the system should be able to run on slow hardware with limited memory footprint.

Expected usage of the embedded DEECo systems does not require storing large objects in the devices non-volatile storage. The data stored in the flash memory usually include only small configuration data. It is still possible that some application may require large objects to be stored as part of systems initial state, but such cases are not considered important for discussion of general DEECo usage.

Most of the non-volatile memory is expected to be occupied by application executable. Due to this limitation the implementation should be space efficient in order to keep enough space for user code including user defined ensembles and components. During the development it was discovered that the code size of the framework itself is not a major problem. Instead, the libraries used are more space consuming than the framework code itself. In order to achieve good space efficiency usage of external libraries should be limited to minimum.

3.2 Choice of programming language and runtime environment

The first and the most important choice for every project that involves writing code is selection of the programming language. The choice of language and especially the choice of runtime is directly affected by the embedded nature of the target system. As the languages are often binded to particular runtime environment the choice of the runtime environment is dependent on the choice of the language.

The options for embedded development are quite rich. When taking into account only the mainline products there are these options: Java Embedded, .NET Compact and Micro frameworks, C, C++ or some framework based on C++ such as Qt Embedded. When choosing a runtime and language for implementation it is important to take account structure of the already existing JDEECo framework in order to be able to reuse its design or even some code. On the other side features of the framework such as memory management, object oriented properties and available libraries can significantly affect implementation complexity.

The main choice was whether to use a low level language such a C++ and C or to use some of the high level JIT compiled language such as Java or C#. High level languages have the advantage of being able to express DEECo in more elegant way using reflection. Java usage would even enable some code sharing with JDEECo which would simplify implementation a lot. On the other hand the low level languages such as C and C++ provide fine control on the instruction level as well as complete control of memory management. Both are important in order to develop efficient and real-time enabled applications.

After evaluation of all possibilities it was decided to use C++ language as it allows to keep tight control of generated code and use classes to model DEECo components and ensembles. It also supports using C libraries and assembly code which are important for direct hardware control. This is especially important as the example implementation with the STM32F4 board uses C library and assembly code to access and initialize hardware.

Once the language was decided the choice of runtime was less complicated as there were not so many choices. The runtime consists of standard C Library and C++ Standard Template Library including standard memory allocator present in the *newlib* library.

3.3 Real-time processing

As many cyber-physical systems contain tasks that require real-time processing the whole framework should meet standards of real-time systems. Even when internal DEECo processes are not required to be real-time, other user defined process in the same system may require real-time scheduling. As DEECo allows user to define triggered and periodic tasks the whole system of handling knowledge changes should be real-time to allow user rely on DEECo tasks in real-time enabled applications. Also the system itself contains tasks in order to handle data processing and knowledge rebroadcasting. These task may lock some shared resources and thus block user tasks from executing. It is important to achieve short

an deterministic execution times for those tasks in order to guarantee real-time properties of the user defined tasks.

Some of the commonly used practices in programing harm real-time properties of the system. These are especially managed memory with unbounded execution of garbage collection as well as many classical memory allocation algorithms. Moreover critical parts of the system should relay only on algorithms with deterministic execution time as those parts of the system may prevent time critical user processes from executing.

The requirement of real-time processing do not affect only choice of runtime and algorithms. These requirements also demand the system to contain task scheduler. Moreover the scheduler should be also real-time enabled. The options were either to implement scheduler from scratch or to use some library implementing real-time operating system.

As the development of the reliable real-time scheduler is time consuming and complex task and there are plenty of implementations to choose from, it was decided not to implement new scheduler. Moreover using well known real-time operating system has also the benefit of the wide hardware support at no extra cost.

After evaluation of options in the field of small embedded library-like operating systems the choice was made to use FreeRTOS. It is simple operating system for use in embedded and real-time systems. It provides basic functions for task creation, task scheduling, timers and synchronization primitives. The whole system acts a bit more like a library than full operating system. The FreeRTOS supports a lot of different hardware. The hardware support is achieved using sort of plug-ins called ports. These plug-ins implement core functionality specific to the CPU. Fortunately there is a plug-in to be used for ARM Cortex-M4 MCU used by framework implementation on the STM32F4 board.

3.4 Distributed wireless communication

Another key aspect of the embedded DEECo implementations is communication. The JDEECo framework relies on IP based communication which requires configuration and some sort of central control. When deploying nodes into environment the need to properly configure their networking interface may be limiting factor. Sometimes the accurate location of the nodes is not known in advance, sometimes the location changes. Moreover the group of the nodes taking part in the communication changes dynamically. Thus technologies like WiFi or even wired Ethernet are of no use.

Embedded framework with nodes deployed directly into environment should use some sort of packet based wireless communication media. In order to enable communication over large distances which are not in the range of radio interface the system should allow data packets to be relayed. The relay mechanism can be as simple as rebroadcasting of received knowledge. The more complex mechanism encounters relaying data based in their content as described by proposed future versions of the DEECo.

Another approach that can be used in parallel with rebroadcasting by embedded nodes is to use backbone network to transport knowledge over long distances. This approach is out of scope of the embedded system itself, but the systems de-

sign should be ready for this option. A backbone network may be wireless or wired and is expected to be configured and centralized. Relaying knowledge over the Internet is one of the possible implementations. The backbone network may be accompanied by back-haul mid-range wireless network that can improve transport of the knowledge in particular area. The back-haul network is expected to be IP based or with similar design which would enable smart routing of the data.

The basic requirements on communication relay can be satisfied by some sort of MANET. As the smart MANET network is difficult to implement and network communication is not the main goal of this thesis it was decided not to implement complex MANET system. Instead of dynamic routing the system will rebroadcast the message received with delay calculated from received signal quality as proposed by DEECo. Implementation of smarter routing as proposed be DEECo requires a lot of work and testing in order to achieve relevant results. Unfortunately, in order to perform large scale testing the deployment of a lot of hardware or complex simulation would have to be used.

The IEEE 802.15.4 radio was used as wireless interface. It provides simple broadcasting and reception of packets and can be configured to work without centralized network elements. There are more options that implement the same or similar functionality such as Bluetooth, IrDA, sonar or mesh enabled Wifi. The 802.15.4 was used as the hardware was available when the framework was implemented.

In order to provide wider range of wireless interface it was decided to design the framework in such way so that more network interfaces of various types can work in parallel. This allows both using another wireless interface and connection of more different interfaces in order to provide backbone or back-haul network broadcasting and reception.

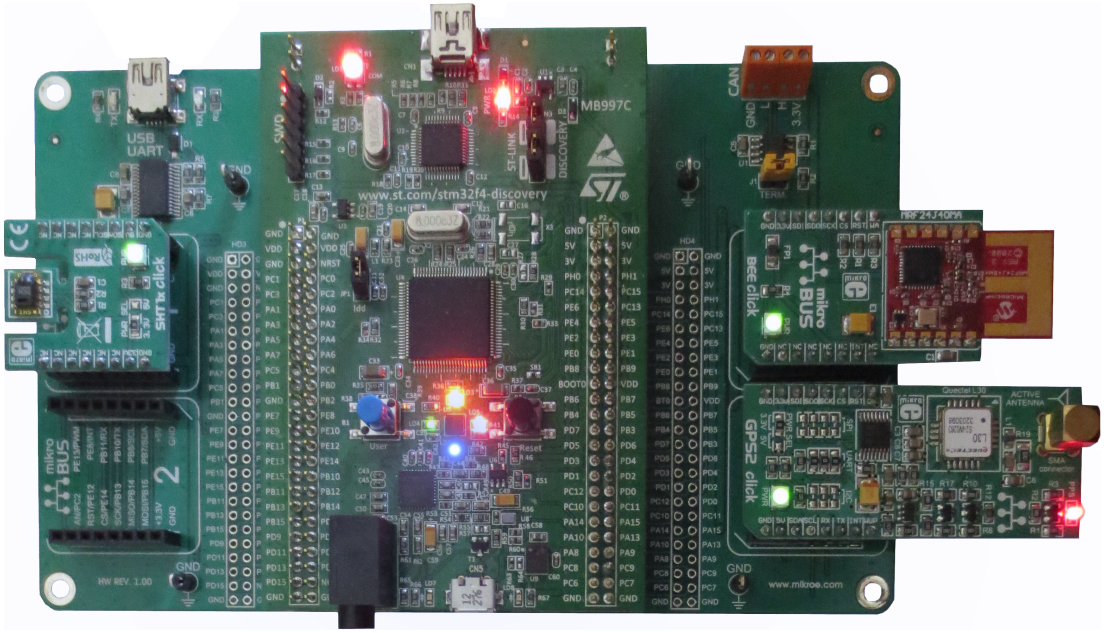
3.5 Testing environment

As the development of the framework do not consist only from design but also from testing a test environment had to be chosen. The designed CDEECo++ framework is intended to be used on variety of different hardware. As the large scale testing on broader range of hardware is not subject of this thesis single hardware platform was chosen as a sample. The implementation is deployed on the STM32F4 [1] development board accompanied by STM32F4 discovery shield which, among others, supports USB serial communication with PC and 4 Mikrobus slots.

The board contains many interesting components. Features used by test framework implementation include 168 MHz ARM Cortex-M4 MCU, FPU, 1 MiB of Flash, 192 KiB of RAM, general purpose IO ports, UARTs and timers. As the framework supports RF communication the MRF24J40 IEEE 802.15.4 module[2] was attached to one of the Mikrobus slots. Moreover SHT1x[3] temperature and humidity sensor and UART GPS[4] were attached to two more Mikrobus slots in order to provide some data that can be used by sample components to demonstrate their functionality. Photo of the testing hardware is displayed in

[1] `http://www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1577/LN11/PF252140`
[2] `http://www.mikroe.com/click/bee`
[3] `http://www.mikroe.com/click/sht1x`
[4] `http://www.mikroe.com/click/gps2`

Figure 3.1: Photography of the testing system node



the Figure 3.1. The used hardware is not chosen to represent the intended usage of the framework. Instead it was used as it was available and provides ways of testing framework ability to process input and output.

3.6 Structure of the solution

The aim of the solution is to implement framework based on DEECo model to be used on embedded hardware and provide sample implementation for STM32F4 board. The solution consist of drafting new system design based on DEECo, defining a mapping of DEECo objects to the target language, implementation of the framework, setup of the runtime environment, implementation of a sample project and documentation of newly created source code.

DEECo on embedded systems design

The analysis of the DEECo based embedded system include definition of the requirements and possible problems and solutions. This is covered by discussion in this chapter as well as by parts of the Chapter 4.

DEECo mapping to framework language

Mapping of the DEECo objects to the framework implementation language is described in the Chapter 4. Mapping is also covered by Doxygen[5] generated documentation and sample usage example code.

[5] `http://www.doxygen.org`

Framework implementation

Framework implementation consists of framework sources and documentation. The documentation is partially included in this thesis in the form of user and programmer documentation included in the Chapter 5 and Doxygen documentation generated from code comments.

Runtime setup

System setup and runtime environment consists of the C++ runtime, FreeRTOS and STM32F4 peripheral library are described mainly in the Chapter 5. The code needed to bootstrap the framework on the STM32F4 board is included in the project.

Usage sample

Sample components and ensembles are described in Chapter 7. Sample code is part of the project and is also covered by Doxygen generated documentation.

4. DEECo to C++ mapping

This chapter describes mapping of DEECo concepts into the C++ language. Description and examples here are intended to explain the mapping itself, but do not focus on the usage. More user oriented description with real-life examples is provided in Section 5.3.

4.1 General concepts

4.1.1 From Java to C++

When a new language mapping is being defined it is good idea to consult already existing mappings. Currently there is DEECo mapping defined for the Java language. When new mapping into C++ language was designed the focus was applied on keeping things same or similar where possible. Similarities between mappings almost always helps in better framework adoption and greater usability of the whole system. Unfortunately Java language offers constructs that cannot be easily reused in C++. Moreover the desired embedded usage adds real-time requirements and demands less complicated design than the one used in Java mapping.

In order to satisfy new requirements on the mapping it was decided to simplify the design and thus reduce flexibility of the mapping. The main simplification was performed by limiting task input and output knowledge. As the embedded system is not expected to contain large knowledge, every task receives full knowledge as input. Similarly the output knowledge is limited to single subtree of the original knowledge as the output is expected to be simple. Such reduced flexibility should not be limiting factor to embedded usage. Additionally an application can achieve multiple outputs by explicit calls to CDEECo++ API.

4.1.2 Design

Consistency with current mapping

In general the C++ mapping should allow creation of the system which satisfies two main requirements. Objects should be instantiated without dynamic memory allocation and the system runtime overhead should be as small as possible. These two requirements on the mapping forced quite heavy usage of templates in order to avoid dynamic memory allocation and limit runtime complexity.

Major differences

The DEECo C++ mapping is similar to the Java mapping in the fact that primary objects of interest components and ensembles are in both cases represented by classes. The C++ mapping goes further and maps also component's process to classes. This decision allows the system to solve process input and output types with templates and enables natural usage of process classes as private task local storage. Thus the component's task can access three types of data. The

knowledge of the component which is considered to contain some public component data. In addition to the knowledge user can define private fields on the component class in order to create component local storage. Additionally and user can define private fields on process class in order to create task local storage. The local storage which is not managed by the system can be used for various handles and state information. These are often required for low level hardware access used in embedded applications.

Knowledge type

After evaluation of several possibilities the decision was made to define component knowledge as plain C++ structure. This approach has some limitations compared to wrapping knowledge parts into richer class structure, but it simplifies user defined code and speedup execution.

Process parameters

Templates are heavily used in order to provide type safety for processes and ensemble mapping functions. As the components and ensembles have the input and output knowledge types as their template arguments the user implemented functions can receive their inputs and return their outputs directly as knowledge structure members.

System objects

The C++ mapping describes classes the user can inherit from in order to implement components, ensembles, triggered and periodic tasks. The mapping also specifies templates intended for implementation of knowledge caches and system object that is used to connect all parts of the application together.

4.2 Component

Component is mapped to C++ class which inherits from the *CDEECO::Component* with knowledge type as template argument. *CDEECO::Component* base class is responsible for knowledge management including keeping knowledge data, knowledge access and knowledge fragment creation. Knowledge fragments are broadcast using communication interface provided to the class constructor. Component class also defines component type and identification numbers. Those are passed together with system object reference to base class constructor. A sample component definition is presented in Listing 4.1.

As the component class is user defined it can contain user defined fields specific to the application such as various handles. These are considered to be component's local data which are not managed by the CDEECo++ system. Beside from user defined data the component is expected to contain instances of its processes.

Listing 4.1: Example component in C++ DEECo mapping

```
1  class Component: public CDEECO::Component<Knowledge> {
2  public:
3      // Component type
4      static const CDEECO::Type Type = 1;
5
6      // Possible process objects here
7
8      // Possible user data here
9
10     Component(CDEECO::Broadcaster &brdcastr, const CDEECO::Id id) :
11             CDEECO::Component<Knowledge>(id, Type, brdcastr) {
12         // Knowledge initialization here
13     }
14 };
```

4.2.1 Knowledge

The key part of the DEECo C++ mapping is representation of the component's knowledge. The knowledge as defined by DEECo is tree structure containing public component data. As virtually all parts of the system handle knowledge or its parts it is important to make knowledge representation performance efficient. The system also has triggered task support that use knowledge change as trigger. Thus there has to be easy way how to detect knowledge subtree changes.

After evaluating several options it was decided to implement knowledge as single C++ structure. The structure has tree layout and accesses to the subtree can be identified by offset in the structure. In order to enforce type safety the knowledge structure has to inherit from *CDDECO:Knoweldge*. Another approach would be to represent the knowledge as explicit tree structure composed of nodes where each node value would be accessed by getters and setters. Unfortunately this approach complicates direct access to knowledge data when knowledge parts are being broadcast.

Listing 4.2: lst:Example knowledge in C++ DEECo mapping

```
1 // Component knowledge
2 struct ExampleKnowledge: CDEECO::Knowledge {
3     int value;
4     float anotherValue;
5 };
```

Surprisingly the knowledge broadcasting is closely related to C++ mapping of the knowledge. As the radio interface sometimes cannot broadcast the knowledge in one packet it is needed to break knowledge into knowledge fragments. The process of breaking knowledge into parts and combining parts together is described in detail in framework description which is explained in Chapter 5. The importance of this fact for the mapping is that the allowed offsets of broadcast data can be defined for the knowledge. These offsets mark the consistent fragments of the knowledge. User can control offsets in order to force consistency in between some knowledge members. Offsets are defined using a trait structure for the particular knowledge type. The trait defines *std::array* which describes the allowed broadcast offsets.

Listing 4.3: Example knowledge trait in C++ DEECo mapping

```
1  // Allowed offsets to guarantee knowledge consistency
2  namespace CDEECO {
3      template<>
4      struct KnowledgeTrait<ExampleKnowledge> {
5          static constexpr std::array<size_t, 2> offsets = { {
6              offsetof(Knowledge, value),
7              offsetof(Knowledge, anotherValue),
8          } };
9      };
10     constexpr decltype(KnowledgeTrait<ExampleKnowledge>::offsets)
11     KnowledgeTrait<ExampleKnowledge>::offsets;
12 }
```

4.2.2 Process

Similar to component itself a component's process is mapped to C++ class. User defined process class inherits either from *CDEECO::PeriodicTask* or from *CDEECO::TriggeredTask* base classes. The base classes have knowledge type, task output and optionally task trigger types as their template arguments. The process class has to implement virtual run method which contains user defined task code. Run method parameter and return types are defined by base class template arguments. When the process should not return anything then the output type can be set to *void*. The process class is expected to contain user defined fields which can hold various task specific private data. These private data are not managed by framework. Base class constructor has to be called with references to task's parent component, output knowledge and optionally with the reference to trigger knowledge or periodic task period.

Listing 4.4: Example periodic process in C++ DEECo mapping

```
1  // Periodic process
2  class ExampleProcess:
3          public CDEECO::PeriodicTask<ExampleKnowledge, int> {
4  public:
5      ExampleProcess(auto &cmpnent):
6              PeriodicTask(3000, cmpnent, cmpnent.knoweldge.value) {
7          // User defined task initialization code here
8      }
9
10 private:
11     // User defined task local data here
12
13     // User defined task code executed each 3000ms
14     int run(const ExampleKnowledge in) {
15         // Compute and return new value
16         int newVal = in.value;
17         return newVal;
18     }
19 };
```

4.3 Ensemble

Ensemble C++ mapping is very similar to periodic process mapping. It is just a bit more complicated as it defines membership function and two mapping functions. First one maps from member to coordinator and second one maps from coordinator to member. User defines an ensemble as class that inherits from *CDEECO::Ensemble* with coordinator knowledge type, coordinator output type, member knowledge type and member output type as template arguments. The base class has virtual methods for membership determination, member to coordinator mapping and coordinator to member mapping. Implementation of both mapping functions is mandatory, but the user can use base class template arguments to set one or both mapping functions' return type to void. Doing so effectively disable selected method's effect on knowledge and the implementation may not call these methods at all.

Listing 4.5: Example ensemble in C++ DEECo mapping

```
1  // Typedef output types in order to clarify their usage
2  typedef int CoordOutType, MemberOutType;
3
4  // Typedef ensemble type as it is complicated
5  typedef CDEECO::Ensemble<
6          CoordKnowledge,
7          CoordOutType,
8          MemberKnowledge,
9          MemberOutType
10     > EnsembleType;
11
12 class ExampleEnsemble: EnsembleType {
13 public:
14     static const auto PERIOD = 5000;
15
16     // Constructor used on node where coordinator is hosted
17     ExampleEnsemble(auto &coord, auto &lib):
18         EnsembleType(&coord, &coord.knowledge.val, &lib, PERIOD) {
19     }
20
21     // Constructor used on the node where member is hosted
22     ExampleEnsemble(auto &member, auto &library):
23         EnsembleType(&member, &member.knowledge.val, &lib, PERIOD) {
24     }
25
26 protected:
27     // Membership function
28     bool isMember(
29         const CDEECO::Id coordId,
30         const CoordKnowledge coordKnowledge,
31         const CDEECO::Id memeberId,
32         const MemberKnowledge memberKnowledge) {
33         // Determine ensemble membership
34         return true;
35     }
36
37     // Map from member to coordinator
38     CoordOutType memberToCoordMap(
39         const CoordKnowledge coordKnowledge,
40         const CDEECO::Id memberId,
41         const MemberKnowledge memberKnowledge) {
42         // Calculate coordinator output value
43         return 42;
44     }
45
46     // Map from coordinator to member
47     MemberKnowledge coordToMemberMap(
48         const MemberKnowledge memberKnowledge,
49         const CDEECO::Id coordId,
50         const CoordKnowledge coordKnowledge) {
51         // Calculate member output knowledge
52         return 42;
53     }
54 };
```

5. CDEECo++ framework

This chapter deals with the framework implementation including the testing project which was used during development. It includes programmer documentation for the test project and the framework. Framework structure and usage description was also covered in order to help other adopters of the framework.

5.1 Used software

Even when the framework and the test environment were developed with standards in mind it is not always possible to maintain compatibility with wide range of tools. In some places experimental features such as C++14 were used in order to produce simpler and more readable code. In order to allow reproduction of the outputs of this thesis the following text provides versions and settings for tools used during the development.

5.1.1 Toolchain

A Toolchain is the core of every development process. It consists of basic utilities and libraries such as compiler, linker and debugger. It is even more important for embedded projects as those require cross-compiler setup. The toolchain used for testing and development of the framework on the STM32F4 board was created using *crossdev* application using these commands:

```
$ crossdev --target armv7m-hardfloat-eabi --ex-gcc --ex-gdb
```

Unfortunately this great tool is Gentoo linux[1] specific as it uses the systems package installer to build the toolchain. As a replacement toolchain provided by ST[2] company or custom build toolchain with the `armv7m-hardfloat-eabi` target triplet can be used. Hardware floating point is essential for framework code to work as FreeRTOS requires it. These versions of software were included in the toolchain used for development of the framework and testing application.

- binutils version: 2.24-r3 with C++ support
- gcc version: 4.9.0 with C++ support
- gdb version: 7.7.1 with XML support
- newlib version: 2.1.0

5.1.2 Libraries

Aside from *newlib* library which is considered to be part of the toolchain the framework uses FreeRTOS operating system and STM32F4 peripheral library. FreeRTOS is a simple operating system which comes in form of a library. It

[1] `http://gentoo.org`
[2] `http://www.st.com`

contains scheduler and basic synchronization primitives. Due to nature of this library the sources and other needed files found in FreeRTOS were integrated directly into the project. The STM32F4 peripheral library was also included in the project as its nature requires some files to be modified in order to suit the application usage.

- FreeRTOS version: 8.0.0
- STM32F4xx DSP and Standard Peripherals Library version: 1.3.0

5.1.3 Other tools

In general many very different tools ranging from hex editor to multimeter were used in development of the framework. Brands and versions of most are not important enough to be noted here. The two tools where version can be important are Eclipse development environment and On-Chip debugger software which was used to debug and flash the application on the STM32F4 development board.

- Eclipse SDK version 4.3.2 (Kepler Service Release 2)
- Eclipse CDT plug-in version: 8.3.0
- Eclipse CDT plug-in C/C++ GDB Hardware Debugging version: 8.3.0
- OpenOCD version 0.8.0 with USB support

5.1.4 Provided project layout and drivers

In order to speedup development a project called *beeclickarm* was used as a base for the implementation. The *beeclikcarm* project focuses on exposing STM32F4 sensors to a application running on general computer via serial link. Slightly modified versions of some drivers and build script were taken from this project and incorporated into the test application. The original project can be found at Github[3].

5.2 System setup

This Section describes how the system is compiled and deployed on the STM32F4 development board. The project includes example usage of the framework. This example application is described separately in Chapter 7.

[3] `https://github.com/bures/beeclickarm/tree/master/beeclickarm`

5.2.1 Source code structure

This is the basic structure of the project source code. Individual files are described in their headers and should contain comments that clarify their purpose.

```
/ ..... Project repository root
├── Doxyfile ..... Doxygen configuration
├── FreeRTOS ..... FreeRTOS files
├── KnowledgeSamples ..... Knowledge data samples
├── Makefile ..... Make configuration
├── STM32F4xx_DSP_StdPeriph_Lib ..... STM32F4 peripheral library
├── build ..... Build directory
├── doc ..... Doxygen generated documentation
├── src
│   ├── FreeRTOSConfig.h ..... FreeRTOS configuration
│   ├── cdeeco ..... CDEECo++ framework sources
│   ├── drivers ..... Hardware drivers
│   ├── main.cpp ..... Example application
│   ├── main.h ..... Example application
│   ├── test ..... Example components and ensembles
│   └── wrappers ..... FreeRTOS C++ wrappers
├── stm32_flash.ld ..... Linker script for STM32F4 CPU
└── tty ..... Linux script for serial console
```

5.2.2 Application setup

The CDEECo++ application is expected to follow template described in usage example which is described in Chapter 7.

In general the user is expected to implement *CDEECO::Radio* interface and create instance of it. Then the user should instantiate *CDEECO::System* with the provided radio implementation. Once the system object is created then the DEECo components inheriting from *CDEECO::Component* can be instantiated. The *CDEECO::KnowledgeChache* instances may also be instantiated and passed to the system object in order to provide caching of remote knowledge. Once the caches and components are in place the ensemble implementations inheriting from *CDEECO::Ensemble* can be instantiated as their constructor requires reference to caches and components. As the last step user program is expected to start FreeRTOS scheduler. Starting the scheduler will start the tasks defined by CDEECo++ system.

5.2.3 Deployment

The deployment of application and the framework is the same as common deployment of embedded C/C++ application. It consists from compiling the application and framework sources using cross-compiler for target architecture. Linking objects together and creating single application image. Once the binary image is ready it is flashed into the embedded hardware memory.

The deployment of the example project on the STM32F4 board is performed using `make` program. The configuration Makefile is placed in the project root.

The makefile may need to adjust tools names and paths. The defaults are set to:

```
OPENOCD=openocd
CC=armv7m-hardfloat-eabi-gcc
CXX=armv7m-hardfloat-eabi-g++
OBJCOPY=armv7m-hardfloat-eabi-objcopy
SIZE=armv7m-hardfloat-eabi-size
```

Once the paths are set the project can be complied, linked and flashed to the connected STM32F4 board by issuing these commands in the project's root directory:

```
$ make clean
$ make
$ make flash
```

In order to speedup development the project can be built in parallel. The parallel build was tested with 9 tasks and produced significantly faster builds on 8 core CPU. The AMD FX8350 CPU was able to build the project in 2 seconds when building in parallel but it took over 12 seconds to build the project using single thread. In order to run 9 parallel build threads these commands may be used:

```
$ make clean
$ make -9
$ make flash
```

5.3 Structure of a CDEECo application

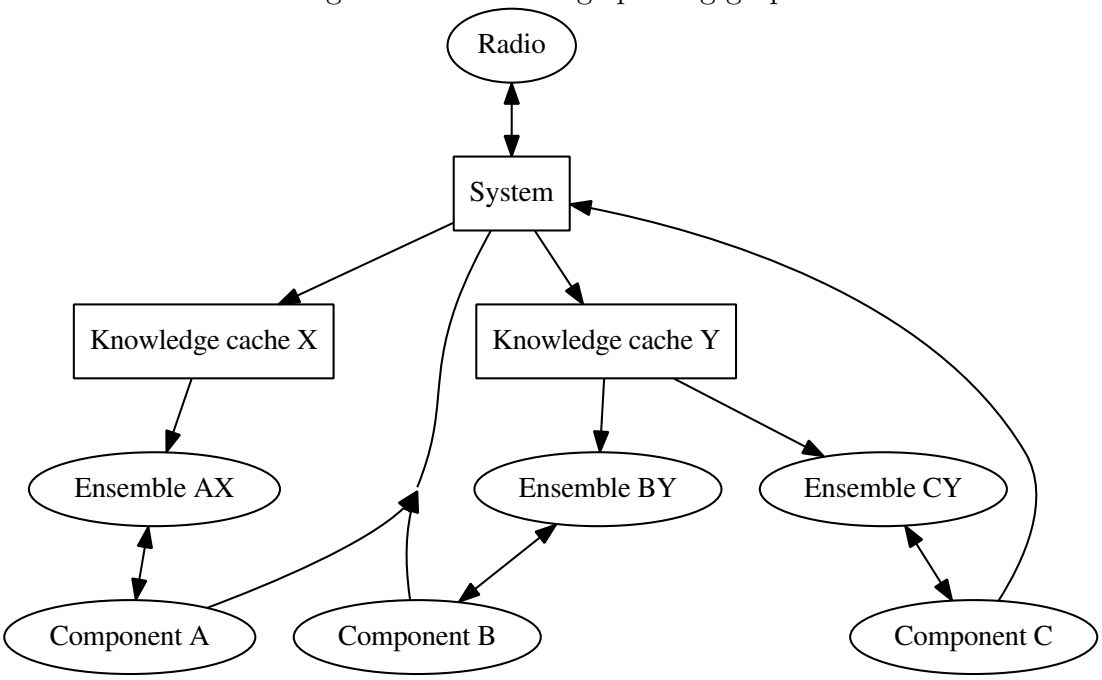
5.3.1 CDEECo++ system structure

Structure of the CDEECo++ application is important for understanding the background of the user defined components and ensembles. The key part of the CDEECo++ application is the system class. Its instance is used as glue between components and radio interface. The system object is created with provided instance of *CDEECO::Radio* implementation which provides the system with communication capabilities. The system object allows components to broadcast their knowledge and processes knowledge fragments received from other nodes.

The system object also handles rebroadcasting of received knowledge fragments. In order to do that the system has internal rebroadcast storage of the size defined by template argument.

The system also has several slots for knowledge caches. The exact number is also defined by system template argument. Knowledge caches are similar to fragment caches, but they do not store knowledge fragments, but instead try to reconstruct complete knowledge data from remote nodes. Each cache reconstructs knowledge of selected type which is together with cache size set by template argument. Knowledge caches are used as remote knowledge sources for ensembles. Thus every node has knowledge caches for knowledge types that are used by local

Figure 5.1: Knowledge passing graph



ensembles. System stores knowledge data into knowledge caches using references to caches which are set using the system's *registerCache* method.

Components are provided with reference to system which they use to broadcast knowledge changes. Ensembles are instantiated with component and knowledge cache references as they use component for knowledge update and knowledge cache as source of remote knowledge.

5.3.2 Implementing radio

One of the things that the framework cannot do for the user is the low level communication. As there are various drivers and communication media the framework cannot come with the complete implementation of driver and communication handling. The CDEECo++ framework deals with packet generation, rebroadcasting and processing of received data using the provided radio implementation.

User defined radio implementation is a class that inherits from *CDEECO::Radio* which is visualized in Listing 5.1. The *CDEECO::Radio* is a simple class that provides radio wrapper. It manages registering a receiver object which will process incoming knowledge fragments. The implementation is expected to call base class *receiveFragment* method when new data is received and implement *broadcastFragment* virtual method in order to allow system broadcasting new data. A simple implementation of the radio using the MRF24J40 driver can be found in the example discussed in Chapter 7.

5.3.3 System and caches

Setting up system object and caches do not require implementation of any classes. Instead the needed objects are created using the templates. As the whole

Listing 5.1: CDEECO::Radio

```
namespace CDEECO {
    class Radio: Broadcaster {
    public:
        void setReceiver(Receiver *receiver);
        virtual void broadcastFragment(const KnowledgeFragment) = 0;

    protected:
        void receiveFragment(const KnowledgeFragment,
            const uint8_t lqi);

    private:
        Receiver *receiver = NULL;
    };
}
```

Listing 5.2: System object instantiation

```
/*
 * Use MrfRadio a CDEECO::Radio implementation taken from
 * the example application.
 */
MrfRadio radio(0, UNIQID, UNIQID);

/*
 * Create system object with 3 slots for knowledge caches
 * and 32 slots in rebroadcast storage.
 */
CDEECO::System<3, 32> system(radio);
```

application should not rely on dynamic allocation, sizes of internal data needs to be set using templates. The system object has to be provided with two sizes. The first one is the maximum number of knowledge caches that can be plugged in. These are used when the knowledge fragment is received by radio in order to store the fragment. The second one is the size of rebroadcast storage. The rebroadcast storage is used to store all received knowledge fragments and rebroadcast them later. In order to do so the rebroadcast storage in the system object runs a FreeRTOS thread. An example of the system object instantiation is shown in the Listing 5.2.

The knowledge caches are also instantiated using a template. In this case the *CDEECO::KnowledgeCache* template is used. The template takes three parameters. The first one is the magic number identifying type of the knowledge the cache should receive. The second one is the knowledge type. And the last one is size of the cache. The instantiation of the cache is illustrated in the Listing 5.3.

The cache needs to be registered with the system in order to receive data. It makes no point to create cache and not to register it. It is advised to create and register all caches before starting the system by running the FreeRTOS scheduler.

Listing 5.3: Cache setup

```
1  /*
2   * Instantiate cache providing magic number which identifies
3   * knowledge/component type, component's knowledge type and
4   * number of slots in the cache.
5   */
6  CDEECO::KnowledgeCache<MAGIC_NUMBER, KNOWLEDGE_TYPE, SIZE> cache;
7
8  /*
9   * Register cache with the system. The cache will not receive
10  * data until it is registered.
11  */
12 system.registerCache(&cache);
```

5.3.4 Implementing component

Component implementation includes definition of component knowledge type, knowledge trait, component class and classes of component processes. As discussed in Chapter 4 the component is just normal class as well as knowledge is a plain structure and processes are also normal classes. Thus the implementation can vary a lot based on the user demands. The following text should be taken as recommendation and source of examples.

Naming conventions

It is recommended to wrap a component related classes in the name-space. Let say a component named submarine is being defined. The component knowledge type may be named *SumbmarineKnowledge* and the component class may be named *SubmarineComponent*. Normally this would not be a problem, but the CDEECo++ framework classes are almost all templates and the knowledge type will be repeated many times. Thus it is wise to create name-space *Submarine* and define knowledge named *Knowledge* and the component named *Component* inside it. Also the processes can be defined inside the name-space to simplify things as shown in the Listing 5.4.

Knowledge definition

The knowledge is just an ordinary C++ structure, but it is handled in special way. The knowledge data are broadcast as contained in the memory. Thus keeping pointers and references in the knowledge makes no sense. Instead all data stored in the knowledge should be direct parts of the knowledge. Thus the knowledge memory region contains all knowledge information and has constant size. Moreover when the knowledge is broadcast it has to be split into a knowledge fragments as it may not fit into a packet. In order to keep the system robust and decentralized the fragments created by single knowledge broadcast are not combined into a knowledge on the receiver. Instead they are considered to be binary patches that can be applied on the old knowledge with the same type and id. This leads to possible inconsistencies as some of the packets may be lost or delayed. This may result in partial update of the old data.

Listing 5.4: Example of a component following naming conventions

```
1  namespace Submarine {
2      // Submarine knowledge
3      struct Knowledge: CDEECO::Knowledge {
4          struct Position {
5              float lat;
6              float lon;
7          } position;
8
9          typedef int Depth;
10         Depth depth;
11     };
12
13     // Position task
14     class Position: public CDEECO::PeriodicTask<Knowledge,
15         Knowledge::Position> {
16         /// ...
17     };
18
19     // Submarine component
20     class Component: public CDEECO::Component<Knowledge> {
21     public:
22         static const CDEECO::Type Type = 0x00000001;
23
24         Position position = Position(*this, this->knowledge.position);
25
26         Component(CDEECO::Broadcaster &sys, const CDEECO::Id id) :
27                 CDEECO::Component<Knowledge>(id, Type, sys) {
28             /// ...
29         }
30     };
31 }
```

Listing 5.5: Example of a knowledge trait

```
1  namespace CDEECO {
2      // Allowed offsets to guarantee knowledge consistency
3      template<>
4      struct KnowledgeTrait<Submarine::Knowledge> {
5          static constexpr std::array<size_t, 1> offsets = { {
6              // Force packet start where position starts
7              offsetof(Submarine::Knowledge, position)
8          } };
9      };
10     constexpr decltype(KnowledgeTrait<Submarine::Knowledge>::offsets)
11         KnowledgeTrait<Submarine::Knowledge>::offsets;
12 }
```

In order to face the inconsistency the user has two options. The first one is to keep knowledge definition simple and handle the inconsistencies manually. The second one is to define a knowledge trait which tells how the knowledge is broken in the fragments. This approach do not solve consistency problems completely, but allows the user to keep small portions of the knowledge which fits into the packet consistent. For instance this can be used to keep position components consistent with each. In other words to be sure that the longitude is consistent with the latitude.

The control on the breaking knowledge into packets is provided by definition of the allowed packet offsets. The user is supposed to define a array which contains allowed offsets in the knowledge where packets can start. When the array is empty then the packet creation is not restricted. On the other side when the array is not empty then the user is responsible for definition of enough offsets to allow broadcasting of the complete knowledge. If the user fails to do so runtime errors may emerge. The definition of the offsets array is performed by specialization of the *CDEECO::KnowledgeTrait* template for the target knowledge type. Unfortunately doing so is quite a lot of code and the specialization must be part of the *CDEECO* name-space. Code listing 5.5 show example of the offsets array definition which forces packets to start at offset position member of the knowledge only. When the template is not specialized a general version of the template is used. It contains empty array which means that the packets can start everywhere.

As the processes and ensembles take many template arguments and the knowledge member types are among them it is wise to typedef knowledge member types in the knowledge structure. Usage of such typedefs is shown in the Listing 5.4. For example when an ensemble which outputs the depth member of the submarine knowledge is defined as an *int* it will be used as one of ensemble's template arguments. Among other arguments it may not be clear what the *int* means. When a *typedef* is used for the depth (as in the example) then the *int* can be replaced by *Submarine::Knowledge::Depth* which tells clearly what it is.

When the knowledge needs to be initialized with certain startup values or the knowledge needs to be filled with zeros at startup then this should be done in the component constructor.

Listing 5.6: Example of a periodic task

```
1  // Position update periodic task
2  class Position:
3          public CDEECO::PeriodicTask<Knowledge, Knowledge::Position> {
4      public:
5          /**
6           * Construct position update periodic task
7           *
8           * @param comp Reference to the component this
9           *  task belongs to.
10          */
11         Position(auto &comp, auto &out) :
12                 PeriodicTask(1259, comp, comp.knowledge.position) {
13             // Some user initialization code
14             gps.init();
15         }
16
17     private:
18         /**
19          * Executed when the period expires
20          *
21          * @param in Constant copy of the component's knowledge
22          * @return Output knowledge. Written to out reference
23          *  specified in the constructor.
24          */
25         Knowledge::Position run(const Knowledge in) {
26             // Return new knowledge position member
27             auto fix = gps.getGPSFix();
28             return fix.position;
29         }
30
31         // Some user defined unmanaged member
32         GPS gps;
33     };
```

Periodic task definition

A periodic task is implemented by inheriting from *CDEECO::PeriodicTask* which is a template. An example of periodic task definition is shown in code listing 5.6. The *PeriodicTask* template has two template arguments. The first one is the component's knowledge type and the second is the output knowledge member type. These arguments are needed in order to provide type safety for the *run* method. The *run* method is virtual in the base class and needs to be implemented by user. The *run* method receives constant copy of the knowledge as parameter and is expected to return output knowledge. When the base class is constructed it needs to be provided with task period, component reference and output knowledge reference. The period will be in most cases provided as constant value. Component reference is reference to the task's component. It is expected to be passed to user defined task constructor when the system is being setup. The output knowledge reference is just reference to knowledge member in the component's knowledge.

Under some conditions it may come handy to create periodic task (or triggered task) which has no output. The *run* method would return *void*. This is possible

Listing 5.7: Example of a triggered task

```
1  // Count position changes
2  class FixCounter: public CDEECO::TriggeredTask<Knowledge,
3          Knowledge::Position, Knowledge::Counter> {
4  public:
5      Critical(auto &component) :
6              TriggeredTask(component.knowledge.position, component,
7                  component.knowledge.counter) {
8      }
9  protected:
10     // Increment counter in position change
11     Knowledge::Counter run(const Knowledge in) {
12         return in.counter + 1;
13     }
14 };
```

by setting *void* as template argument that stands for the output knowledge type. Then the *run* method will be declared as *void* and the base class constructor will not take output knowledge reference as parameter. It may seem that it makes no sense to have tasks with no output, but it may come handy when the task performs hardware control instead of pure knowledge processing. Example of such triggered task is presented in code listing 5.8.

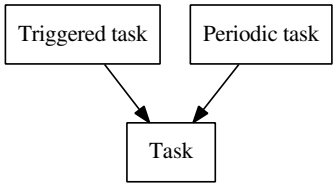
Triggered task definition

Definition of a triggered task is very similar to the periodic task. User defined triggered task implementation inherits from the *CDEECO::TriggeredTask*. The base class is also a template and besides a trigger knowledge the arguments are the same as for the periodic task. The triggered task differs from periodic task by extra template argument that specifies type of the knowledge member used to trigger task execution. Also the triggered task constructor do not require period, but instead a reference to the trigger knowledge member must be provided. The code listing 5.7 shows example of the triggered task. A special version of a triggered task that has no output is shown in code listing 5.8.

Other tasks

The current implementation provides only periodic and triggered task. Moreover the triggered tasks can react only on the knowledge change. The framework itself implement the base task in a separate class *CDEECO::Task*. It is possible to inherit from this class and create customized versions of periodic and triggered tasks or introduce a whole new task concept. Task inheritance is displayed in Figure 5.2.

Figure 5.2: Task inheritance



Listing 5.8: Example of a triggered task returing no output

```
1  // Horn triggered task (note void in TriggeredTask template)
2  class Horn:
3      public CDEECO::TriggeredTask<Knowledge, Knowledge::Position,
4          void> {
5  public:
6      Critical(auto &component) :
7              TriggeredTask(component.knowledge.wreckPosition,
8                  component) {
9      }
10 protected:
11     // Sound horn when wreckage position changes
12     void run(const Knowledge in) {
13         horn.sound();
14         // This has no effect on the knowledge as void is returned
15     }
16 };
```

Task execution

When a task is executed the knowledge access lock is acquired, the knowledge is copied and the lock is released again. Then the task is executed with the copied knowledge as constant input. When task finishes the knowledge access lock is acquired again, the knowledge is updated with the task output and the lock is released. Thus the task should be guaranteed not to work with partially updated knowledge.

Periodic and triggered task base class constructors have two more parameters that have default values set and are not discussed in triggered task nor in periodic task description. These are used to set execution priority and stack size of the task. Default values are set to *FreeRTOS* wrapper default stack size and priority which are priority level 1 and 1024 bytes for stack. Overriding these default constructor values can be used to set different priority or another stack size.

5.3.5 Implementing ensemble

Similar to components and tasks an ensemble is implemented by inheriting from its base class template. In case of an ensemble it is *CDEECO::Ensemble* template. Unfortunately ensemble template arguments are quite many. Ensemble works with two components of different type so it needs to know types of their knowledge. It also has two mapping functions so it needs to have two output types specified. Thus ensemble template has four template arguments. First pair is formed by coordinator knowledge type and coordinator output type. The second pair is formed by member knowledge type and member output type. Similarly to tasks the output type can be specified as *void* which disables the output. This can be used to achieve one way only mapping. As there are many template arguments and the type is frequently used when inheriting from template, it is recommended to *typedef* custom ensemble type.

The ensemble base class as well as the implemented ensemble class has two constructors. One is used on the coordinator node in order to provide mapping from member to coordinator and the other one is used on the member node where

the mapping is from coordinator to member. In both cases four parameters are provided to the base class constructor. The first one is pointer to the component. The second one is pointer to the output member of the component's knowledge. The third one is pointer to the library of the remote knowledge which is periodically scanned for possible knowledge exchange candidates. The last one is the exchange execution period. As well as in case of processes an ensemble is free to store some user defined values in the ensemble class. These will not be affected by the framework.

An example of ensemble is shown in code listing 5.9. Example ensemble maps ids from component to member and vice versa. Thus it demonstrates ability to create dynamic groups/ensembles. All components are members while the ones with lower id are coordinators.

5.3.6 Public API

Beside the classes and methods discussed in the previous parts there are some methods that can extend usage of the framework. Tasks and ensembles are restricted to output only one value. It is possible to contain more output values to substructure and then output that structure. This is how position is returned as structure containing longitude and latitude in the examples. Unfortunately sometimes the knowledge cannot be restructured in this way. In such cases component's methods *lockReadKnoweldge* and especially *lockWriteKnowledge* may be handy. These two are used by the framework when a task is executed. The first one is used to copy the task input and the second is sued to write task output. Using the second one the user can output more values in the task at the cost of losing synchronization between writes. It may happen that the two subsequent calls to *lockWriteKnowledge* will be split by some other call to the same method in another task or ensemble.

The component class also contains methods *getId* and *getType* which may also be handy in some situations.

5.4 Implementation details

This section reveals framework internal structures and principles. It should be consulted when the framework is about to be extended or functionality modified. It also discusses collection of drivers that were included in order to provide hardware access for the example code.

The drivers are implemented as C++ classes. They are designed to be used as global objects due to need of calling their methods from interrupt servicing routines.

5.4.1 Drivers

Beside console driver the drivers are not used by the framework. Their purpose is to accompany example usage of the framework. The drivers reside in the *src/drivers* directory, see the Section 5.2.1 for details of source code structure.

Listing 5.9: Example of an ensemble

```
1  // Typedef base ensemble type in order not to repeat it
2  typedef CDEECO::Ensemble<Coord::Knowledge, Coord::Knowledge::Id,
3              Member::Knowledge, Memeber::Knowledge::Id> EnsembleType;
4
5  // Id exchange ensemble
6  class Ensemble: EnsembleType {
7  public:
8      // Define period as it will be used twice
9      static const auto PERIOD_MS = 2027;
10
11     Ensemble(CDEECO::Component<Coord::Knowledge> &coordinator,
12             auto &library): EnsembleType(&coordinator,
13             &coordinator.knowledge.id, &library, PERIOD_MS) {
14     }
15
16     Ensemble(CDEECO::Component<Member::Knowledge> &member,
17             auto &library): EnsembleType(&member,
18             &member.knowledge.id, &library, PERIOD_MS) {
19     }
20
21 protected:
22     bool isMember(const CDEECO::Id coordId,
23             const Coord::Knowledge coordKnowledge,
24             const CDEECO::Id memeberId,
25             const Member::Knowledge memberKnowledge) {
26         // When coordinator has lower id the membership success
27         return coordId < memberId;
28     }
29
30     // Map member id to coordinator
31     Coord::Knowledge::Id memberToCoordMap(
32             const Coord::Knowledge coord,
33             const CDEECO::Id memberId,
34             const Member::Knowledge memberKnowledge) {
35         return memberId;
36     }
37
38     // Map coordinator Id to member
39     Member::Knowledge::Id coordToMemberMap(
40             const Member::Knowledge member,
41             const CDEECO::Id coordId,
42             const Coord::Knowledge coordKnowledge) {
43         return coordId;
44     }
45 };
```

SHT1x

This driver was implemented from scratch in order to provide sensor access in the example application code. It provides control over *SHT1x* temperature and humidity sensor. The *SHT1x* chip has many features, but this driver aims to provide just basic access. It allows user to read temperature and humidity using the highest precision without CRC control.

GMD1602

The *GMD1602* driver provides interface to 16x2 character alphanumerical display using 4 data lines. It was implemented as practice in the low level hardware control and used in early project stages. Unfortunately the nature of the device and lack of configuration in the driver prevent usage of the device with this driver once the shield with additional hardware is connected to the *STM32F4* board.

Console

Console is not a true driver. In fact it is front-end to the *UART* driver and provides logging of messages to the serial port connected via USB to the PC. The serial link can be used as both great debugging tool and communication channel to reveal DEECo information to other parts of the application. The console is the only driver used directly by the framework. The current implementation of the framework to console binding requires quite a bit of user cooperation when including header files. Fortunately the console usage in the framework is limited just to logging so the console usage can be easily removed from the framework.

StopWatch

The *StopWatch* driver is not intended for general usage. Instead it is designed to be used for execution time measurements at the microsecond level. It has very short maximum measurement period but when used with enabled interrupts, it should detect underlaying timer overruns.

Provided drivers

Various drivers were included in the project that were provided as a base for implementation of this thesis. More about the project can be read in the Section 5.1.4. These drivers include Button driver, GPS driver, LED drivers, MRF24J40 radio driver, Timer and UART driver.

5.4.2 FreeRTOS integration

The FreeRTOS is a C library so it has C language API. In order to simplify usage of the FreeRTOS features in the framework a set of C++ wrapping classes were added to the project. These are placed in the *src/wrappers* directory and include wrappers for FreeRTOS mutex, FreeRTOS semaphore and FreeRTOS task. Wrappers are simple classes that hold the wrapped object handle and provide methods that wrap the related FreeRTOS function calls. Some less

frequent FreeRTOS functions are called directly from the framework without the wrappers.

As the FreeRTOS allocates memory when an task or semaphore is created it needs a memory allocator. FreeRTOS is quite flexible with allocator selection. There is option to use own allocator or to use some implementation provided by FreeRTOS itself. It was decided to use provided allocator that internally uses *malloc* and *free* provided by C runtime. This approach has been tested to work the best of provided implementations. There are concerns about the real-time properties of the system as the *malloc* and *free* calls may block execution in critical sections. Fortunately the whole system can work such way that it allocates only at startup and never call *free*. Thus it will never allocate from fragmented heap.

Features provided by FreeRTOS are used for creation of processes/threads found in components and ensembles as well as for internal tasks inside the framework. Beside task management functions, synchronization primitives provided by FreeRTOS, are heavily used by the framework.

5.4.3 Remote knowledge management

There are two subsystems that handle the remote knowledge in the framework. The first one is responsible for rebroadcasting of received knowledge fragments. The second one is used to store fragments of defined types and combine them into a complete knowledge. Complete knowledge records are used as source of remote component knowledge for ensembles.

Knowledge fragment rebroadcasting

Rebroadcasting of received fragment is performed by *RebroadcastStorage* class. It is a template the only argument of which is the size of the rebroadcast storage. The class contains static array of *RebroadcastRecord* type. Each record contains knowledge fragment data, received time-stamp, scheduled rebroadcast time-stamp and used flag.

When new knowledge fragment is received it is added to the storage with certain probability in order to implement stochastic time-to-live. If there is no free slot in the storage the oldest record is rebroadcast and replaced by new one. When new record is added a rebroadcast time-stamp is calculated based on the received link quality. The calculation is quite simple. Rebroadcast interval is linear function of receiver link quality. Both stochastic time-to-live and rebroadcast interval calculation may need further tunning to adapt real environment with more deployed nodes.

RebroadcastStorage class has internal thread which is used to periodically check storage for records to be rebroadcast. Access to the storage is protected by mutex. Possible delay causes by waiting on the mutex is limited by rebroadcast storage size and generally do not cause problems.

Typed knowledge cache

Unlike rebroadcast cache the knowledge cache is a bit more complicated. The actual storage is implemented by class *KnowledgeCache* which is also template. It takes three template arguments. The first one specifies component type magic

number. The second one is the knowledge type. The last one is size of the cache. Each type of knowledge is handled by custom instance of *KnowledgeCache* class template.

The cache is also formed by fixed array of records. Each record holds: knowledge data, mask (valid regions of the data), time-stamp and complete flag. Each time new knowledge fragment of matching cache type and record id is processed its data are added to the record and the availability mask is updated. When the mask covers whole knowledge than the *complete* flag is set to true. If the cache is full then the oldest record is replaced.

The *KnowledgeCache* class inherits from two helper classes. The first one is the *KnowledgeStorage* class which is an interface for storing fragments in the cache. It is not a template thus its type can be used to store array of caches in the *CDEECO::System* class. Instances of this type can be used to store received fragments. The second one is the *KnowledgeLibrary* template. It has the only template argument which specifies knowledge type. The library can iterate over the complete records in the cache. Thus it allows ensembles to query complete cache records for membership and possible knowledge exchange. The library interface simplifies cache handling as the access to the library is possible without knowing cache size and knowledge magic, but still the library has the knowledge type so it can return properly typed data.

5.4.4 Component

Component is represented by class template *Component* the only template argument of which is the knowledge type. The component class is responsible for knowledge storage, access and running triggered tasks.

Knowledge is public member of component class and the class provides methods to safely read and write knowledge. These are *lockReadKnowledge* which is used to obtain consistent copy of the knowledge and *lockWriteKnowledge* which is used to consistently write part of the knowledge.

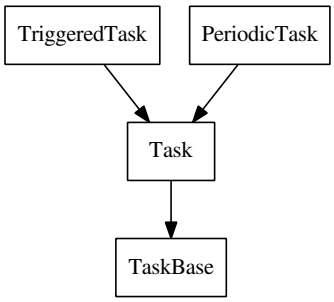
Triggered tasks are added using the component's method *addTriggeredTask*. It stores triggered tasks in the linked list using members provided via *ListedTriggerTask* interface which is implemented by *TriggeredTask* class. The root of the linked list is stored in the *Component* class itself. When the *lockWriteKnowledge* method is executed and new knowledge is different from the old one then the listed tasks are consulted and those affected are executed.

The component also owns a thread which is set to periodically broadcast complete component knowledge. Doing so is important as broadcasting changed parts is not enough. Remote nodes can miss some rare updates of several knowledge areas. Thus their cached knowledge will remain incomplete and unusable.

5.4.5 Processes

A DEECo process is called task in the CDEECo++ context. Task class templates form a hierarchy where the responsibilities are split. The top level templates called *TriggeredTask* and *PeriodicTask* are responsible for scheduling while they inherit ability to execute the task code from base *Task*. The base task itself is composed of the *Task* and its base *TaskBase* in order to reduce code duplication.

Figure 5.3: Task class hierarchy



All the task related classes are templates and take knowledge type and output knowledge type as template arguments. The triggered task also takes a trigger knowledge type as template argument. As the output knowledge type may be *void* and creating references to *void* is not allowed the *Task* template has to be specialized for the *void* output knowledge. The specialized implementation do not contain output knowledge reference, thus it do not write output knowledge and avoids creating references to *void*. The common parts of the specialized and normal implementation were moved to *TaskBase* in order not to duplicate the code.

Constructors of the task related classes take several parameters. The *TriggeredTask* and the *PeriodicTask* are the only instantiated by user. Those take all parameters and pass some of them to base class constructors. The period and trigger knowledge reference are used directly by *PeriodicTask* respective *TriggeredTask*. Component reference and output knowledge reference (if used) are passed to the base classes. In order to allow user not to pass output knowledge reference when the output type is void the top-level classes has two constructors. One passes output knowledge while the other one do not. The output knowledge reference is passed from constructor to base constructor as *auto* type reference which allows the code to be valid even when the output knowledge type is *void*. Doing so requires usage of C++1y features.

The *TaskBase* class which is at the base of the task class hierarchy has the virtual method *run*. The method takes constant copy of the knowledge as parameter and returns output knowledge type. The *run* method is not implemented inside the framework as it is expected to be implemented by the user and contain task code to be executed when the task runs.

Periodic task

Periodic task is quite simple compared to triggered task. Its only parameter is the period. The *PeriodicTask* class also inherits from *FreeRTOSThread* and uses the thread to perform periodic execution of the *execute* method provided by base task.

Triggered task

On the other side the triggered task is more complicated. It also has internal thread which runs the base class *execute* method. The thread lowers the semaphore and execute the task in infinite cycle. The semaphore has initial value of zero. So each rise of the semaphore value causes task to execute. In order to perform check whenever the watched knowledge has been changed the *TriggeredTask* inherits from *ListedTriggerTask*. Listed trigger task members are used to form a linked list of triggered tasks. The list is rooted in the component and check methods are executed on the whole list when the knowledge is changed. As the template function cannot be virtual the checked region is passed in form of two pointers that mark the area in the knowledge structure.

5.4.6 Ensemble

The ensemble design is similar to two combined periodic tasks. Class *Ensemble* is a template which takes two pairs of knowledge and knowledge output types as template arguments. The first pair is used for coordinator and the second one for the member. The user implements virtual ensemble methods in order to provide membership decision method, member to coordinator mapping method and coordinator to member mapping method.

Once the ensemble is implemented it has to be able to be used in two different ways. They can be instantiated on the node where coordinator resides and map from member to coordinator. It also can be instantiated on the node where member resides and thus provide mapping from coordinator to member. In order to accomplish this the ensemble base type has pointers to both member and coordinator knowledge output, but just one pair is used by every instance. The *Ensemble* has two constructors one takes coordinator component and member library, the second one takes member component and coordinator library. Where the *KnowledgeLibrary* is interface to *KnowledgeCache* that provides iterating over remote knowledge of specified type.

Similarly to the tasks the output knowledge type for either coordinator output knowledge or member output knowledge may be defined as *void*. In case of ensembles it makes very good sense to do so as it may be desired to provide just one-way mapping. Unfortunately when the template argument is set to void an illegal code occurs in the ensemble template. This is caused mainly by execution of mapping functions as the code stores return in the variable and the the variable cannot be declared to have *void* type. In order to avoid compilation errors SFINAE feature is used to handle the cases where the error can occur. The resulting template code looks complicated, but the only point is to mask methods that makes no sense when the particular template argument is set to *void*.

In order to run the membership tests and the knowledge exchange an ensemble inherits from the *FreeRTOSTask* thus it contains a thread. It uses periodic scheduling to execute membership tests and possibly run the knowledge exchange.

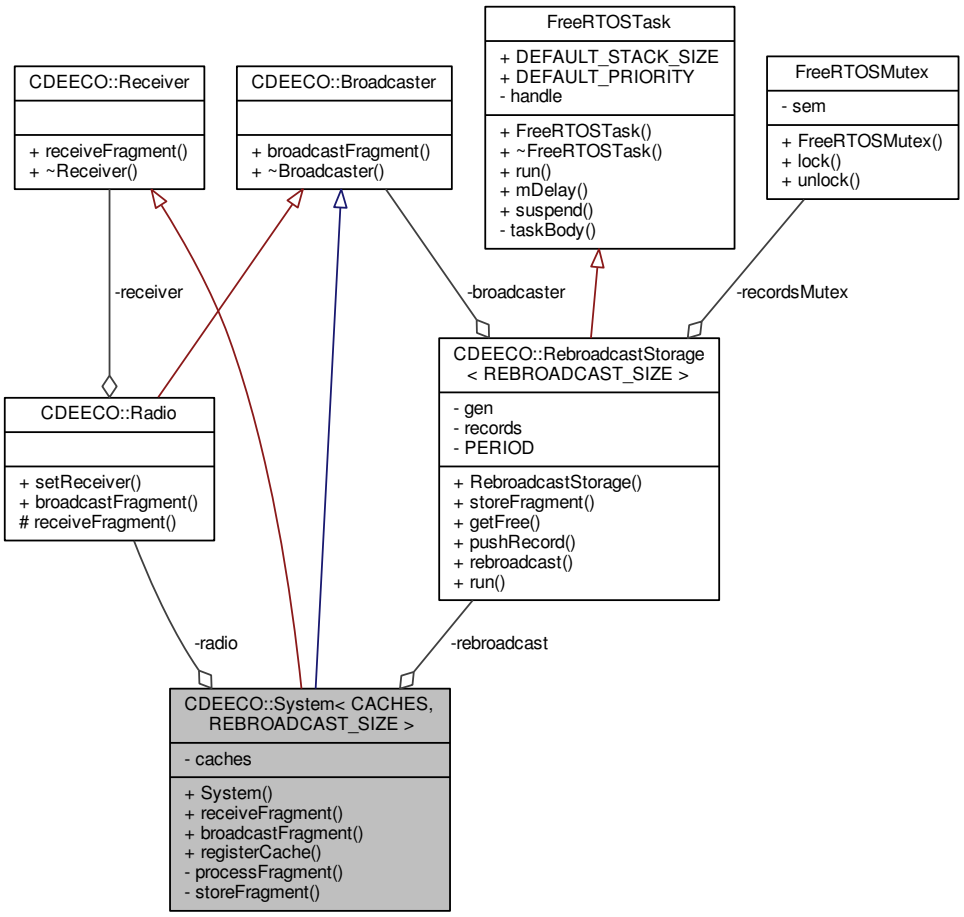
5.4.7 System

The system provides binding between radio and other parts of the system. It is quite simple class template. Template arguments specify maximum number of

Listing 5.10: Selection of member to coordnator mapping method implementation using SFINAE rule

```
1  // Map member -> coordinator, full, COORD_OUT_KNOWLEDGE != void
2  template<typename T>
3  typename std::enable_if<!std::is_void<T>::value, void>::type m2c() {
4      COORD_KNOWLEDGE in = coord->lockReadKnowledge();
5      for(const auto &r : *memberLibrary) {
6          if(r.complete &&
7                  isMember(coord->getId(), in, r.id, r.knowledge)) {
8              COORD_OUT_KNOWLEDGE out = m2cMap(in, r.id, r.knowledge);
9              coord->lockWriteKnowledge(*coordOutKnowledge, out);
10         }
11     }
12 }
13
14 // Map member -> coordinator, dummy, COORD_OUT_KNOWLEDGE == void
15 template<typename T>
16 typename std::enable_if<std::is_void<T>::value, void>::type m2c() {
17     // COORD_OUT_KNOWLEDGE is void, do nothing
18 }
```

Figure 5.4: System class collaboration diagram



caches and size of rebroadcast storage which is also hosted in the system class. The collaboration diagram of the system class is displayed in the Figure 5.4.

The system is just a proxy through which the components broadcast their knowledge fragments. It is also responsible for processing received data. The received data is stored in the rebroadcast cache which is included in the *CDEECO::System* and the received data are also passed to registered knowledge caches. The caches are registered using the *CDEECO::KnowledgeStorage* interface. The interface hides template arguments of the *CDEECO::KnowledgeCache*, thus it simplifies storage of pointer pointing to the registered caches.

As many classes in the system has template arguments which complicate their usage as those needs to be passed to every other class that will use those templates it was decided to implement interfaces which hide those template arguments. As the system is used to broadcast and receive the knowledge fragments it inherits from *Receiver* and *Broadcaster*. These are simple interfaces which take no template arguments and can be used easily without complicated template constructs. Thanks to those interfaces component template do not have to have size of rebroadcast storage as argument.

5.4.8 Portability to different hardware

Embedded hardware differs quite a lot when dealing with different models, brands and kinds of hardware. As the output of this thesis is not a single application but a framework that is expected to be used on wide range of hardware the framework was designed not to rely on particular hardware features. For instance the framework do not use floating point variables, nor it talks to hardware directly. Instead it moves application specific hardware access implementation to the user who has to implement sensor drivers and communication interface. The rest of the hardware specific code, namely the scheduling and synchronization, was implemented using the widely used real-time operating system called FreeRTOS. It was ported to 34 architectures at the time of writing. Thus porting the framework to one of those architectures should be as easy as changing the configuration. Detailed FreeRTOS hardware support can be found at FreeRTOS website[4].

[4] `http://www.freertos.org`

6. Real-time properties of the CDEECo++ Framework

6.1 Memory allocation

One of the concerns with real-time system implementation is memory allocation. Classical heap allocation mechanisms usually do not run in a bounded time. Once the heap gets fragmented it is not clear how long it will take to allocate or free a block of memory. Moreover the typical *malloc* and *free* implementation is not reentrant thus the memory allocation in one task may block another one.

In order to avoid delays caused by memory allocation it was decided to avoid dynamic memory allocation where possible. Thus all the buffers used in framework are fixed size arrays and the linked list of the trigger tasks uses fields contained directly in the task object for keeping references. Unfortunately it was not possible to avoid dynamic allocations in all situations. For instance the FreeRTOS itself relies on memory allocation and the currently used allocation back-end for FreeRTOS uses heap allocation provided by *newlib* runtime. In order to contain those situations the decision was made to allocate memory dynamically, but never to free it. This approach prevents heap from fragmentation, thus the calls to *malloc* are guaranteed to be fast. This approach is possible due to the nature of the target application. As the application first sets up itself and then runs without further memory demands it is not a problem that the memory is never freed.

6.2 Tasks in the system

Every task in the system causes higher load on the scheduler as well as it consumes memory resources. Thus keeping number of tasks in the system low is one of the goals of fast and responsive system. There are two kinds of tasks in the system. Those which are critical to be scheduled on time as their code may be time critical and those which are not so important.

The critical tasks are those:

- Each periodic task has a thread which may need to execute on time.
- Each triggered task has a thread which may need to execute on time.
- Event when not a part of framework, an example radio implementation has two tasks which are time critical as those process data received via interrupt handler.

The following tasks are less sensitive on proper scheduling. Thus can be set to lower priority in case the system is under heavy load and has problems with the tasks from previous group. Those tasks are listed here:

- RebroadcastStorage includes thread which periodically checks for fragments to rebroadcast.

- Each ensemble contains thread that checks for ensemble members and executes knowledge exchange periodically. As the availability of remote knowledge is not deterministic the knowledge exchange is also not reliable by design. Thus the scheduling of knowledge exchange may be considered less important.

- Each component contains thread for periodic rebroadcasting of complete knowledge. As the broadcast fragments may get lost the proper scheduling of the thread is not as important as in some other cases.

6.3 Interrupt servicing

Interrupts have caused many complications during the development of the framework and example application. As the interrupts are used to signal state changes in the hardware, it is crucial not to miss any important interrupt. Thus setting the correct interrupt priorities, enabling only needed interrupts and of course correct implementation of the interrupt servicing routines is mission critical. The configuration of received interrupts, even when it was causing many problems during the development, is not a problem which deserves detailed discussion here. On the other hand setting up interrupt priorities is quite important as it affects responsiveness of the resulting application. The example project used priorities listed in the Figure 6.1.

Priority	Interrupt purpose
-	Highest priority
0	MRF24J40 SPI
-	FreeRTOS critical section
1	MRF24J40 RF
2	System scheduler
3	UART(Serial console)
5	UART(GPS)
7	TIM7(Pulse LED tick)
8	User button
-	Lowest priority

Figure 6.1: Interrupt priorities

As well as interrupt priorities an execution time of the interrupt servicing routines is important. Execution of the servicing routine may cause another interrupts to be missed and time critical tasks to be delayed. Thus the interrupt servicing routines should run in bounded and possibly short time. A long term experiments with the example project implementation revealed that the latest version at last do not miss the interrupts.

6.4 Real word measurements of the live system

This section tries to analyze some real world measured execution times. The data were measured with example application which is described closer in Chapter 7 using the STM32F4 board. In order to perform execution time measurements a driver called *StopWatch* was implemented.

The execution times of various parts of the system seems to be similar to each other in low value variability. Figure 6.2 show execution time values for *MRF24J40* RF interrupt. To no surprise execution times depend on the code path used during the interrupt servicing. as all the lengths and sizes are static

the executions times for different code paths are deterministic, thus execution times for single code path do not vary a lot. As seen on the example in most cases the interrupt is handled very fast as nothing should be done. The other cases seem to be connected to situations when the interrupt marks new packet and broadcast end.

Under some workloads another unexpected behavior is visible. These workloads should have constant execution time, but instead their execution time is formed by several clusters. In most cases execution times are short. In some cases the execution times are higher by certain constant. In the remaining very few cases the execution times are higher by certain higher constant. From the nature of the delay it seems that the task execution was delayed by execution of another task with higher priority.

6.4.1 Time measurement technique

Time measurements were provided by driver implemented by *StopWatch* class. It provides 1 microsecond short period time measurement using one of the hardware timers. The timer is configured to one tick per microsecond. Due to the limited 16 bit hardware timer value may overflow. An interrupt was used to detect timer value overflow and report broken measurement. Unfortunately the overflow detection with interrupt do not work with interrupts disabled. Thus when the code is instrumented with calls to stopwatch the user has to either make sure interrupts are enabled, or guarantee the measured time interval is shorter than 65 milliseconds.

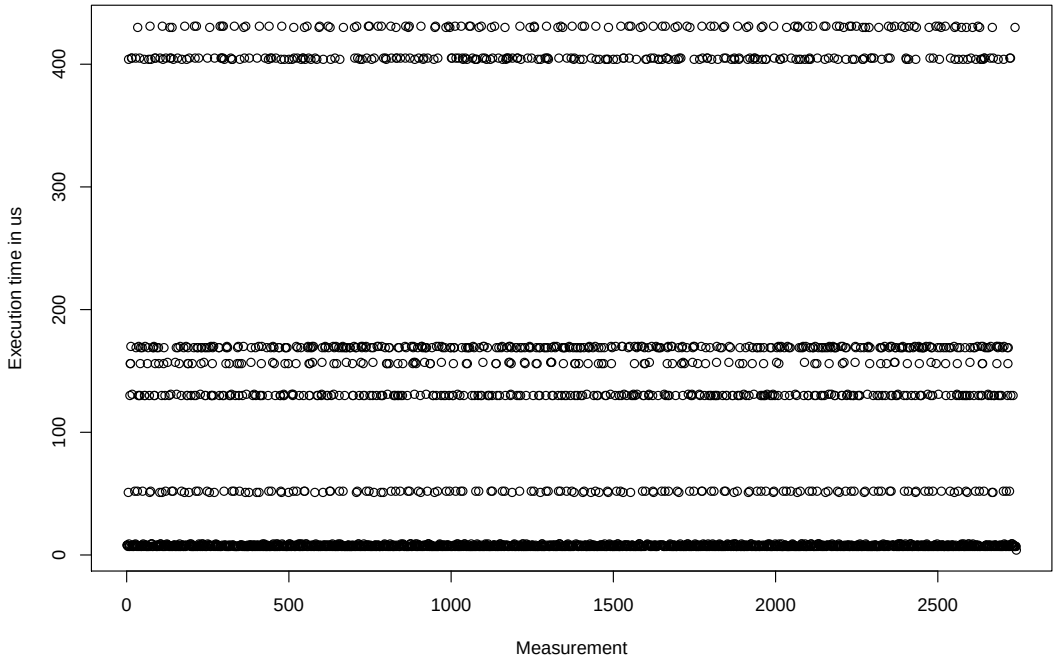
The *StopWatch* class provides static methods for initialization, interval measurements and result printing. The measurement is performed by manually instrumenting the source code of the measured method or code block with calls to *StopWatch::start* and *StopWatch::end*. The time between those calls is then printed to the system's console using the *StopWatch::print*. naturally the calls to the *start* and *stop* have some time overhead and the compiler may reorder their code. Code reordering problem is hopefully minimized by using *volatile* keyword on variables holding timestamps. Time overhead cannot be removed, but as some measurements returned values of 1 or even 0 timer ticks the overhead can be considered minimal.

6.4.2 Execution times of measured code parts

Interrupts

Interrupts are the most execution time sensible parts of code as those usually have high priority and cannot be missed. Unfortunately high demands on interrupt execution time prevented measurement of some of them. Only meaningful data were obtained from execution of GPS UART interrupt and MRF24J40 RF interrupt. These two are thus used as examples. The MRF24J40 RF interrupt is an example of interrupt with longer execution time. On the other side the GPS UART interrupt is very simple and executes within the microsecond range. Execution times of these two interrupts are visualized in the boxplot contained in the Figure 6.3. In order to demonstrate interrupt execution in more detailed view

Figure 6.2: MRF24J40 RF interrupt execution times



a detailed plot of MRF24J40 RF interrupt execution in described in the Figure 6.2.

System tasks

It is hard to demonstrate execution times of different system features as those are usually not contained in bounded part of the code, thus instrumenting their code is not simple. Four tasks that are performed by the system were selected to demonstrate system behavior. Their execution times are described in Figure 6.4. From the graph follows that the trigger check is extremely fast. This is due to only one trigger being set in the example component.

These tasks were selected for the demonstration:

Fragment receive Received fragment processing (including storing into caches).

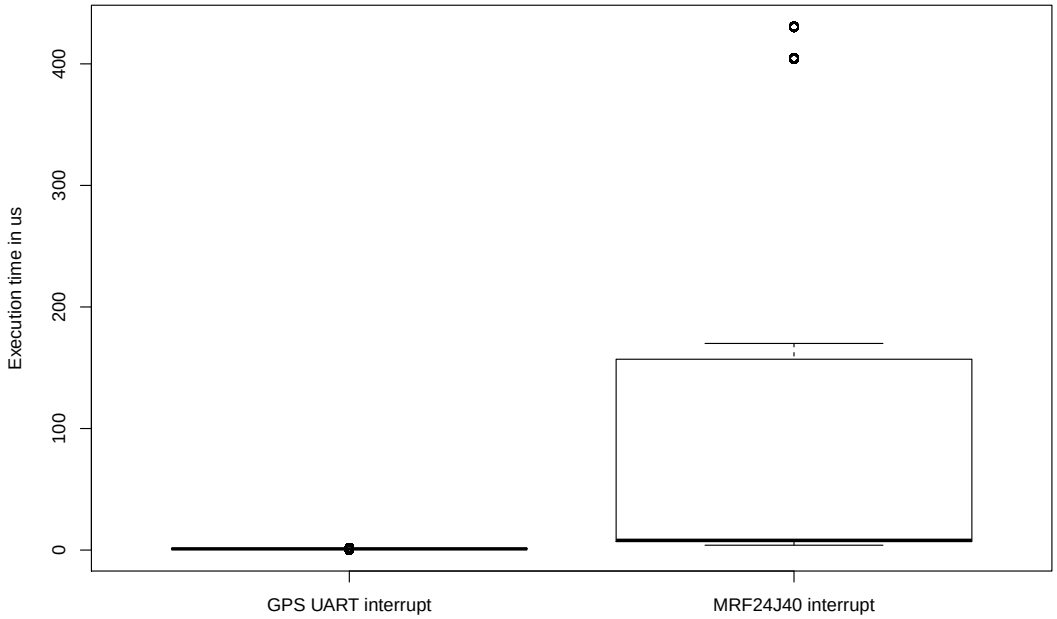
Rebroadcast store Storage of fragment into rebroadcast storage.

Knowledge cache store Storage of fragment into knowledge cache.

Trigger check Check of trigger condition after knowledge update (single trigger task).

Increased number of out-layers in the graph is caused probably by the shortage of cache storage. In such case a record has to be forced out of the cache in order to store a new one. This causes rebroadcasting of the packet in case of rebroadcast

Figure 6.3: Interrupt execution times



cache and finding oldest record in case of the knowledge cache. Both actions cause execution overhead thus enforce higher execution times. Fragment receive contains calls to both knowledge and rebroadcast cache stores thus its out-layers may be caused by those.

Execution times of these system tasks seems to be reasonable as their execution time is lower than 5 milliseconds. The only problems can be caused if some executions would be dramatically slower than those measured. But this is improbable as the measurement of each task contained several thousands of individual measurements.

Beside from pure system tasks also some mixed system/user code executions were measured. These are broadcast of *PortableSensor* knowledge and execution of task that does nothing. Empty task execution is very fast as it contains only reading and writing the knowledge. Change broadcasting takes longer as it contains packet forming and queuing, but still it finished in maximum of 6 milliseconds.

Example user processes

Examples of execution times for user defined code is not important for evaluation of framework properties, but it is important to evaluate possible use cases of the system on tested hardware. Thus some execution times of some nontrivial parts of the example code were measured. These include position task from the *PortableSensor* component and both mappings from the *TemperatureExchange* ensemble.

Both position task and alarm to sensor mapping run quite fast. Position task

Figure 6.4: System tasks execution times

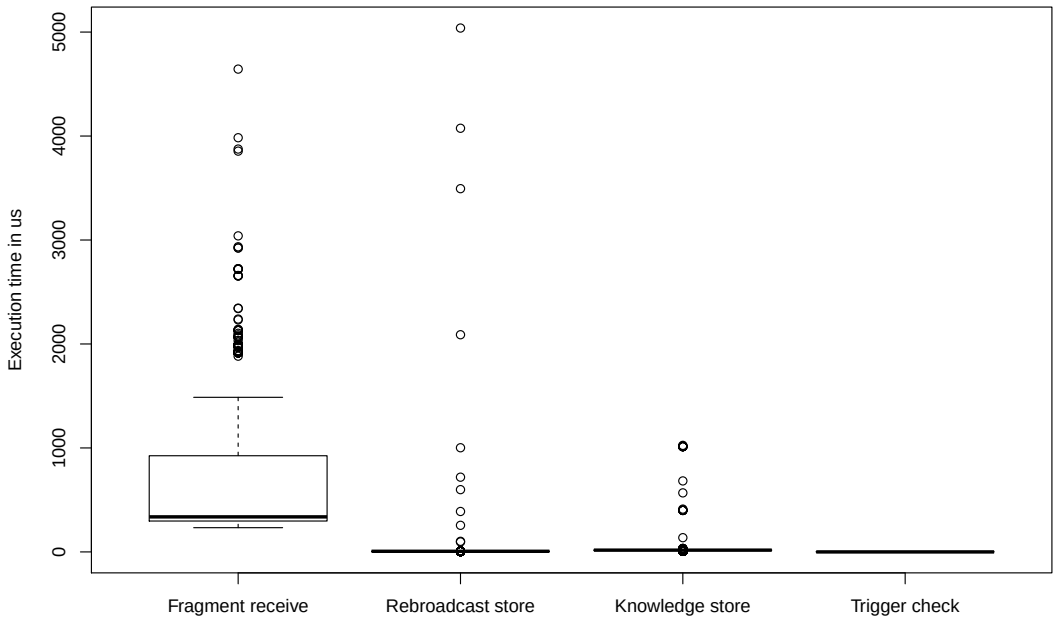


Figure 6.5: Mixed system and user execution times

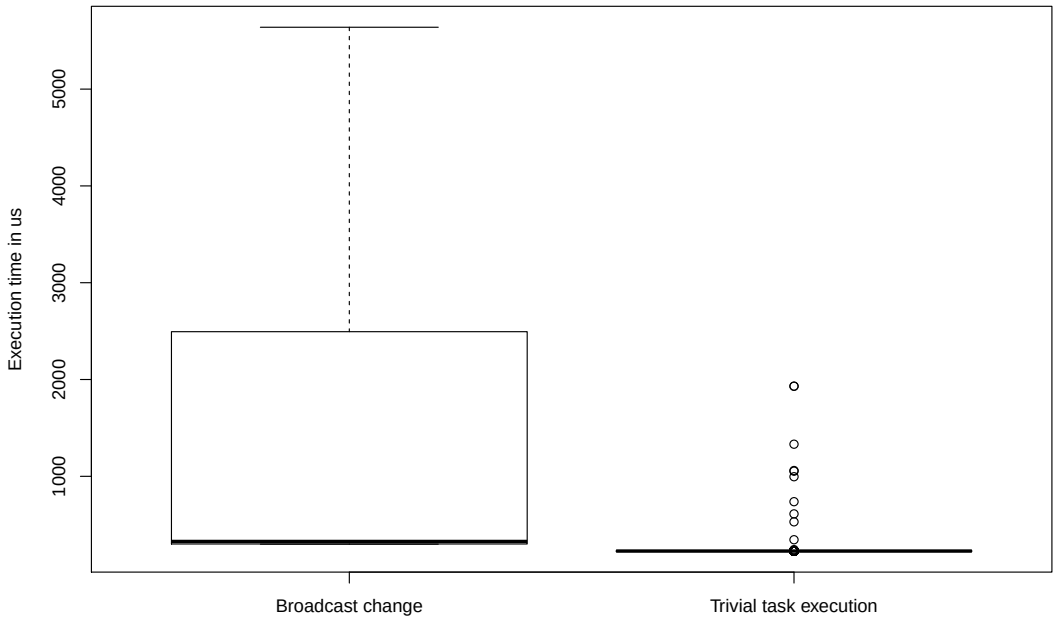
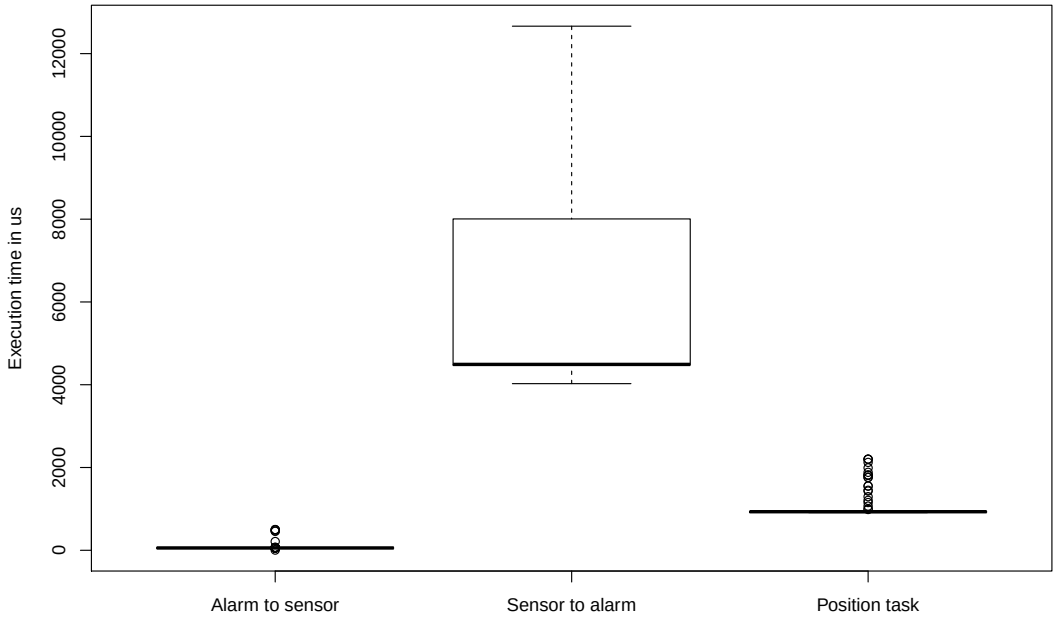


Figure 6.6: User process execution times

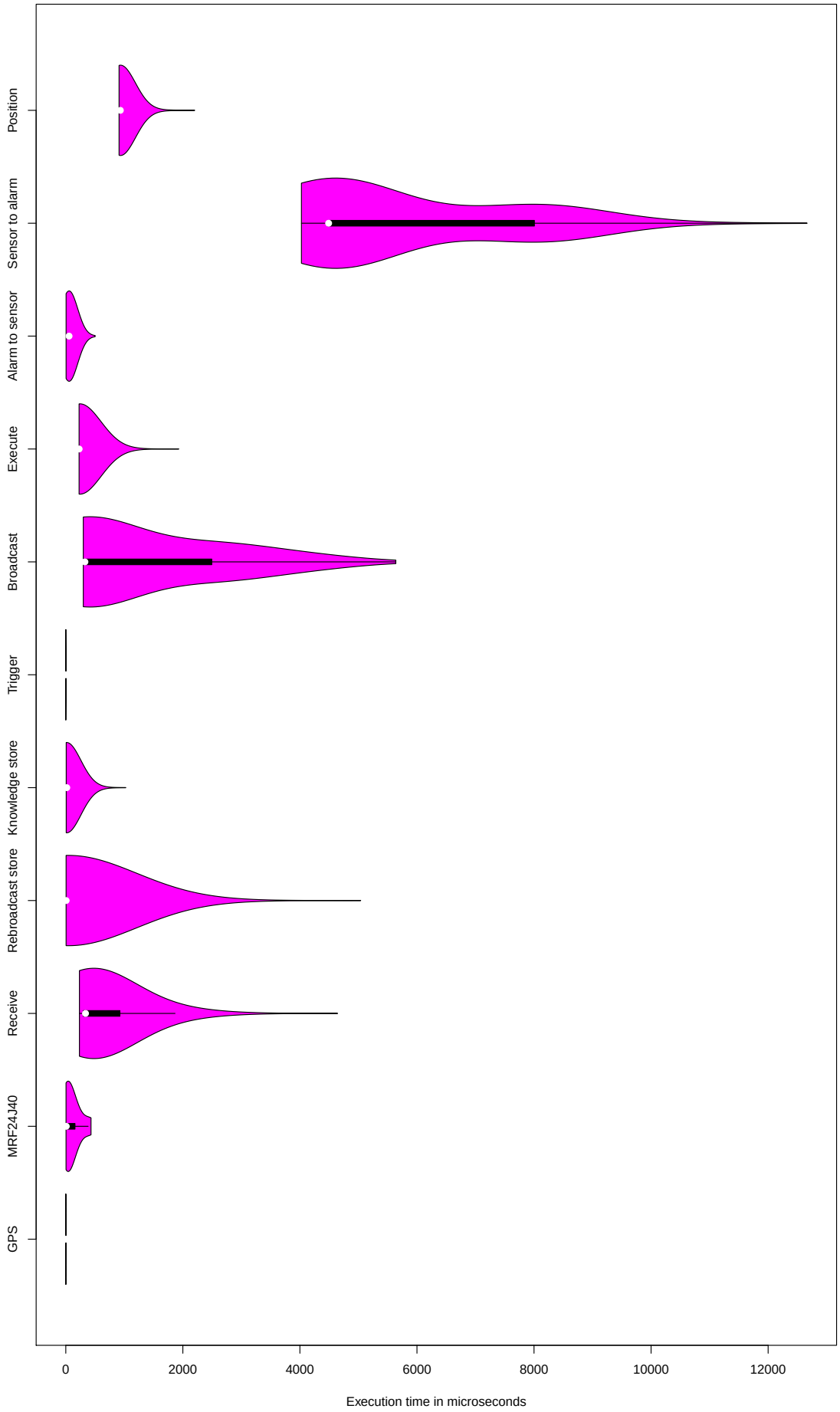


executes in maximum of 2 milliseconds with most executions within 1 millisecond. Alarm to sensor mapping executes in maximum of 1 millisecond without significant differences between executions. Sensor to alarm knowledge exchange is quite different as it surprisingly takes up to 13 milliseconds to execute. This may be caused by increased complexity of the mapping code compared to other tasks.

Overview

For reference a comparison of all measured events in the system a violin plot was included. The resulting graph is displayed in the Figure 6.7.

Figure 6.7: All measured execution times visualized as violin plots



7. Evaluation and use cases

This chapter discusses example or real usage of the CDEECo++ framework. Example application is very simple temperature monitoring system with sensors and alarms. The code listed here is mostly the same as the one used in the example application which is part of the project. Some sources were modified in order to fit document layout. Full sources of the project as well as framework sources are available as attachment to this thesis.

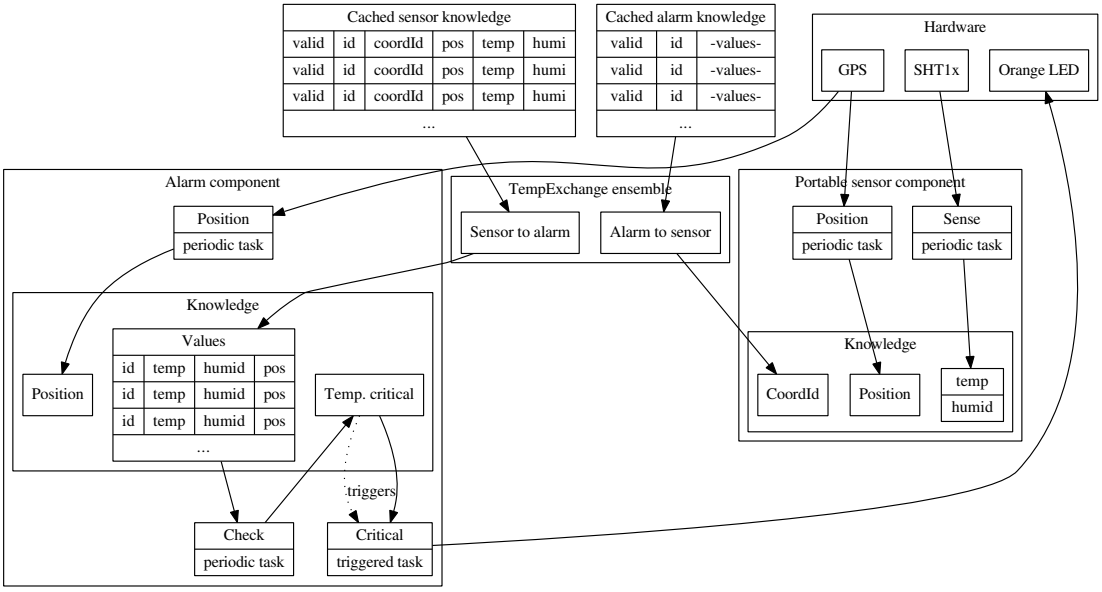
7.1 Example application design

Example application does not intend to do anything usable. Instead, it aims at demonstration of framework capabilities on the simple example of components and ensembles that can be realistically implemented with the given hardware and available sensors. The example application as contained in the project sources, contains several components and ensembles. These are *Alarm*, *PortableSensor*, *TestComponent* and *TempExchange*. The *TestComponent* is not worth mentioning here as it is intended to be used only for very basic framework testing and does not contain any interesting code.

The rest forms a simple application for temperature monitoring. The system contains two type of components: PortableSensors and Alarms. PortableSensors read temperature and humidity values from hardware and store them in the knowledge together with location obtained from GPS. TempExchange ensemble maps sensor values from sensors to nearby alarms. Alarm keeps static array of mapped values and periodically checks them for reaching the threshold. When some value reaches the threshold then temperature critical flag is set in the alarm knowledge and the event is reported using serial console and LED.

The example application was tested and developed on two nodes formed by STM32F4 boards. Each was equipped with SHT1x temperature and humidity sensor, MRF24J40 radio interface, UART GPS and UART to USB serial port interface. As the two nodes are not enough to test distributed framework there is a instance of PortableSensor and Alarm on each node. Also the framework was slightly modified in order to process locally broadcast knowledge. Thus the whole system contains two PortableSensors and two Alarms. Sensor and alarm on the single node can talk to each other via the framework. Also the TempExchange ensemble is hosted on both nodes twice in order to provide mapping in both directions. Thus there are four instances of the TempExchange ensemble in the system. The TempExchange ensemble was also modified in order not to check whether the sensor is near to the alarm, as in laboratory conditions it is hard to obtain good GPS reception. This usage is not natural for target type of the system, but enables at least some demonstration. Temperature monitoring system function is illustrated by data flow graph of a single node. The graph is displayed in Figure 7.1. Arrows in the graph visualize inputs and outputs of the processes and knowledge mapping. The graph is simplified in order to maintain readability.

Figure 7.1: Example system node



7.2 Components and Ensembles

This section displays source code of the components and ensembles taken from example project. The sources were simplified in order to fit the layout of the printed document.

7.2.1 PortableSensor

Portable sensor component reads values of temperature and humidity sensors and stores them in its knowledge. In order to enable creation of ensembles based on geographical location, the portable sensor also reads location from on-board GPS device. Thus the Portable sensor has two tasks. First one called *Sensor* is responsible for reading sensor data. The second one called *Position* is responsible for reading position from GPS device. Both tasks output their data into knowledge of the component.

As displayed in code listing 7.1 and 7.2 the *PortableSensor* class has quite simple knowledge and two periodic tasks. The knowledge contains position, assigned coordinator id and structure containing values. It is initialized in the constructor of the component to zeros using *memset* function. The sense task has a period of 1800 milliseconds, reads sensor values and outputs to the *Value* structure in the knowledge. The position task has a period 1259 of milliseconds, reads GPS position and outputs the *Position* structure in the knowledge.

Component definition is followed by specialization of the *CDEECO::KnowledgeTrait* template. It defines allowed offsets of the knowledge fragments in the knowledge of the component. The definition is included in order to illustrate its usage. It has no effect as the whole knowledge of the portable sensor fits into a single packet.

Listing 7.1: Portable sensor component header

```
1  #include <cdeeco/Component.h>
2  #include <cdeeco/PeriodicTask.h>
3  #include "drivers/SHT1x.h"
4  #include "drivers/GPS.h"
5
6  namespace PortableSensor {
7      struct Knowledge: CDEECO::Knowledge {
8          struct Position {
9              float lat;
10             float lon;
11         } position;
12
13         typedef CDEECO::Id CoordId;
14         static const CoordId NO_COORD = ULONG_MAX;
15         CoordId coordId;
16
17         struct Value {
18             float temperature;
19             float humidity;
20         } value;
21     };
22
23     class Sense:
24             public CDEECO::PeriodicTask<Knowledge, Knowledge::Value> {
25         SHT1x::Properties sensorProps {
26             GPIOB, RCC_AHB1Periph_GPIOB, GPIO_Pin_8, GPIO_Pin_7
27         };
28         SHT1x sensor = SHT1x(sensorProps);
29
30         Knowledge::Value run(const Knowledge in);
31     public:
32             Sense(auto &component);
33     };
34
35     class Position: public CDEECO::PeriodicTask<Knowledge,
36                 Knowledge::Position> {
37         Knowledge::Position run(const Knowledge in);
38     public:
39         Position(auto &component);
40     };
41
42     class Component: public CDEECO::Component<Knowledge> {
43     public:
44         static const CDEECO::Type Type = 0x00000001;
45
46         Sense sense = Sense(*this);
47         Position position = Position(*this);
48
49         Component(CDEECO::Broadcaster &brdcst, const CDEECO::Id id);
50     };
51 }
```

Listing 7.2: Portable sensor component source

```
1  namespace PortableSensor {
2      Sense::Sense(auto &component):
3              PeriodicTask(1800, component,
4                  component.knowledge.value) {
5          sensor.init();
6      }
7
8      Knowledge::Value Sense::run(const Knowledge in) {
9          float temp = sensor.readTemperature();
10         float humi = sensor.readHumidity();
11
12         console.print(TaskInfo, ">> SENSOR PRINTING OMITTED <<");
13
14         return {temp, humi};
15     }
16
17     Position::Position(auto &component) :
18             PeriodicTask(1259, component,
19                 component.knowledge.position) {
20     }
21
22     Knowledge::Position Position::run(const Knowledge in) {
23         GPSL10::GPSFix fix = gps.getGPSFix();
24
25         console.print(TaskInfo, ">> POSITION PRINTING OMITTED <<");
26
27         if(fix.valid)
28             return {fix.latitude, fix.longitude};
29         else
30             return in.position;
31     }
32
33     Component::Component(CDEECO::Broadcaster &broadcaster,
34                 const CDEECO::Id id) :
35             CDEECO::Component<Knowledge>(id, Type, broadcaster) {
36         // Initialize knowledge
37         memset(&knowledge, 0, sizeof(Knowledge));
38     }
39 }
40
41 // Allowed offsets to guarantee knowledge consistency
42 namespace CDEECO {
43     template<>
44     struct KnowledgeTrait<PortableSensor::Knowledge> {
45         static constexpr std::array<size_t, 1> offsets = { {
46             offsetof(PortableSensor::Knowledge, position)
47         } };
48     };
49     constexpr decltype(KnowledgeTrait<TestKnowledge>::offsets)
50         KnowledgeTrait<TestKnowledge>::offsets;
51 }
```

7.2.2 Alarm

Alarm component stores values and positions of nearby portable sensors. The values and positions of sensors are mapped via *TempExchange* ensemble. Own position of the alarm is set by position task which is identical to the one contained in the *PortableSensor* component. Alarm periodically monitors temperatures contained in its knowledge and sets *tempCritical* flag in its knowledge when any value reaches threshold.

Alarm component code is displayed in code listing 7.3 and 7.4. The code defines two periodic and one triggered tasks. The Position task is identical to the one found in the *PortableSensor* component. It is executed once every 1259 milliseconds and stores current location into the knowledge of the component. The remaining two tasks are more interesting. The Check task executes every 3 seconds and checks values, stored into knowledge by *TempExchange* ensemble, for values higher than threshold. When value higher than threshold is found then the *tempCritical* value is set to true and outputted into the knowledge of the component. The change of the *tempCritical* value triggers execution of the *Critical* task. The *Critical* task check whether the trigger value changed to false or true and set orange LED state accordingly.

7.2.3 TempExchange

TempExchange ensemble, even when it is quite big piece of code, does nothing complicated. It just stores sensor values and positions from members into a fixed array stored in the coordinator component. Intended members of the ensemble are the sensors near to the alarm, but for the laboratory testing it was decided that every sensor is a member of the ensemble with every alarm. Code change that implements requirement on proximity is trivial and is included in the comments found in the full version of source codes. In order to demonstrate coordinator to member mapping the ensemble also sets id of the coordinator in the member knowledge. Knowledge mapping tries are performed once in every 2027 milliseconds. Ensemble source code can be seen in the code listing 7.5 and 7.6.

7.3 Example application output

Previous sections explained internal workings of the example application. This section explains outputs of the application. The application controls four LEDs placed in the central part of the STM32F4 board. LEDs provide some basic information about the application state. More information can be obtained from serial output.

Listing 7.3: Alarm component header

```
1  #include <cdeeco/Component.h>
2  #include <cdeeco/PeriodicTask.h>
3  #include cdeeco/TriggeredTask.h>
4
5  #include "test/PortableSensor.h"
6
7  namespace Alarm {
8      struct Knowledge: CDEECO::Knowledge {
9          struct Position {
10             float lat;
11             float lon;
12         } position;
13
14         static const CDEECO::Id NO_MEMBER = ULONG_MAX;
15         struct SensorInfo {
16             CDEECO::Id id;
17             PortableSensor::Knowledge::Value value;
18             PortableSensor::Knowledge::Position position;
19         };
20         typedef std::array<SensorInfo, 10> SensorData;
21         SensorData nearbySensors;
22
23         bool tempCritical;
24     };
25
26     class Check: public CDEECO::PeriodicTask<Knowledge, bool> {
27         bool run(const Knowledge in);
28     public:
29         Check(auto &component);
30     };
31
32     class Critical: public CDEECO::TriggeredTask<Knowledge, bool,
33             void> {
34         void run(const Knowledge in);
35     public:
36         Critical(auto &component);
37     };
38
39     class Position: public CDEECO::PeriodicTask<Knowledge,
40             Knowledge::Position> {
41         Knowledge::Position run(const Knowledge in);
42     public:
43         Position(auto &component);
44     };
45
46     class Component: public CDEECO::Component<Knowledge> {
47     public:
48         static const CDEECO::Type Type = 0x00000002;
49
50         Check check = Check(*this);
51         Critical critical = Critical(*this);
52
53         Component(CDEECO::Broadcaster &brd, const CDEECO::Id id);
54     };
55 }
```

Listing 7.4: Alarm component source

```
#include "Alarm.h"

namespace Alarm {
    Check::Check(auto &cmpnt) :
            PeriodicTask(3000, cmpnt, cmpnt.knowledge.tempCritical) {
    }

    bool Check::run(const Knowledge in) {
        console.print(TaskInfo, "Alarm check task\n");
        for(auto info : in.nearbySensors)
            if(info.id != Knowledge::NO_MEMBER)
                console.print(TaskInfo, ">> VALUES OMITTED <<");
            else
                console.print(TaskInfo, "> No data in this slot\n");

        // Check temperatures for dangerous conditions
        const float threshold = 26.0f;
        for(auto info : in.nearbySensors)
            if(info.value.temperature > threshold)
                return true;
        return false;
    }

    Critical::Critical(auto &component) :
            TriggeredTask(component.knowledge.tempCritical,
                component) {}

    void Critical::run(const Knowledge in) {
        if(in.tempCritical) {
            console.print(TaskInfo, ">> WARNING OMITTED <<");
            orangeLED.on();
        } else {
            orangeLED.off();
        }
    }

    Position::Position(auto &component) :
            PeriodicTask(1259, component,
                component.knowledge.position) {}

    Knowledge::Position Position::run(const Knowledge in) {
        GPSL10::GPSFix fix = gps.getGPSFix();

        console.print(TaskInfo, ">> POSITION OMITTED <<");

        if(fix.valid)
            return {fix.latitude, fix.longitude};
        else
            return in.position;
    }

    Component::Component(CDEECO::Broadcaster &bradcaster,
                const CDEECO::Id id) :
            CDEECO::Component<Knowledge>(id, Type, broadcaster) {
        // Initialize knowledge - zero and set all sensors as unused
        memset(&knowledge, 0, sizeof(Knowledge));
        knowledge.nearbySensors.fill({Knowledge::NO_MEMBER, {0, 0}});
    }
}
```

Listing 7.5: TempExchange ensemble header

```
1  #include <cdeeco/Component.h>
2  #include <cdeeco/Ensemble.h>
3  #include <cdeeco/KnowledgeCache.h>
4
5  #include "Alarm.h"
6  #include "PortableSensor.h"
7
8  namespace TempExchange {
9      typedef CDEECO::Ensemble<Alarm::Knowledge,
10             Alarm::Knowledge::SensorData, PortableSensor::Knowledge,
11             PortableSensor::Knowledge::CoordId> EnsembleType;
12
13     class Ensemble: EnsembleType {
14         std::default_random_engine gen;
15     public:
16         static const auto PERIOD_MS = 2027;
17
18         Ensemble(Component<Alarm::Knowledge> &coord,
19                 KnowledgeLibrary<PortableSensor::Knowledge> &lib);
20
21         Ensemble(Component<PortableSensor::Knowledge> &member,
22                 KnowledgeLibrary<Alarm::Knowledge> &lib);
23     protected:
24         bool isMember(const Id, const Alarm::Knowledge, const Id,
25                 const PortableSensor::Knowledge);
26
27         Alarm::Knowledge::SensorData memberToCoordMap(
28                 const Alarm::Knowledge, const Id,
29                 const PortableSensor::Knowledge);
30
31         PortableSensor::Knowledge::CoordId coordToMemberMap(
32                 const PortableSensor::Knowledge,
33                 const Id, const Alarm::Knowledge);
34     };
35 }
```

Listing 7.6: TempExchange ensemble source

```
#include "TempExchange.h"

namespace TempExchange {
    Ensemble::Ensemble(Component<Alarm::Knowledge> &coor,
            KnowledgeLibrary<PortableSensor::Knowledge> &lib
        ): EnsembleType(&coor, &coor.knowledge.nearbySensors,
            &lib, PERIOD_MS) {
    }

    Ensemble::Ensemble(
            CDEECO::Component<PortableSensor::Knowledge> &mem,
            CDEECO::KnowledgeLibrary<Alarm::Knowledge> &lib
        ): EnsembleType(&mem, &mem.knowledge.coordId,
            &lib, PERIOD_MS) {
    }

    bool Ensemble::isMember(const CDEECO::Id cooId,
            const Alarm::Knowledge cooKnow,
            const CDEECO::Id memeberId,
            const PortableSensor::Knowledge memKnow) {
        // For testing we consider all sensors are members
        return true;
    }

    Alarm::Knowledge::SensorData Ensemble::memberToCoordMap(
            const Alarm::Knowledge coord, const CDEECO::Id memberId,
            const PortableSensor::Knowledge memberKnowledge) {
        auto values = coord.nearbySensors;
        // Try to update record ... OMITTED
        // Try to replace free record ... OMITTED
        // Replace random record
        size_t index = gen() % values.size();
        values[index].id = memberId;
        values[index].value = memberKnowledge.value;
        return values;
    }

    PortableSensor::Knowledge::CoordId Ensemble::coordToMemberMap(
            const PortableSensor::Knowledge member,
            const CDEECO::Id coordId,
            const Alarm::Knowledge coordKnowledge) {
        return coordId;
    }
}
```

Listing 7.7: USB serial port setup

```
#!/bin/sh
serial=/dev/ttyUSB0;
stty -F $serial raw speed 921600 -crtscts cs8 -parenb -cstopb;
cat $serial;
```

7.3.1 LEDs

The following paragraph describes functions associated with the LEDs on the STM32F4 board within the example application.

Green LED flashes when packet is received by radio interface.

Red LED flashes when packet is broadcast by radio interface.

Blue LED blinks under the *TestComponent* task control. Its function is to mark that the system is "alive".

Orange LED is used by *Alarm* component to mark critical temperature is being detected by nearby *PortableSensors*.

7.3.2 Serial output

Serial output provides more detailed information about example application state. Serial output uses USB UART interface found on the STM32F4 discovery shield. The interface acts as USB to serial converter and the example application is sends its text output to it. Serial transmission parameters are listed in the Figure Serial port parameters. Configuration of the serial port can be done on the Linux system using the script displayed in the code listing 7.7.

Figure 7.2: Serial port parameters

Speed	921600
Flow control	software
Parity	none
Stop bits	1
Character size	8

In the default mode the temperature monitoring system node outputs information about sensors and alarms. The default output is displayed in the Listing 7.8. Provided information contains outputs of tasks in the system. Position task prints details on position obtained from GPS device including location which is written to the knowledge. Sensor task prints sensor values and surprisingly also alarm id assigned by *TempExchange* ensemble. Alarm check task prints content of the sensor array. When the state of the *termCritical* value in the alarm knowledge changes to true then a warning message is printed into default output.

When the user (blue) button on the STM32F4 board is pressed the output debug level is changed and new level is printed into output. Supported levels are: *None*, *TaskInfo*, *Info*, *Error* and *Debug*. *None* level is handy for minimizing delays in tasks caused by serial output formating and transmitting. *Info* and *Error* level are used to display various errors and suspicious situations in the framework. And the *TaskInfo* is used to display user defined task output only. *TaskInfo* is the default level.

Listing 7.8: Example application normal serial output sample

```
Sensor task:
> Temp: 26.31 C
> Humi: 56.1%
> AlarmId: 300038

Position task:
> valid:1
> date:6.7.2014
> time:21:46:27
> pos: 50.125672 14.529721

Alarm check task
> Id: 300038   Temp: 26.34 C Humi: 56.1%   Pos: 50.125672 14.529713
> Id: 3f0038   Temp: 26.9 C  Humi: 56.48%  Pos: 50.125660 14.529838
> No data in this slot
> No data in this slot
> No data in this slot
> No data in this slot
> No data in this slot
> No data in this slot
> No data in this slot
> No data in this slot
```

An example of the output with *Debug* level set is contained in the Listing 7.9. Debugging output level is the one with maximum verbosity. It contains output from all other levels as well as its own. As seen in the example it contains detailed information about internal framework processes as well as hex dumps of received and transmitted knowledge fragments. When using the debug level it is important to know that the long and frequent prints disrupt the internal tasks in the system. It may and usually leads to various problems as interrupts may be missed and system tasks delayed.

7.4 Evaluation

As only two nodes were used during the development and testing of the example application, it is hard to predict framework behavior when deployed on the system containing large number of nodes. Anyway, from the testing of the example application it follows that the framework can handle the task of monitoring temperatures and reporting threshold exceeding without major problems. It seems that the combination of STM32F4 board CPU and effective code can handle the task without problems. Moreover as stated in Chapter 6 the execution times of various tasks are quite short. Thus it is possible to execute periodic tasks with radically shorter intervals without overloading the hardware.

The only problem with execution is caused by shortages of RAM. As the framework and the user code contains several threads the memory is mostly occupied by thread stacks. The shortage of RAM space is not a problem for example application as it is quite small. But some more complex application with even more threads, larger knowledge and more cache slots may run out of RAM. The situation can be handled by properly setting stack sizes to lower

Listing 7.9: Example application debug serial output sample

```
>>>> Found complete record, trying membership <<<<
>>>> Record is member of this Ensemble, running member->coord exchange

Broadcasting local knowledge
>>>> Created fragment 0 of the changed knowledge
>>>>>>>>> Sending knowledge fragment:
Fragment:Type:2 Id:300038 Size:70 Offset:8
         0200 0000 3800 3000 7000 0000 0800 0000
         3800 3000 cacc d841 03cd 6242 2a81 4842
         fd76 6841 3800 3f00 52b8 d641 af4c 6342
         bb80 4842 b879 6841 ffff ffff 0000 0000
         0000 0000 0000 0000 0000 0000 ffff ffff
         0000 0000 0000 0000 0000 0000 0000 0000
         ffff ffff 0000 0000 0000 0000 0000 0000
         0000 0000 ffff ffff 0000 0000 0000 0000

>>> Storing fragment in cache
>>>>>>>>> Processing knowledge fragment:
Fragment:Type:1 Id:3f0038 Size:14 Offset:0
         0100 0000 3800 3f00 1400 0000 0000 0000
         bb80 4842 c179 6841 3800 3f00 52b8 d641
         67ac 6342
```

values. Currently the stack sizes are left to default value of 1024 bytes, even when they can be easily overridden. The default value is probably too high for most threads which usually do some trivial job.

The downsides of the framework usage are mostly the complicated component configuration and type system. Sometimes it happens that an incorrect type is mistaken as one of template arguments. In such situations a compiler generates a series of more or less cryptic errors which give very few clues to the true nature of the problem.

8. Related work

This chapter discusses other models and frameworks dealing with the same or similar class of applications as the CDEECo++ framework and DEECo model. Intent of this discussion is to enumerate related work in the field of interest and briefly compare implemented CDEECo++ framework with other approaches to the same or similar problems.

8.1 General embedded component models

When starting discussion about models and frameworks similar to DEECo and CDEECo++ it is wise to start with embedded component models. Some of these models are derived from their non-embedded well-known full-feature counterparts. These are Lightweight Corba Component Model [6] and SOFA 2 High Integrity extension [7],[5]. Both these models base on the their non-embedded versions. They do restrict the standard version with additional requirements that enable embedded and real-time features of the derived models. The restrictions focus mainly on deployment, hardware access and overall variability of the model.

Speaking about the frameworks implementing the models the SOFA 2 implementation is a part of SOFA 2 specification thus the implementation should be available in the same way as the standard SOFA 2 implementation. In contrast to the Lightweight CCM also known as CCM HI which do not have any mainstream implementation. Similarly to standard Corba there are many different implementations. MyCCM-HI[1] may stand for example.

Another sort of embedded component models is formed by models designed specially to serve embedded applications. These models are represented by ProCom [2], Space4U and SCADE [11]. These models are also very similar to mainstream component models, but are tuned directly for embedded usage. The ProCom model aims at splitting the components by their size and real-time processing demands. It splits the components into two layers. The upper layer components are more formal an communicate via messages and the lower layer components form a set of filters and pipes that provides real-time guarantees. The Space4U model and framework were derived from older ROBOCOP model. Its hallmarks are power management and advanced fault management. SCADE model is well known to be used widely in the manufacturing and heavy machinery control. It focuses on the tasks related to control of industry processes. Due to its wide adoption, numerous extensions are found to support different features of embedded component models.

8.2 DEECo based systems

As far as DEECo based systems are concerned, there is only one framework different from CDEECo++ available at the time of writing. It is the reference implementation of the DEECo model called JDEECo. It has DEECo design thus it is very similar to the CDEECo++ in the design and ability of working

[1] `http://sourceforge.net/projects/myccm-hi`

in distributed environment with autonomous nodes. As it is written in Java and uses Java Runtime Environment, it enables neither development of embedded nor real-time applications.

Thus there is no DEECo based framework written in C or C++ that would enable development of embedded applications, or at last application using native code.

8.3 Component real-time systems

As there is no DEECo based real-time implementation it is worth noting some non DEECo real-time embedded component models. When focusing on the models which are primarily designed to handle real-time processing there are: ARINC 653 [10], SaveCCM [4] and RubusCMv3 [3]. It is not a surprise that these models also focus on the safety-critical systems with advanced exception and fault handling. SafeCCM model focuses on limiting flexibility in order to make real-time analysis less demanding. Its area of use ranges from common embedded control applications to vehicle applications. ARNIC 653 aims at providing compatibility layer for many real-time operating systems while using components as reasonable means to do that. RubusCMv3 aims at providing component model with real-time support and the features demanded by industry.

Aside from component models focusing directly on the real-time systems those listed in the Section 8.1 usually have some degree of real-time support as it is needed in wide range of applications.

8.4 Related systems of different design

There are other approaches used to solve similar problems to those solved by DEECo. Despite the design of the systems discussed here is different from DEECo they were included as they also enable development of distributed and decentralized applications.

One of the interesting approaches that deals with distributed modeling is Kevoree [8]. Kevoree project enables development of distributed adaptive component modeled applications while using model at runtime approach. An important fact about Kevoree is that it does not focus only on the cloud and high level computing but also supports embedded Java applications which are similar to those supported by CDEECo++. Compared to CDEECo++ Kevoree does not support either real-time applications or any concept similar to ensembles.

Another approach to deal with the problem are agent based frameworks. These are naturally distributed and dynamic. JADE[2] and JaCaMo[3] are considered well known examples of multi-agent frameworks. Both these frameworks are implemented in Java language, thus do not enable creation of real-time applications. Similarly to Kevoree agent based systems do not provide concept similar to ensemble.

[2] `http://jade.tilab.com`

[3] `http://jacamo.sourceforge.net/`

9. Conclusion

In the previous chapters usage of the DEECo component model for development of embedded real-time applications was discussed. Problems related to the real-time enabled implementation of the DEECo model were discussed and processed into requirements on the implementation. Analysis based on these requirements resulted into the selection of the execution environment, programming language and framework design. In order to illustrate possible usage of the framework an example of the usage was suggested and described in detail.

Once the C++ language was chosen and mapping of DEECo concepts into the C++ was designed to match demands of the real-time embedded applications, designed mapping was presented in examples and described in detail.

With finished mapping specification the focus was moved to the implementation of the framework. The STM32F4 development board was chosen as hardware testing environment. In order to overcome problems with initial deployment of C++ code on the chosen hardware platform a project called *beeclickarm* was used as base for the implementation. The build script and set of drivers for hardware found on the STM32F4 board was ported from this project into new project which contain the framework implementation. Once the deployment of application on the target hardware was solved the development progressed into a phase where the framework code classes were implemented. Even when the code base of the CDEECo++ framework is not large the code complexity is quite high as it was tough to meet requirements on the system while keeping user friendly API, meaningful code structure and specified C++ mapping. Implementation of the framework is available as well as all other sources developed as part of this thesis as attachment to this thesis.

Implementation of the CDEECo++ framework was followed by its description and documentation. The documentation is structured into two main parts where one deals with the facts important for framework users and the other one describes framework internals which are important for possible future extensions and other modifications of the framework.

For testing and usage illustration purposes an example application was developed. On one hand the application is too simple to provide a relevant use case of the framework. On the other hand it is easy to understand and works on the pair of STM32F4 boards equipped with sensors. It would be nice to demonstrate the framework usage with a fleet of submarines as described in running example but the application with temperature monitoring is a sufficient replacement. The example application was also used for testing the framework. Its sources are part of the same project as the framework sources, thus the project compiles into a temperature monitoring application using the framework.

As the real-time features are required from the framework the example application was used as subject of execution time measurements and analysis of the overall usability of the framework. The measured values are presented in the form of graphs and brief analysis of the results. The analysis of the results focuses on the explanation of the measured data and the effect of measured execution times on the usability of the system.

In the end of this thesis frameworks dealing with similar sort of component

models as DEECo and CDEECo++ were briefly discussed. From brief analysis of the research in the area follows that there is no other embedded and real-time component framework with the same features as CDEECo++. On the other hand there are many embedded frameworks which enable development of real-time and fault-critical systems. Thus these frameworks enable development of applications similar, but not identical to those developed with CDEECo++ framework.

References

[1] Etienne Borde, Jan Carlson, Juraj Feljan, Luka Lednicki, Thomas Leveque, Josip Maras, Ana Petricic, and Séverine Sentilles. Pride-an environment for component-based development of distributed real-time embedded systems. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 351–354. IEEE, 2011. URL: `https://bib.irb.hr/datoteka/516964.PRIDE-demo-paper.pdf`.

[2] Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis. *ProCom — the Progress Component Model Reference Manual*, June 2010. URL: `http://www.idt.mdh.se/pride/data/documents/ProComRefManual-v1.1.pdf`.

[3] Kaj Hänninen, Jukka Mäki-Turja, Mikael Sjödin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. Supporting engineering requirements in the rubus component model. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-223/2008-1-SE, February 2008. URL: `http://www.es.mdh.se/publications/578-`.

[4] Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, and Martin Törngren. Saveccm a component model for safety-critical real-time systems. In *Euromicro Conference, Special Session Component Models for Dependable Systems*. IEEE, September 2004. URL: `http://www.es.mdh.se/publications/632-`.

[5] Petr Hošek, Malohlava M. Pop T., Hnětynka P., and Bureš T. Supporting real-time features in a hierarchical component system. Technical Report No. 2010/5, December 2010. URL: `http://d3s.mff.cuni.cz/publications/download/HosekPopMalohlavaHnetynkaBures-Technical2010-12-SOFAHi.pdf`.

[6] OMG. Corba component model. URL: `http://www.omg.org/spec/CCM`.

[7] OW2. Sofa 2 high integrity, 2014. URL: `http://sofa.ow2.org/sofahi`.

[8] Kevoree project. Kevoree. URL: `http://kevoree.org`.

[9] Uwe Rastofer and Frank Bellosa. A component model for distributed embedded real-time systems. Technical Report TR-I4-99-01, September 1999. URL: `http://i30www.ira.uka.de/`.

[10] José Rufino and Joao Craveiro. Robust partitioning and composability in arinc 653 conformant real-time operating systems. In *1st INTERAC Research Network Plenary Workshop, Braga, Portugal*, 2008.

[11] Esterel technologies. Scade systems, 2014. URL: `http://www.esterel-technologies.com/products/scade-system/`.

[12] Information technology for European Advancement, 2005. URL: `http://www.eurekanetwork.org/c/document_library/get_file?uuid=c857ddd8-4625-4ed5-8939-839fb4604aa5&groupId=10137`.

List of Figures

Listings

Glossary

component DEECo component is a unit which contains knowledge and computation processes. 3, 4, 6, 8, 17, 18, 20, 71

ensemble Ensemble is dynamic group of components. The knowledge is exchanged inside the ensemble. 3, 6, 8, 17, 21, 41, 65, 71

knowledge component knowledge is an information produced by components processes and exchanged inside ensemble. 3, 6, 71

knowledge fragment Part of the knowledge contained in the radio packet in form of a binary patch. The fragment contains patch data, type, id, size and offset.. 19, 27–29

reflection Runtime inspection and modification of the application code. 12

Acronyms

CDEECo++ C++ DEECo implementation. 5, 11, 14, 17, 18, 25–27, 29, 40, 52, 64–66

CRC Cyclic Redundancy Check. 37

DEECo Distributed Emergent Ensembles of Components. 3–8, 11, 13, 15, 17, 19, 40, 64–66, 71

FreeRTOS Free Real Time Operating System. 13, 16, 23–25, 28, 34, 37, 38, 41, 43, 44

IP Internet Protocol. 3, 4, 13, 14

IrDA Infrared Data Association. 14

JDEECo Java DEECo implementation. 3, 13, 64

JIT Just-in-time compilation. 12

MANET Mobile Ad Hoc Network. 5, 14

PC Personal Computer. 3, 4, 37

RAM Random Access Memory. 4, 62

SFINAE Substitution Failure Is Not An Error. 41

UART Universal Asynchronous Receiver/Transmitter. 5, 14, 37, 46, 52, 61

Attachments

Attachments are placed on the CD provided with the printed version of this thesis. The CD root directory contains README file with description of the content which include:

1. Framework sources
2. Example application sources
3. Doxygen documentation