A A R H U S   U N I V E R S I T E T

Master thesis

Master in Information Technology
Software Construction

A New Architectural Approach for Context-Aware Self-Adaptable Systems

By

Michael Lykke

15. Juni 2010

______________________________ ______________________________

Michael Lykke, student Jørgen Lindskov Knudsen, supervisor

Abstract

Mobile devices have started to become more and more used in the environment. Here the system must complement user's requirements, changing environments and still response adequately to user's expectations. Therefore performance and scalability are important.

However, the majority of these systems are constructed using a three layered closed architecture with a central feedback loop. This approach limits the performance and the ability to scale in order to support a diverse environment.

This thesis addresses the performance and scalability issues by introducing a new architecture style for constructing context-aware self-adaptable systems. The new architecture is built around the concept of a matrix, and consists of three collaborating parties; services, components and a decision unit. Here the communication between the collaborating parties is controlled by a semi-decentralized feedback loop and not a central feedback loop. The responsibility of managing the view of the environment is divided among the components and the decision unit. Furthermore, the architecture is open and the components can be added and removed dynamically to and from the architecture using a plug-in behavior. By dividing the responsibility among multiple components and introducing an open architecture with a plug-in behavior, this approach provides a better performance and scalability.

Content

1 Motivation

Over the past years, mobile devices started to become more and more used in the environment. Not just for making phone calls, but also to find your way and use services on the Internet. This way of using the mobile device, combined with its sensors, allows the device to be able to support the activities of the user. This is also true, even when the user interact with the device, when it is carried around the environment.

For mobile devices to better complement users' needs and requirements and adapt to the environment, it must support two complementing research fields. One is being context awareness, which allows a mobile device to be able of "knowing when and where a service can happen, what triggers a service provision, how to provide a service to whom and so on." [1, page 1]. The other addresses the challenge of adapting to changes in the environment, in order to respond adequately to users expectations. "Such context-aware software systems automatically adapt their behavior according to context changes." [2, page 1].

In order to fulfill the requirements for such a system, researchers in the two fields has combined the knowledge from context-aware systems and self-* systems and created a new domain of systems called *context-aware self-manageable systems*. The self-manage part covers different disciplines required for this kind of systems to become an autonomous system, such as self-configuration, self-optimization and self- protection [3, page 1].

Research groups have achieved great knowledge in the field of constructing self-managed systems. These systems handle subjects such as monitoring the environment of the system and modelling the context using ontologies. But also the ability to adapt to changes in the environment by constructing the architecture dynamic. These subjects are merged together to a complete system that further provides feedback loops and reasoning about changes.

However, the majority of these systems are constructed using a three layered closed architecture with a central feedback loop. This approach limits the performance and the ability to scale in order to support a diverse environment. The performance and scalability issues are important to be able to adapt to user's needs and requirements, and still response adequately to users expectations.

2 Problem Statement

This thesis addresses the performance and scalability issues by introducing a new architecture style for constructing context-aware self-manageable systems. The new architecture is build around the concept of a matrix, and consists of three collaborating parties; services, components and a decision unit. Here the communication between the collaborating parties is controlled by a semi-decentralized feedback loop and not a central feedback loop. The responsibility of managing the view of the environment is divided among the components and the decision unit. Further the architecture is open and the components can be added and removed dynamically to and from the architecture using a plug-in behavior. By dividing the responsibility among multiple components and introducing an open architecture with a plug-in behavior, this approach provides a better performance and scalability.

2.1 Delimitation

Due to the time limitations of this thesis, it is not possible to construct a complete context-aware self-adaptable system, useful for evaluating the architecture in a live environment. The parts of a context-aware self-adaptable system that is not included in this thesis, is how context modeling and reasoning are used in connection with the new architecture style. This thesis concentrates on the architecture and how the collaborating parties interacts using the semi-decentralized feedback loop, in order to provide a better performance and scalability. A prototype of the architecture is implemented, to be able to evaluate the architecture.

3 Context-Aware Self-Adaptable Systems

The expanded usability of mobile devices, beyond just making phone calls, allows the device to be able to support user activities. In order to meet the needs and requirements, a context-aware self-adaptable system needs to support two research fields:

- Self-management system: Self-management systems originate from the research of robots. Robots may be used for exploring unknown locations/territories/surfaces, such as the surface of Mars. In these situations, robots need to manage their own "life", as they are more or less autonomous systems. Put simply, they need to be self-healing, self-configurable, self-adaptable etc. and in that sense they exhibit a self-manageable behavior.
- Context-aware system: a special field of pervasive computing is concentrating on context-aware systems. These systems provide "the ability of a device or program to sense, react or adapt to its environment of use" [6, page 1]. Though they constantly complement the user and provide facilities for the user to use.

This section gives an overview of the parts constituting a context-aware self-adaptable system. It starts be giving an architectural overview of self-manageable systems by introducing the three layer reference model introduced in [5]. Then follows a description of the context modeling which is used by the system to model the environment. Next, the elements of a feedback loop are explained, which combines the layers of the reference model. This knowledge is then used in section 4 to address some limitations and challenges of the approach reviewed here in section 3.

3.1 Architecture of Self-Manageable Systems

Self-management systems have the property of being able to adjust and / or adapt it-self to the environment. This is necessary to continually support the expectations of the user, if something unexpected happens and thus keep the system running. In this line of business there is often more than one solution to a problem; however, to cope with this task, architectures explained in section 3, builds on the reference model mentioned in [5].

The reference model describes a three layered architecture for self-manageable systems, and as in good design practice, each layer has its own responsibility.

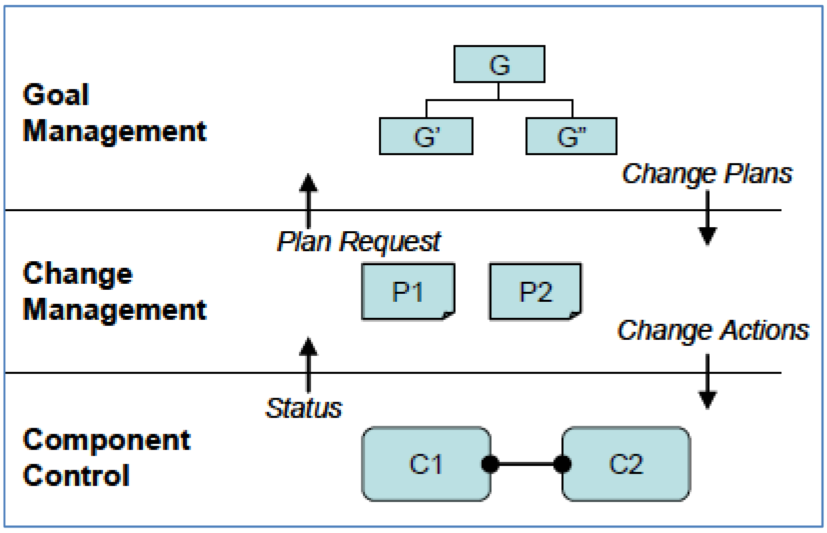


Figure 1: The three layer reference model used for self-manageable systems

The *Component Control* layer consists of a set of components which are interconnected. They try to meet the requirements produced by the change management layer. When the component control layer detects changes in the environment, the changes must be reported to the change management layer. The change management layer then either adjust the current set of components or instruct the component control layer to create new components.

At the middle is the *Change Management* layer, which response to state change notifications from the component layer. It executes plans, which adjusts the current behavior of the system. This adjustment may be expressed either as new operation parameters, to change the behavior of the current set of components. Or it instructs the component control layer to create new components to handle new behavior, if required. If a notification contains changes which cannot be handled by the current set of plans, then this layer most notify the goal management layer. When new goals are added to the system, new plans are introduced into this layer, when necessary.

The topmost layer is the *Goal Management* layer. At times a new situation happens which cannot be handled by the change management layer. When this happens, the goal management layer uses the current state of the system and a specification of high-level goals. It uses this knowledge to attempt to achieve the high-level goals, by making a new plan which is given to the change management layer. Also if new goals are introduced, a new plan are made and submitted to the change management layer.

Figure 2 shows the architecture used in the Hydra project, as described in [1, 3]. It follows the three layering architecture style form the robotic research field mentioned in [5]. At the lower layer the environment is monitored and information is applied to a relevant context ontology[1]. There is a context ontology for each relevant runtime context area, such as the device, hardware, device malfunction, QoS etc. The lower layer triggers the middle layer when there are state changes or service calls. The middle layer then activates the SWRL[2] self-management rules for adaption, monitoring, diagnosis and other aspects of the self-management system. The top layer resolves the conflicts based on QoS regulations or user preference etc. The SWRL self-management rules are located in the "*MasterCopyOfRules/Ontologies*" component of Figure 2.

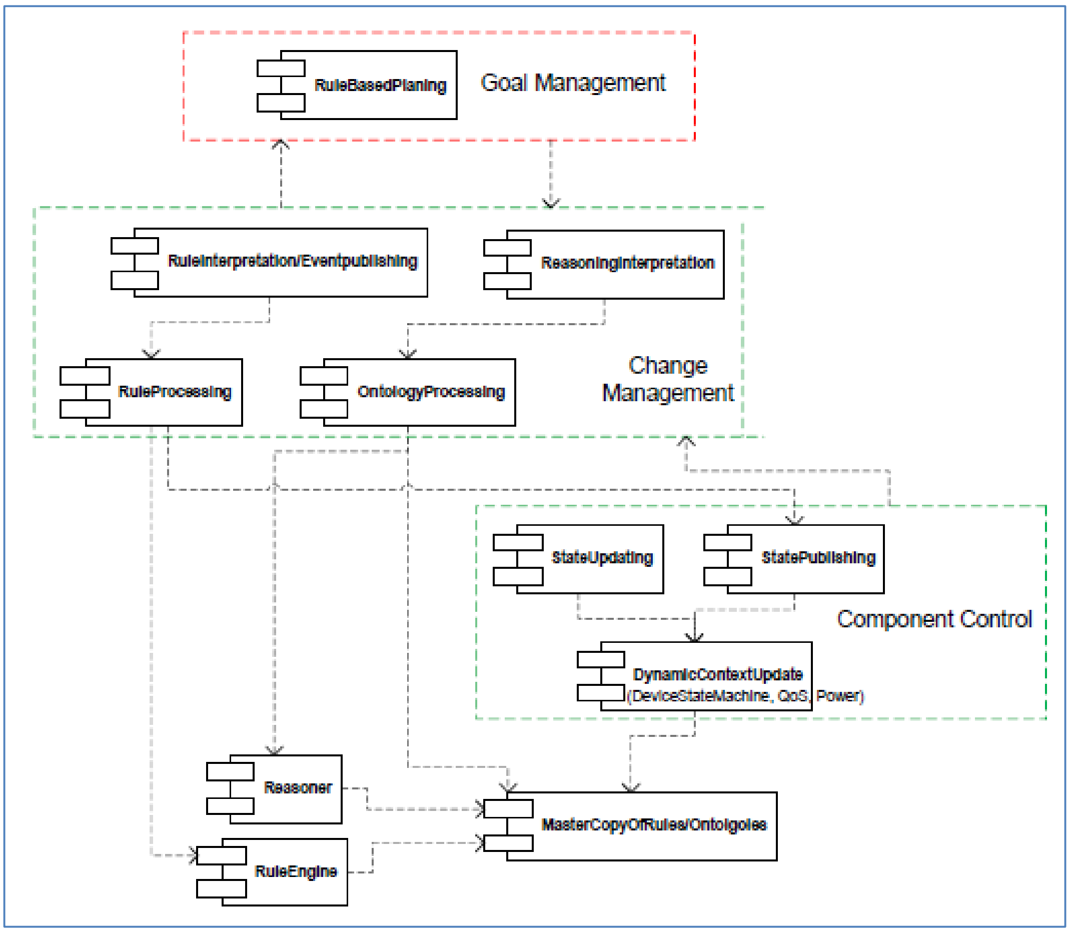


Figure 2: Architecture of the Hydra system

In the MADAM project (Mobility- and ADaptation-enAbling Middleware), the architecture is as shown in Figure 3. When the *context manager*, at the lower layer, monitors the environment, it assigns values to the properties of the components plugged into the framework. Later the *framework architecture model* calculates a set of

[1] An ontology defines the concepts and relationships used to describe and represent an area of knowledge. Basic ontologies published by W3C includes RDF (Resource Description Framework), RDFS (RDF Schema) and OWL (Ontology Web Language).

[2] SWRL stands for Semantic Web Rule Language. "The proposed rules are of the form of an implication between an antecedent (body) and consequent (head). The intended meaning can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold." (http://www.w3.org/Submission/SWRL/#1).

scalar values using the properties of each component plugged into the framework. Each scalar represents a variation of plugged-in components. At the middle layer is the *adaption manager*, which gets change notifications from the *context manager*. It tries to reason about the impact of the changes on the application. Then it selects a component variation, from the *architectural model*, that best suits the current context and users needs. If a new variation is found the *configurator* is notified. The configurator derives a reconfiguration plan that will adapt the system to the new variation. This may require new components to be created at the *context manager* layer. The *core* component provides platform independent services and access to resources such as memory, CPU and network information.

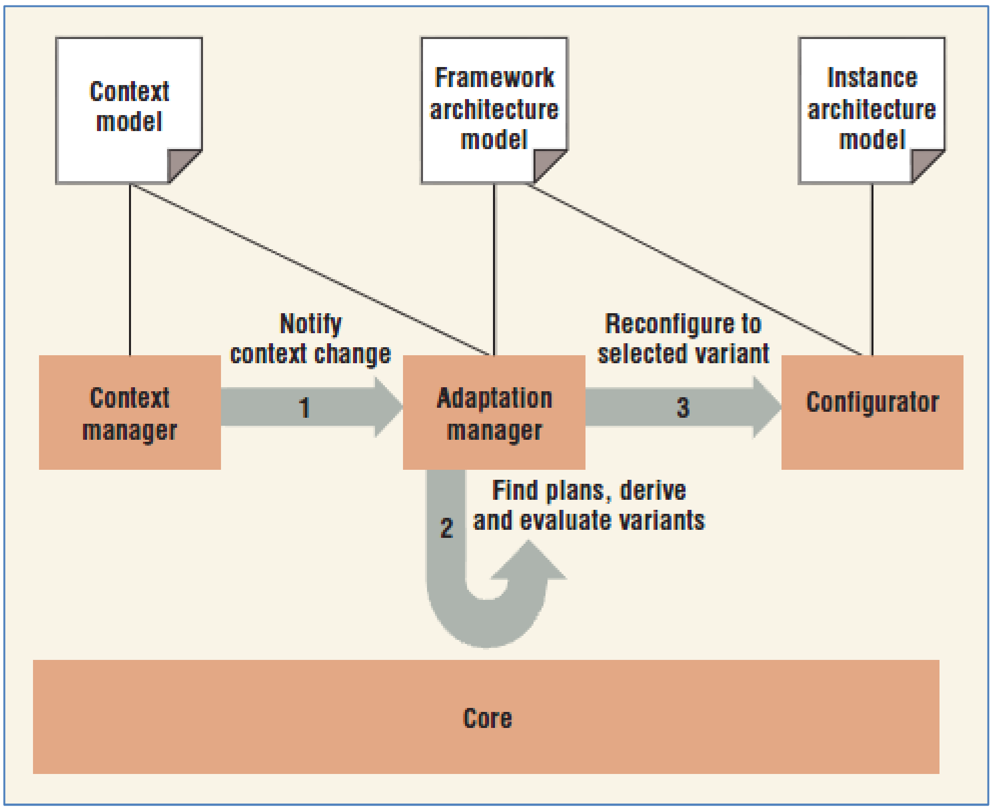

Context model
Framework architecture model
Instance architecture model
Notify context change
Reconfigure to selected variant
Context manager
1
Adaptation manager
3
Configurator
Find plans, derive and evaluate variants
2
Core

Figure 3: Architecture of the MADAM system

One of the main differences between the MADAM and the Hydra project, is how the environments is represented (Object oriented and context ontologies respectively). Also in the MADAM project, the monitored values are applied to properties of a relevant component plugged into the framework, while at the Hydra project values are applied to elements of a relevant context ontology. Both the MADAM and Hydra project are used as examples throughout section 3.

[5] further proposes "an autonomic control loop of actions *collect* (monitoring), *analyze*, *decide* and *act*", which is a result of research work on context awareness, autonomic algorithms and appropriate adoptions techniques [13, page 2]. This control loop has

been named the feedback loop in [4, page 7]. The purpose of the feedback loop is to fulfill the architectural specification and achieve the goals of the system. It does this by gluing together the layers of the reference model, by ensuring that the actions of the feedback loop occur. The feedback loop and its actions are explained in section 3.3.

A further topic relevant for self-manageable systems is how the environment is represented in the system, and how this representation is modeled. This is the responsibility of a context model, which is shortly introduced in section 3.2. The context model may be considered the nerve system of self-manageable systems. It not only transport environment data through the system, using the actions of the feedback loop, but it also structures the environment data in a format understandable by the system.

3.2 Context Modeling

In order for a mobile device to be able to sense and react or adapt to the environment, the mobile device needs a symbolic model of the environment. The context model helps the device to understand the context in which it is being used. It is as such capable of reacting or adapting to the current situation, based on the stimuli provided by the environment.

The term *context* has been defined many times in the research of context-aware systems. Context is important as it identifies any information that can be used to characterize a situation. A situation can include a person, place or object which is relevant for the system, in order to complement and support the user’s activities [7, page 2].

However, in traditional context-aware systems the focus is often on static context, such as location and information about sights. It rarely considers dynamic context to the extent necessary for self-management systems. Dynamic context includes information such as device runtime status (CPU, memory usage, battery level), service call/response relationship and service execution time (availability, reliability, latency). This information is therefore important for self-managed systems, as it is used to handle runtime related requirements [3, page 1].

Many methods of representing context have been used in context-aware systems. They are ranging from set theory, directed graphs, first-order logic to key-value pairs, tagged encoding and object-oriented models [8 page 10, 9 page 5]. However, none of them provide semantic representations, which make it possible to perform reasoning about the context. For this reason, recent research in context modeling has embraced the concept of ontologies (based on OWL), from the field of the Semantic Web. Ontologies add semantic to context information using RDF technology, a feature not achieved by other context modeling approaches.

To take on this approach, the Hydra project has developed a SeMaPS design (Self-Management for Pervasive Services) [1, page 1]. This design is used to model several dynamic context ontologies, relevant for self-manageable systems. Examples of ontologies includes, among others, the Device (models the device and its properties), Malfunction (models knowledge of malfunction and recovery resolutions) and QoS (defines important QoS parameters, such as availability, reliability, latency etc.). In order to reason about context changes, a set of rules has been developed, using the SWRL language, which are used by components of the *Change Management* layer.

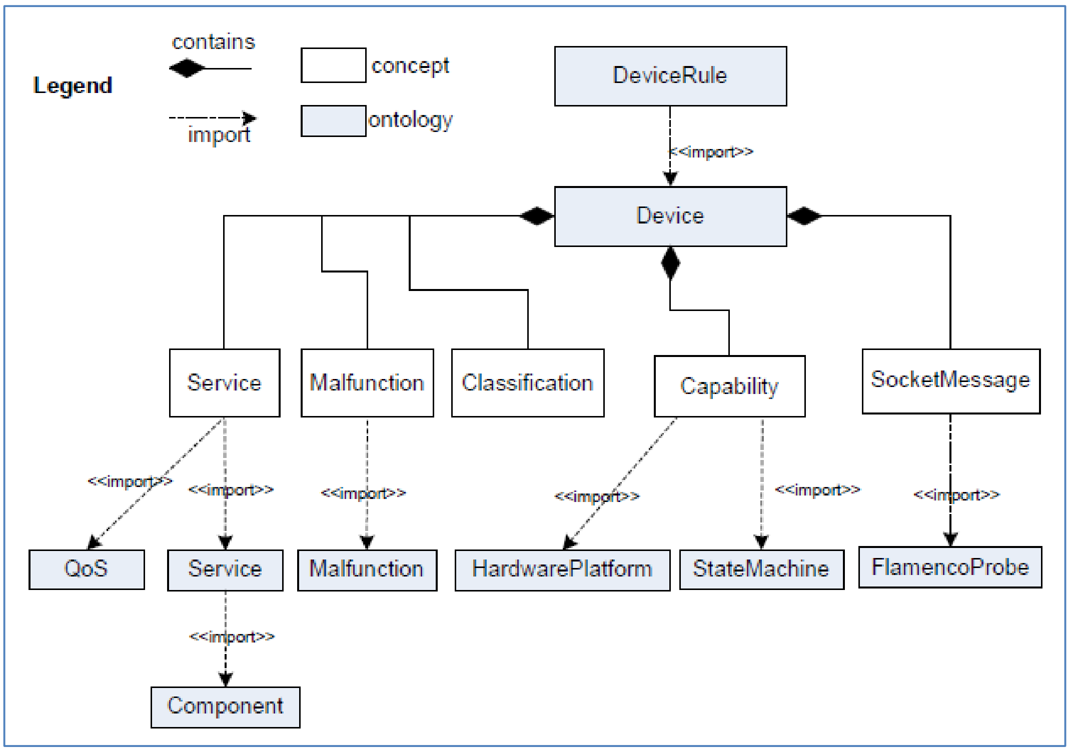


Figure 4: The context modeling approach in the Hydra system uses a structure of SeMaPS ontologies.

The MADAM project uses an object-oriented approach to model the context [13, page 19]. This approach takes an abstract view of the contextual entity that may affect the system and therefore must be included in the process of reasoning. In order to reason about context, each context entity has been associated with four types of meta-data; Time stamp (when was information on the entity updated), Source (a unique id of the source that provided the information), Probability (indication provided by the information source on the trustworthiness of the information) and User rating (a value indicating how important this context entity is for the user). The values of the meta-data may be included in the calculation of the scalar value performed by the utility function.

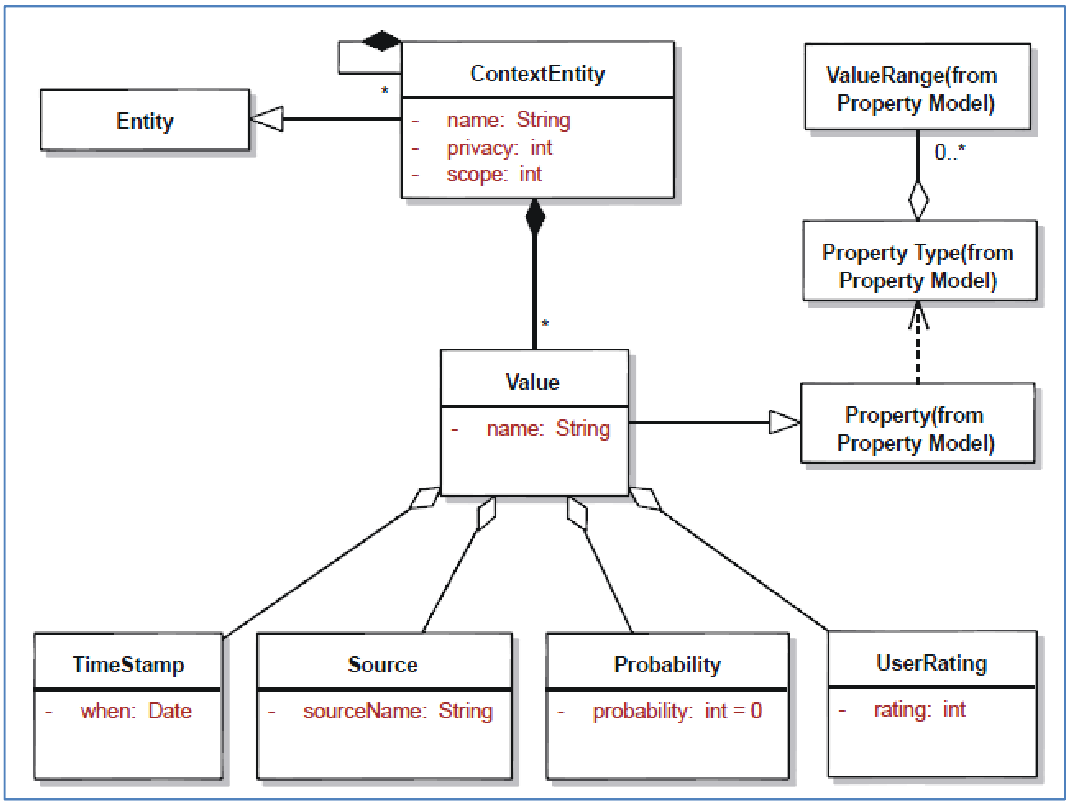


Figure 5: The object-oriented approach to context modeling in the MADAM project.

3.3 Feedback Loop

As stated above, self-management systems originate from the research of robotics. The purpose of self-management systems is to provide autonomousity to software or devices[3]. To cope with this challenge, self-management systems needs to continuously monitor the environment. And further, make some reasoning about the information collected from the environment. This may lead to changes of the behavior to adapt to the circumstances.

This job is performed by feedback loops, which must be considered the heart of a self-manageable system, as it manages the process of:

- Collecting data: a feedback loop cycle starts with collecting data from the environment. Note that explicit user interactions are also considered input and may influence other steps of the feedback loop.
- Analyzing data: as raw data enters the system, they must be analyzed and structured according to models and / or rules.
- Deciding what to do: the decision should consider the current context of the system and may be based on different strategies to fulfill the goals of the system.

[3] Examples of autonomous software and devices are Semantic Web agents and robots, respectively.

- Executing any required actions: when the system has decided what to do, the decision must be executed.

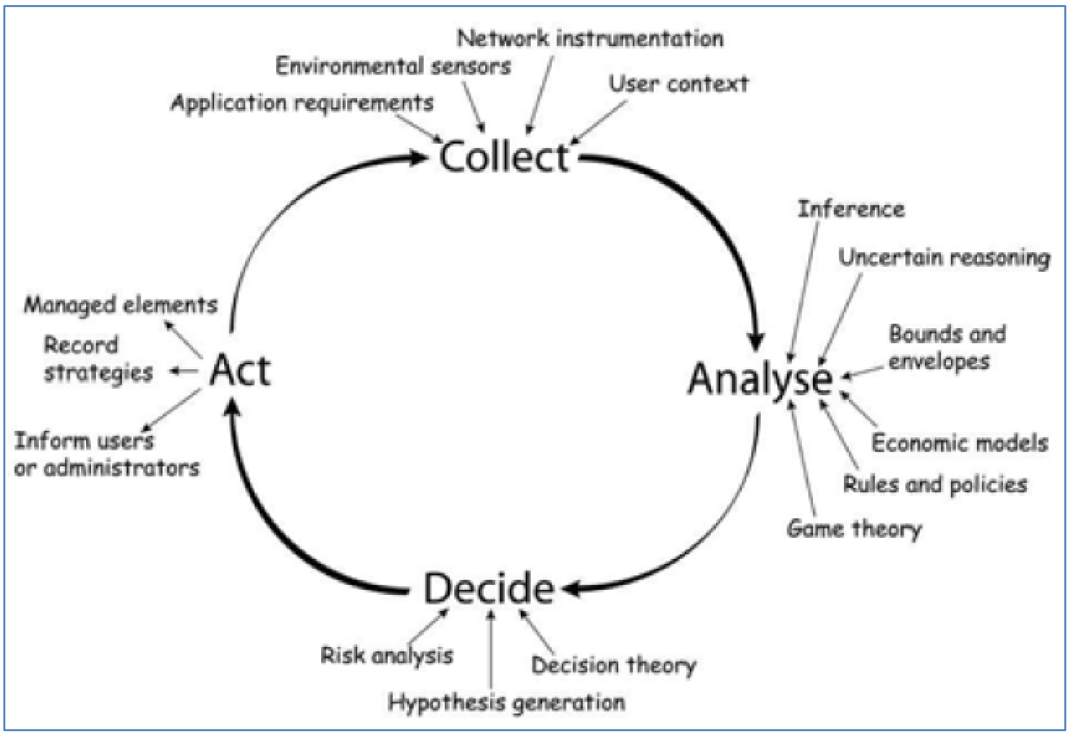


Figure 6: The feedback loop and the actions it consists of.

Each step in the feedback loop may be implemented in its own component. This makes it possible to construct a system more modular, which provides the possibility to use the programming paradigms best suited at each step [11].

Since each step can be implemented in separate components, it also introduces concurrency. Communication between concurrent components may be performed using message passing, event-based communication, remote method invocation etc.

In the Hydra project [3, page 3] the feedback loop is realized using a blackboard pattern and a layered architecture style. Here low level components publish events to the blackboard and high level components subscribes to these events. Hence, the Hydra project uses the event-based communication between components, which is implemented using the publish / subscribe pattern.

However, it is not only appropriate to do concurrent communications between the individual steps in a single feedback loop. It is also possible to introduce concurrent communication between several feedback loops. In this way modularity not only applies at individuals steps in a feedback loop, it may also be desirable for a self-manageable system to use several feedback loops. For instance, the amount of information being monitored in the environment, may be too diverse and thus exert the complexity in a single feedback loop. If this happens, supporting multiple feedback loops may be considered. An example of a situation where two feedback loops helps reduce the complexity, is addressed in [11, page 4] and explained here below.

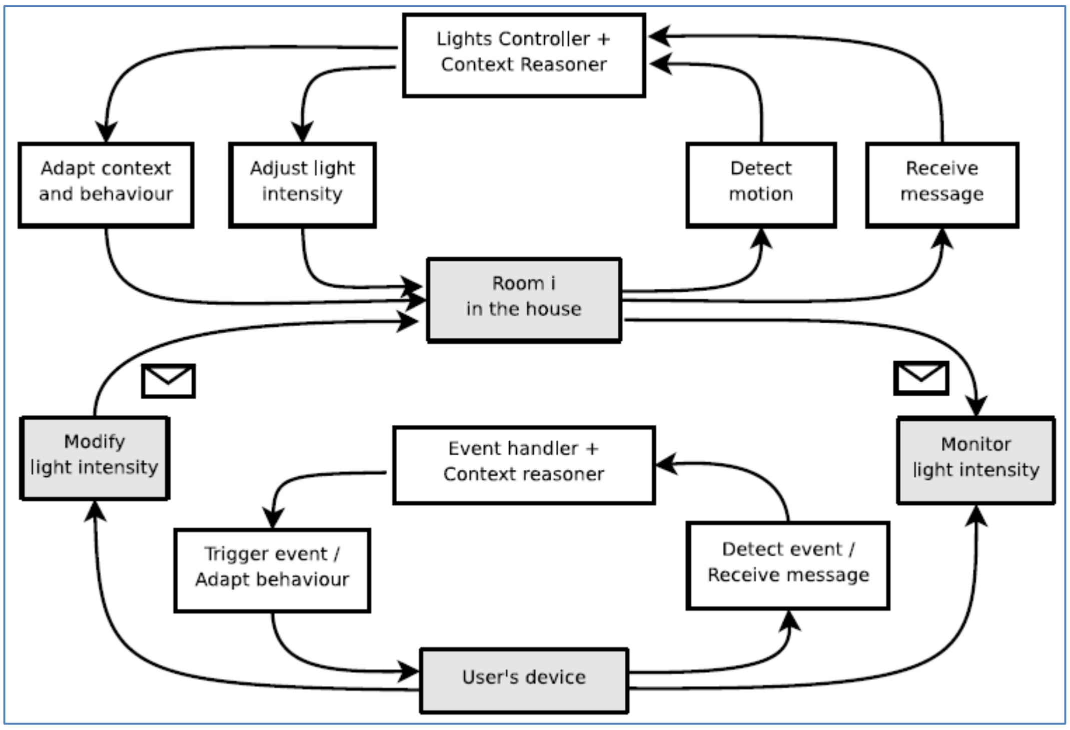

Lights Controller + Context Reasoner
Adapt context and behaviour
Adjust light intensity
Detect motion
Receive message
Room i in the house
Modify light intensity
Monitor light intensity
Event handler + Context reasoner
Trigger event / Adapt behaviour
Detect event / Receive message
User's device

Figure 7: Multiple feedback loops used to reduce the complexity

The feedback loop at the bottom, controls the device and monitors the intensity of the lights while still handling events and receives messages. The device also transmits signals. Then when entering the house, lights are switched on in the rooms the user typically enters. The upper loop handles light in the house. When entering the house and the children are asleep, the light in their rooms are not turned on, because they are in “sleep mode”.

In this way, each feedback loop concentrates only on controlling a few related critical goals, in order to reduce the amount of information monitored from the environment.

The use of a feedback loop then promotes the idea of identifying what information should be monitored in the environment and then dedicates special purpose software to this task. This embraces the concept addressed in [4, page 3], that monitoring everything in the environment may not be possible or desirable. Instead focus on critical high-level goals. The special purpose software could then be used to monitor these critical high-level goals.

Another subject, concerning feedback loops, is if the device controls some predefined contexts. Consider for example, that the device is a mobile phone and it can automatically enter a silent mode when the user is in a meeting. Whenever the user receives an SMS or phone call, it is visualized on the phone with identification of the sender.

3.3.1 Collect - Monitoring of the Environment

In order for at self-manageable system to be able to adapt to the environment, it must monitor the environment. Input from the environment can both come from sensors and the user. How sensor input is accomplished is very individual and depends on both the context information, what kind of sensor that is needed to collect this context information [8, page 4] and the API supported by the vendor of the sensor or device. For instance, if a person is running, is it because that person is in a hurry or just jogging. This properly depends on the cloth / shoes that person is wearing and if cloth / shoes could communicate with the device, the device can then help find the shortest path to the destination. If the person is jogging, the device can be used to capture the track and measure the time it takes to run that track. Further, input from one sensor may activate or deactivate other sensors. For instance, if the device gets input from running shoes, it will activate GPS.

User input, on the other hand, can be considered more reliable as it can be seen as an explicit event. These events happens when the user activates a new task, such as “I’m running” or “Find the shortest path”. Each event may activate a different set of requirements, which may require data from a different set of sensors.

However, there is a limitation that is common to all devices; limited resources such as camera, accelerometer, GPS, Bluetooth, Wi-Fi, etc. which must be shared between all tasks requiring any of these resources.

The challenge is then, to figure out what aspects of the environment that is relevant to fulfill a task – that is, what set of sensor data is relevant to fulfill a given task. Information about these data must be available in the context-model and applied to the model from the environment. What is relevant to fulfill a task may be difficult to forecast, not only by the developers who makes the system, but also for the system itself, if it should truly be a self-adaptable system. A recent trend in context-aware systems, is to make these systems as self-adaptable as possible. That is, with as little interaction of the user as possible. A system that supports this approach, must monitor many aspects of the environment, to try to figure out what the intention of the user is. And it may not have all data available in advance, to make the right decision about what tasks to activate. It can then activate a task or a part of a task, with a smaller set of data. When or if more accurate data are collected, the task can be fine-tuned with more confidence.

3.3.2 Analyze - Detect Context Changes

Topics close related to monitoring the environment, is how to interpret the collected data. The interpretation must consider the context and current state of the device, before it can decide what to do. These steps need to be performed fairly quickly, to provide a good response time to the user. And furthermore, it must also be performed with some degree of certainty, in order to provide the user with the activity expected, based on the given context.

As mentioned previously, monitoring everything in the environment may not be possible or desirable for two main reasons. First, it drains the battery pretty fast and, second, there might not be CPU power enough, while also providing a responsive UI feedback. A possible solution to accomplish this is to identify some critical high-level goals that must be met. These high-level goals are then governed by the system.

Changes in the environment can occur in both small and large increments or even evolve over time. Small changes do not need to have a direct impact on the system at the moment they are discovered by the system. This supports the concept of having a few critical high-level goals, instead of considering all the small changes before they have matured or possibly disappear. However, in order to measure whether a small change has matured to a point where it can have some influence on the system, a historic evolution of the data needs to be present; hence data needs to be recorded.

A current implementation of detecting context changes, used in the MADAM project described in [10], uses a utility function. This function takes some application properties as input and produces a scalar value as output. The application properties are annotations of components and describe the services that a component offers or needs. For instance, "properties can describe that a user interface component supports hands-free operation [and] requires system resources such as memory or network". In this way there is a dependency between the implementation of the component and the context – e.g. if network is available, then hands-free operation can be supported.

Another implementation of detecting context changes is used in the Hydra project [3] which uses the SWRL[4] ontology based rule language. In the Hydra project all variables used in the SWRL expressions only binds to elements in an ontology. A SWRL expression can take advantages of mathematic operations and return either true or false. This makes it possible to construct specific rules for, for example, monitoring battery level and to evaluate a GPS distance.

3.3.3 Decide - Reasoning

Changes in the environment can occur either more or less instantaneous or over a period of time. Either way, when they occur they can have an impact on the system. The system must therefore support some variation points, in order to adapt to these changes. This means that some of the objects being monitored in the environment, have a greater or lesser influence on the variation points in the system. Furthermore, some object may come and go, while others must be monitored constantly. This suggests that some requirements are allowed to vary or evolve at runtime, while other must always be maintained.

[4] The SWRL ontology based rule language "is composed of an antecedent part (body), and a consequent part (head) [where each] consist of positive conjunctions of atoms" [3, page 6]. If all atoms in the body are true, then the head must also be true. Further, the SWRL language supports mathematic operations, such as add, multiply, greater that etc.

In the MADAM project, a variation point could be UI which can implement both hands-free and normal usages. Each of these implementations has some weighted properties, which are used to determine the kind of UI to use in the given context. The weighting resembles the high-level goals. Small changes in the environment may not change the scalar value, in such a degree, that a new architecture variation needs to be deployed. Further, historic data may not need to be recorded, as it is the distance between the current scalar value and the new value that determines if a new architecture variation is needed.

In the approach used in Hydra, SWRL rules can be used to determine the outcome of a variation point. However, since the result of a SWRL expression is only true or false, either a sequence of expressions may be needed or a complex expression, if there are more than two options in a variation point. Because SWRL supports mathematical operations, it is possible to take into account small changes in the environment, so they do not require architectural changes before they become necessary. For instance, GPS is durable if the network bandwidth is greater than 1GB. If the network bandwidth only increases slightly, there is no need for architectural changes, as long as it stays below 1GB. The high-level goals, then needs to be expressed in the mathematic operations, that is, the atoms of the rules.

3.3.4 Act - Dynamic Architectures

When the system has reasoned about the environment and found a solution which fits the requirements, the architecture must be adapted to this solution. When building regular systems, the process of constructing the architecture, is mostly done at compile- or build-time. Different techniques may be applied, such as pre-compiler directives to include or exclude code fragments during compilation, include or exclude components during building etc. To support a self-manageable system, the construction of the architecture must be done at run-time. The techniques mentioned in [4, page 6] for constructing the architecture both complies with regular - and self-manageable systems. However, the challenge of self-manageable systems *is* precisely, to be able to leverage them *at run-time*:

- Architecture-based adaptation: concerned with structural changes at the level of software components. For instance, dynamic loading of components, implementation of interfaces, polymorphism etc.
- Parametric-based adaptation: input parameters to configure software components. For instance, for the system to adapt to the environment, it may adjust the behavior of an existing component, by providing more accurate values at runtime. Or it can use template parameterization to create a new type of component.
- Aspect-oriented based adaptation: changes the source code of a running system via dynamic source-code weaving. Source-code weaving is the process of injecting source code lines into the existing source code, before it is compiled. Doing this at runtime, would require some kind of reflection support of the

programming language, in order to inject additional functionality into existing code.

All these techniques are also used in the Product Line approach [12, page 361]. However, a technique related to the parametric-based adaptation approach, but rarely mentioned in the same context, is the Mixin Layers technique. In this technique, classes have no fixed superclasses and can as such be combined easily to reflect different combinations of context [2, page 1]. This approach is upside down to traditional object-oriented programming, where superclasses are fixed.

Before the new architecture is actually adapted to the system, it may be validated for feasibility. In the MADAM project [10, page 68], this is done by calculating a scalar value for each architecture variant feasible, to the solution and the current architecture. An adaption manager then determines, if one of the variants has a better coverage of the context, than the current architecture. If this is the case, the new architecture is deployed.

When the system has decided to adapt to the environment, this decision may sometime conflict with the expectation the user thought would happen; for instance if the system misinterpret the context. Doing so, the user may need to get involved, to make the final decision. This implies that it may be necessary to allow the user to adjust the degree of autonomics on either all or selected critical goals. Or at least has the power to correct the system. If no user involvement is required at any time, the systems may be considered fully autonomous.

4 Limitations and Challenges

Section 3 explains an often used approach for building context-aware self-adaptable systems. Even though this approach makes good and stable systems, it does have some limitations and challenges, which are described in this section. These limitations and challenges are either directly or indirectly related to the performance and scalability of the system.

Context-aware self-adaptable systems incorporate many aspects of both the context-aware and the self-manageable systems research fields. Therefore, the limitations and challenges described here, are not an exhaustive list, but are based on issues mentioned in section 3. Section 4.1 addresses the problem of dealing with too many non-related information from the environment at the same time. Then, in section 4.2, follows the issue of how the system should support the changes in the environment. And finally section 4.3 addresses the issue of having a centralized feedback loop. For each limitation and challenge addressed, a solution is mentioned which is further elaborated in section 5, where design goals of the matrix architecture style are explained.

To make it easier to explain limitations and challenges, a fictive example, using a GPS application, is used as an illustration.

A user is on foot and using his mobile device to find around town to shop and see the sights. The device is often in a flat horizontal position, so the user can see the screen either in a portrait or landscape orientation. The user can also hold the device in an upright position and use the camera to take pictures, either for his albums or to send to Google to get further information about the content on the image. The use of the camera requires an action by the user, as the system does not know the intentions of the user, when the device suddenly is in an upright position. When the user gets into his car and places the device in a horizontal orientation in a dock, the dock tells the device that it is now positioned in a car. The device now knows, from information from the dock and it is in a horizontal position, that it should now serve as a GPS for driving. It can then make contact with the gas tank and find out how much gas is on the car. If the tank is nearly empty, it can find the nearest gas station and guide the user in the right direction. If there is enough gas on the car, it can ask about the destination and show the path to the user. The system is specialized for GPS for car and on foot. It does not know how to use a camera to find information about a sight, because it does not know how to react, if it is in an upright position. Therefore, it remains a GPS system as it is its main goal. The user must then explicitly start another task, to use the camera.

4.1 A Simple Environment

Before constructing a context-aware self-adaptable system, the important question is; what kind of environmental information must the system be able to manage, to solve the task at hand. The amount of data in the environment is huge and it must be processed fast and with accuracy. For instance clothing or humans may contain RFID tags to identify and provide information about them. A camera is another way for identification and information retrieval, while GPS can be used for location and distance measurement. Voice recognition can activate commands, whereas accelerometer is ideal to identify direction of the device. Not to mention Internet access to maps or Wikipedia, music / video services etc. With all this available, you can make properly whatever system you like, that includes many tasks. Clearly, a system that must be able to complement humans in every situation possible, must use all the resources, services and sensors provided by the device.

However, humans are complicated creatures which are none-deterministic by nature. This sometimes makes it difficult to determine the intention in a given situation, especially for a context-aware self-adaptable system, which are deterministic by nature[5]. As explained in section 3.3.1, for a system to become a truly self-adaptable system, it must monitor many aspects of the environment. The more diverse data the system receives from the environment, the more complicated rules are needed to narrow down the intention of the user. As in the example at the beginning of this section; what is the intention when the user wants to use the camera? This suggests that the system may / should not monitor everything in the environment at all times. Instead it should focus on a smaller subset of the environment, and hence, focus on a smaller subset of critical high level goals.

[4] does not exactly specify if the system should only concentrate on the same critical high-level goals at all times. Ideally – the idea to only monitor parts of the environment, does not mean that the system cannot manage multiple tasks. It only means that the system must manage a few tasks at a time, where each task only has a few high-level goals, to be handled by the system. Therefore, the system should only monitor the aspects of the environment that is required to fulfil the current set of active tasks. Over time, the information received by the system, may then deviate from the overall goals of the current active task(s). If this happens, it is possibly because the user is about to do something else, so other tasks may need to be activated. This implies that the current set of goals is about to change. The system must close down or adjust existing tasks and / or start some new tasks. This is the subject of section 4.2. The new tasks then have other high-level goals which the system must support. How the responsibility of handling goals, may be managed by the system, is explained in section 4.3.

Therefore, by providing non-overlapping tasks which each handles a specific aspect of the environment, the environment gets simpler and easier to manage. If tasks overlap, it gets difficult for the system to estimate which task to activate to monitor a particular aspect of the environment. It is not, however, a limitation to only monitor some of the environment at the time. It is, however, a challenge to identify what is required to monitor, in order to fulfil a task that does not overlap. And further, to estimate when a new task should be started, to complement user activities. In the example, mentioned at the beginning of section 4, the device manages three different tasks; GPS on foot and for

[5] The author of this thesis has not been able to find context-aware or self-manageable systems, which has a non-deterministic behavior.

driving and the use of camera to find information about sights. Each task requires the system to monitor different things, which is latitude and longitude for GPS and images (however, GPS on foot and for driving requires the same input, but the information provided for the user is different). When the device concentrates on GPS, it does not care about input from the camera – it is not even switched on. Further, when the system switches tasks, it is because the user instructs it – it is an explicit action. Making this an implicit action is very difficult, as it requires the device to interpret the situation and intention of the user (it is, however, possible to support simple implicit actions, such as when the network bandwidth falls below a certain threshold, then stop sending or receiving data). And this is where the deterministic and non-deterministic worlds collide. Therefore, keeping the environment simple, means that the system should only monitor what is required to fulfil the current set of active tasks and make support for explicit user actions. The tasks should not overlap, which makes it easier for the system to activate the right task. This approach is better than embracing everything and makes actions implicit. This would require overhead of monitoring too much, while at the same time trying to interpret the users activates, which is difficult (also for humans).

4.2 Supporting Self-Adaptability

As suggested in section 4.1, the system should be able to determine, either by itself or with the help from the user, when a new task should be activated or at least adjust existing tasks. The approach for self-adaptability described in this section, is in many ways similar to the approach used in the MADAM project from section 3. However, due to the time limits of the thesis, a detailed description of a technique, addressing the issue of deciding when a new architecture is required, is not included.

For the architecture to be able to adapt to the environment, two things are required. The first is a decision unit which provides a technique which can acknowledge that a change is required and makes a decision that instructs the architecture to change. Approaches as those used in the MADAM or Hydra projects, from section 3.3.2, can be used. Or one that is similar to hazard analysis[6], which is used in real-time systems. For instance, the current set of active tasks may receive information from the environment that they do not know how to handle. The decision unit then uses a hazard analysis to find a way to start other tasks and maybe deactivate some of the tasks in the current set of tasks, if necessary. The data the decision unit receives from the environment, may not be exact enough to ensure correct identification of the task(s) to activate. It can, however, always fine-tune that decision later as more data becomes available. That way you will be able to say "the system might do this ...", "but it can also do that ... as long as it does this", "the system ought to do this ... but if it cannot, it shall eventually do this ..." which is one of the challenges mentioned in [4, page 3].

What is covered in this section is the second requirement for supporting self-adaptability. This is the adaption of the environment at both the architecture and

[6] Hazard analysis is a technique to analyze data to identify unacceptable risks and then select a way to control or eliminate them [http://en.wikipedia.org/wiki/Hazard_analysis].

component level. A common approach for context-aware services is to react only to context states that are entirely predicted by the developer. Since environment and system context may change, there is the need to identify new contexts dynamically [4, page 11]. This may require the construction of behaviour not known to the device in advance. Even though techniques like source-code weaving exists, it sounds next to impossible to construct a new component from scratch or just append new behaviour to handle specific environment or system aspects. However, in this business you should never say never.

Instead, imagine that a component of the system knows how to handle a specific kind of task, such as GPS on foot or for driving, as in the example above. Each component may use some services, provided by the system, which delivers environmental information to the component. For components using GPS, this would be longitude and altitude. Then if GPS is used on foot, information about sights or shopping is displayed. Or if GPS is used for driving, then gas stations and paths are displayed. The device is still context-aware and self-adaptable. When the user wants to use the camera, the decision unit is notified and locates a component, specialized for this purpose. The component is then plugged into the architecture of system. This component must be known to the system (either located on the device or the system can use a service which knows the location of this component). The output of the camera, the picture, is sent to Google to collect information about the item on the image. This information may then be merged into a map provided by the GPS service.

The architecture then controls the life of the component and provides a framework for the component to operate within. Components can be added and removed from the framework dynamically. And some components may need to be able to communicate with each other. A component further needs to be either platform specific, so they can interact directly with services provided by the system, or they are generic components which use a Hardware Abstraction Layer (HAL) on the device.

	Components (Tasks)		
Services	*GPS on foot*	*GPS for driving*	*Take pictures*
GPS	*	*	
Camera	*		*

Figure 8: Components plugged in into a matrix

The architecture would then consist of components which are plugged in into a matrix and look something like Figure 8. A '*' implies that there is a connection between a task and a service.

The task "GPS on foot" uses both the GPS and the Camera services, but the "Take pictures" task also use the Camera service. When the "GPS on foot" task uses the Camera, the framework connects the Camera to both the "GPS on foot" and "Take pictures" tasks, as they are active tasks (non-active tasks are not included in the matrix). If the "Take pictures" task should use a GPS service, maybe for geo-tagging the pictures, the framework would then connect these two. However, the GPS service for walking and driving may get too complicated to support both tasks, e.g. if the tasks need special environmental information to operate correctly. Should this occur, it can be split into two separated services, which reuses some subcomponents containing common behaviour. This can be done using techniques from section 3.3.4. The technique used by the decision unit for deciding when the architecture has to change, needs to comply with the high-level goals at run-time. And at the same time, any active components must receive information at run-time from the environment. How the responsibility is divided between the component and a decision unit, is explained in section 4.3.

As mentioned at the beginning of this section, this approach is very similar to the MADAM project. Both support components plugged in into a framework, which are able to communicate with each other. The components can further support the techniques mentioned in section 3.3.4. One main difference between the two approaches, which is covered in section 4.3, is the way responsibility is distributed among the components of the system.

4.3 Delegating the Responsibility

Self-manageable systems which comply with the approach, explained in section 3, use a centralized feedback loop. This approach is addressed in [4] as a *weak organization*, as the "adaption is achieved through a global system model, which incorporates a feedback control loop" [4, page 4]. The global system model is the view of the environment which is handled by the decision unit, used in the decision action of the feedback loop. The decision unit is located in the Change management layer of Figure 1. In this organization form, the dynamic architecture is adapted to the environment, based on a set of high-level goals which is guarded, single handed, by the decision unit. To handle the global system model single handed, may cause scalability issues, when more and more goals and tasks are added to the system. At the other extreme, [4] uses the term *strong organization*, where there is no single subsystem managing the global system model. Instead, the components of the architecture, manages the global system model, by changing their structure or behavior to adapt to the changes of the environment. This is done by collaboration between the components. It can be argued that this approach also lead to a scalability issue. For instance, when there are many components collaborating to achieve consensus, before adapting to changes, it requires more communication. However, the two approaches can briefly be summarised as, the strong organization form may be seen as a bottom-up approach, while the weak organization form is a top-down approach for adapting.

Self-manageable systems using a weak organization, does not necessary have scalability issues, but the organization form must comply with the purpose of the self-manageable system. As stated above, handling many goals and tasks may cause scalability issues; therefore a weak organization form is best suited for handling a limited environment. An example is to drive an unmanned vehicle around obstacles to avoid collisions. Sensors "only" need to monitor the surroundings, in the direction of movement and steering the vehicle around obstacles. The kind of sensors used is often similar, which suggest that the data entering the system needs similar treatment. However, a context-aware self-adaptable system, deployed in a diverse environment, requires the system to use different kind of sensors. Hence the data entering the system is diverse and needs different treatment. Therefore, it will most likely introduce scalability issues, using a weak organization form. For instance to support the environmental information mentioned at the beginning of section 4.1, requires different treatment. For example, it needs to understand the position of the device (vertical, horizontal, upright position or flat) in connection with context data and many (different) tasks which may be active at the same time. E.g. GPS for finding paths and friends nearby, camera for reading barcode[7] tags and use this data to find additional information on the Internet while streaming Internet radio or music using a 3G network.

However, supporting a complete strong organization form, does not lead to an acceptable solution either, as communication is slower than computation. For example, because the architecture must dynamically adapt to the diversity of the environment, components of the architecture may come and go. And because knowledge of the global system model is distributed among the components, some knowledge may be lost when components are removed. On the other hand, new components must acquire knowledge, before they can be properly used, and find its place in the architecture. All this collaboration slows the system and causing the adapting process to acquire more resources. Thus the response time to the user deteriorates. A strong organization form is then best suited if the time to adapt is not that critical. The environment may be diverse, but it must not change frequently.

This thesis therefore introduces a semi-decentralized feedback loop. This feedback loop corresponds to an organization form that lies between the weak and the strong. The purpose of the semi-decentralized feedback loop is to split the responsibility between the decision unit and the components of the architecture. In other words, the decision unit delegates some of the responsibility to the components. This approach fits nicely with the matrix architecture style, explained later.

This division makes the decision unit partially responsible for monitoring the system's overall goal. The components then operate within the framework that has been made available by the decision unit. Each component is specialized to support a specific aspect of the environment. It further makes local decisions within this area, when governing the responsibilities provided by the decision unit. In contrast to the strong organization form, components do not directly communicate with each other, to find out how they should be organized. Instead they communicate with the decision unit, when they receive data from the environment which is outside its area of responsibilities. The decision unit can then adjust the component, by fine tuning the responsibilities. Although the components are not self-organisable, the semi-decentralized approach,

[7] The camera both be used for reading 1st, 2nd and 3rd generation barcodes - http://en.wikipedia.org/wiki/Barcode.

does have some in common with another close related research field - Multi-Agent Systems (MAS)[8].

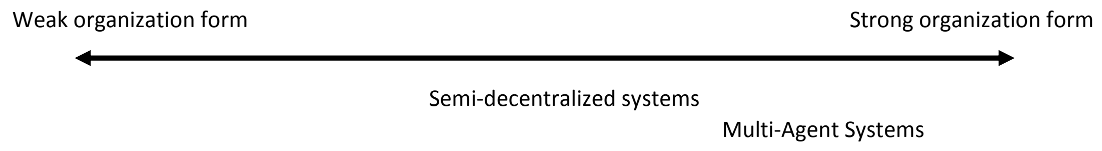


Figure 9: The relationship between different systems

MAS systems consist of “specialized reusable software components, acting on behalf of users, devices, and applications to effectively provide transparent interfaces between different entities in the environment” [14, page 2]. These characteristics are the same as the tasks mentioned in section 4.2. And further, “Agents' decision making is based on local information only, which may lead to suboptimal system behavior” [15, page 11]. The tasks, mentioned in section 4.2, receive necessary information from the environment. Based on this, the tasks make local decisions, which comply with the overall high-level goals of the system. The difference between the semi-decentralized approach and the MAS systems is how the communication is performed. The decentralized control, used in MAS, typically requires more communication and may affect system performance. This is why this approach uses a semi-decentralized control, where the control is divided between the decision unit and the component of the system. However, despite this difference, MAS systems can provide information about (1) the design of the tasks[9] as they resemble agents in MAS and (2) how the architecture is made pluggable to support tasks being added and removed from the architecture.

In this chapter, subjects which support the matrix architecture style for context-aware self-adaptable systems, have been presented; (1) keep the environment simple, only focus on what is necessary, (2) the architecture should be pluggable and components of the system should make local decision and (3) use a semi-decentralized feedback loop to delegate the responsibility among the components, while the governance of high-level goals are kept under central control. These subjects are further elaborated in chapter 5, where the new matrix architecture style will be designed. This architecture will support a context-aware self-adaptable system, deployed in a diverse environment.

[8] Multi-Agent Systems (MAS) is a system composed of multiple intelligent agents which communicates to solve problems which are difficult or impossible to solve for an individual agent or monolithic system (http://en.wikipedia.org/wiki/Multi-agent_system). Agents of a MAS system are characterized by being *autonomous* (or at least partially autonomous), have a *local view* of the global system view and have a *decentralized control* (there is no designated controlling agent).

[9] The term task, from section 4.2, is consistent with the term component used in this section. Therefore, a task corresponds to a component.

5 The Matrix Architecture Style

Section 4.3 argued that neither a complete weak nor strong organization form is useful for context-aware self-adaptable systems, deployed in a diverse environment. In this section the design goals of the matrix architecture style is described, while the architectural description is reviewed in section 6.

The matrix architecture style consists of three elements which supports the concept of the semi-decentralized feedback loop mentioned in section 4.3. It further has some common characteristics with the Multi-Agent Systems (MAS). Therefore, section 5.1 provides an introduction to MAS systems with focus on these common characteristics. Section 5.2 uses the limitation and challenges from section 4, to describe the elements of the matrix architecture style. Finally section 5.3 first explains how the matrix architecture is different from the reference model from Figure 1. And, second, how it benefits from the approach of context-aware self-adaptable system described in section 3. The design that is captured in section 5.2 is further used when implementing the matrix architecture in section 6.

5.1 Multi-Agent Systems – An Introduction

The concept of Multi-Agent Systems (MAS) is a collection of autonomous agents. They interact with each other and the environment, in order to solve complex problems which are unsolvable for any single agent.

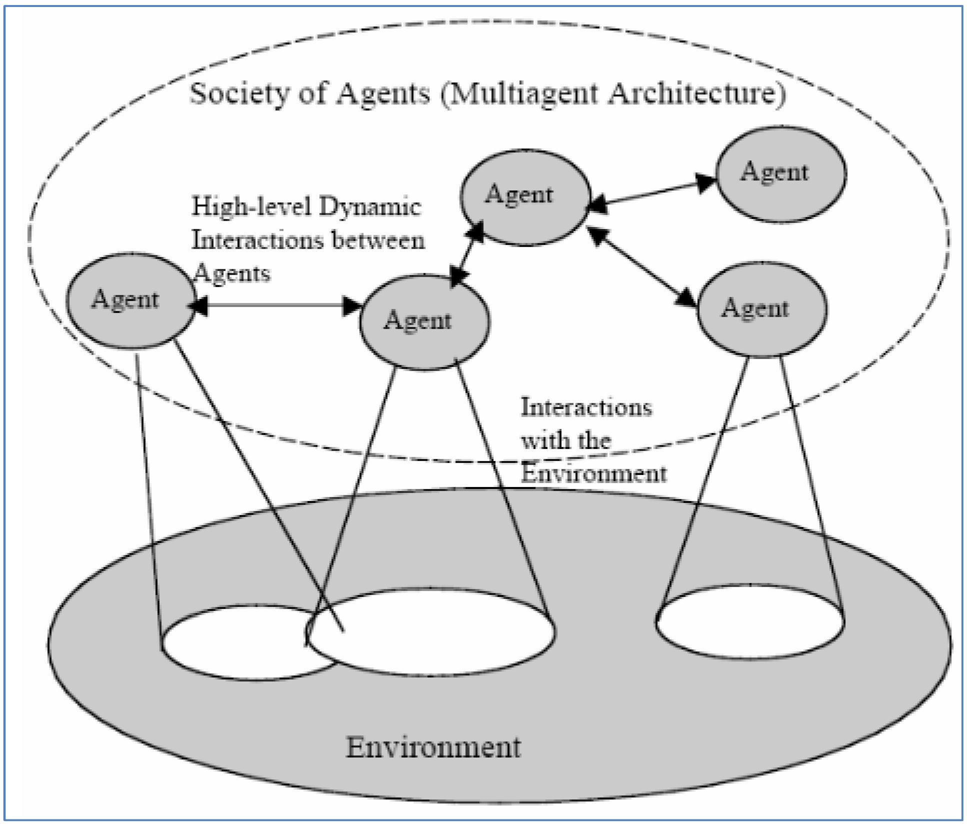

Society of Agents (Multiagent Architecture)
High-level Dynamic Interactions between Agents
Agent
Agent
Agent
Agent
Agent
Interactions with the Environment
Environment

Figure 10: Multi-Agent System

MAS systems can be used to solve problems in the domain of “automated transportation systems, mobile and ad-hoc networks and wireless sensor networks.” [15, page 8]. This means they must be able to support environments which contain conflicting requirements and dynamic operating conditions. To do this, a MAS system is characterized by [16, page 80]:

1. *An agent has incomplete information*: Since the problems are complex, each agent only handles a limited view of the problem domain and therefore does not have the full picture. The agent is then responsible for interacting with a certain part of the environment. This enables an agent to be specialized and designed for a particular area of concern and either supports a reactive or proactive behaviour. This allows an agent to be constructed using the most appropriated paradigm for solving its particular area of concern.
2. *No system global control*: In a MAS system there is not a single agent who has the central control and a global system overview. And because problem solving is done by multiple autonomous agents, it requires the agents to interoperate and coordinate with each other in a peer-to-peer fashion.
3. *Decentralized data*: As each agent covers a particular area of concern of the problem to be solved, the information required to solve the whole problem is distributed among the agents.
4. *Asynchronous computation*: Both because each agent covers a particular area of concern and there is no global system control, some agents may either require other agents to perform operations on their behalf or span another thread of execution to do the job. The other agent or thread will then return the result asynchronously on completion, if no errors happens during executing, otherwise an error or exception occurs.

There are, however a couple of weak points with MAS systems, with regards to context-aware self-adaptable systems. First, the lack of a central control of managing the global system view. And, second, the communication that can occur among agents to solve a problem. Especially asynchronous computation can degrade performance when adapting to changes in environment and as such the response time to the user decreases. However the other characteristics of MAS systems fits nicely with the semi-decentralised approach as presented in section 4.

5.2 Design of the Matrix Architecture Style

This subsection presents the matrix architecture style, which supports the limitations and challenges discussed in section 4. The first topic covered, is how to deal with the scalability issues, discussed in section 4.3. This architecture style requires both a central unit to manage the global system view, but also delegation of responsibilities. This may be achieved using the semi-decentralized feedback loop. The second topic supporting the matrix architecture, is how the system should support the adaption challenge from section 4.2. Here a framework should provide the behavior of adding and removing components dynamically at runtime. Just as agents in MAS systems may come and go.

Finally, the tasks of the system are explained, which not only must interact with the environment as discussed in section 4.1. They should also support the plug-in behavior as suggested in section 4.2 and help managing the global system view as explained in section 4.3. Each task covers a particular area of concern. In this way they resembles agents of a MAS system, except there is no interaction between the tasks when adapting to the environment.

5.2.1 Managing the Global System View

Even though MAS systems require more communication between the agents, each agent does however contain a subset of the global system view. When context in the environment changes agents communicate with each other. This is necessary to figure out how to handle the environmental data, so they can comply with the high-level goals of the system. This capability may be seen as a crosscutting concern, as similar functionality must be applied in every agent across the system. Using Aspect Oriented Software Development[10] handles this concern, as it "extracts" this non-functional requirement into a separate unit (let's call it a decision unit, just to give it a more descriptive name) and makes the system more modularized and maintainable.

Now that the global system view is handled by a separate decision unit, it looks just like the approach explained in section 3. In order for this to take on a semi-decentralized approach, the challenge is then to provide the right split of responsibility between the decision unit and the tasks. Even though this topic is not covered in the thesis, one thing is certain, the decision unit must handle high-level goals while the tasks must handle low-level goals.

This means that the goals must be divided into high-level and low-level goals, to support the semi-decentralized approach:

- High-level goals: These goals are used by the decision unit to decide when a new task is required. Actions from either the user or the environment can influence the decision unit. Actions are subdivided into two categories:
    - Explicit actions: Actions which are activated by the user. For instance, when the user activates a new application. This application may use a new set or subset of the current set of tasks to fulfill its requirements.
    - Implicit actions: Actions which are activated by other tasks. A task receives information from the environment. When this information begins to deviate from the information that is required by the tasks to function normally, it must inform the decision unit about it (the decision

[10] In traditional software development, systems are decomposed into units supporting the functionality of the system. Aspect Oriented Software Development (AOSD) complements the traditional software development, by focusing on the non-functional aspects of the system (http://en.wikipedia.org/wiki/Aspect-oriented_software_development). E.g. in systems where classes resembles tables in a database, these classes have functionality to support CRUD, using the traditional approach. Using AOSD, CRUD is a non-functional aspect of the system, which is implemented in a component of its own, and not distributed to several different classes.

unit can then activate another task or adjust the current task). For instance, the user has activated a task which streams music from the local Wi-Fi network inside the house. When the user later leaves the house, the signal from the Wi-Fi network begins to get weaker as the user moves away from the house. At some point the signal begins to reach a threshold. This is when the task notifies the decision unit, that it is not able to proceed its activity. The user has informed the system that when or if Wi-Fi is not an option, uses a 3G network instead. The decision unit then either activates a new task, which supports 3G streaming or it instructs the current task to use a 3G service. The choice depends on the implementation of the task. It can be argued that the task, should not notify the decision unit, if it could just use a 3G service. However, that depends if the task should have knowledge about which services exists on the device or not.

- Low-level goals: Are goals that a task should operate within to fulfill its requirements. These are the responsibilities a task receives from the decision unit. When a task has been activated, it receives instructions from the decision unit, about how its behavior should be. For instance, a task should only use GPS data within a certain latitude and longitude otherwise it should notify the decision unit. Or if the Wi-Fi signal drops below a certain threshold, the decision unit should be notified. After the initial startup procedure, the task receives information directly from the environment and not from the decision unit. Information from the decision unit only has to do with changing the behavior of a task.

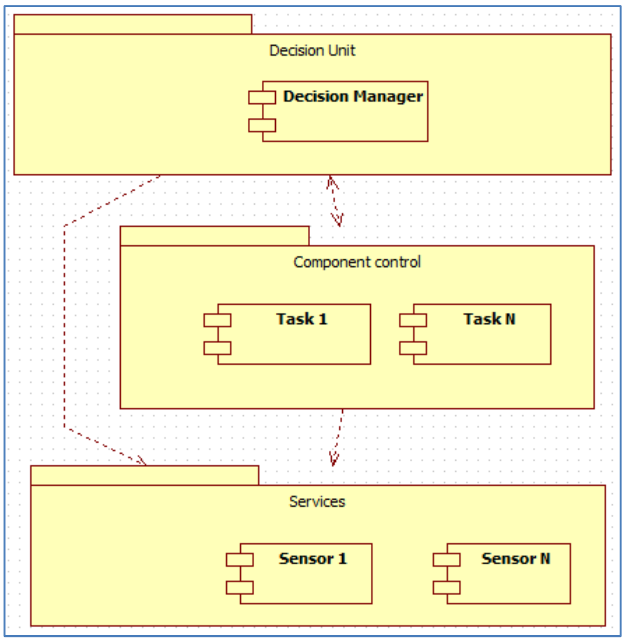


Figure 11: The relationship between the components of the system

The relationship between the decision unit, the tasks and the environment is illustrated in Figure 11. The environment is represented by services, as they are responsible for monitoring the environment.

5.2.2 Adding and Removing Tasks Dynamically

In order for a task, to fulfill the concept mentioned in section 4.2, it must support plug-in features. When a new task is required, it is activated by the decision unit and initial instructions are provided to it. Later when the task is not required anymore, it is told to release its resources before it is removed. The life of a task is then controlled by the decision unit.

The approach of combining plug-in features with MAS systems has been investigated in [17]. However, a matrix system is not exactly a MAS system, but the basic behaviour that both systems share, is the ability to adapt to the environment dynamically. In [17, page 191] this is described like "One of the foremost benefits is, that the plug-in architecture becomes dynamically extensible i.e. functionality can be altered, added or removed at runtime without the need to restart the application."

Because tasks and agents have many similarities, it is natural to apply a similar plug-in feature. [17] takes the plug-in paradigm a step further, by applying it recursively to agents. This means, that an agent can add and remove other agents to it-self. So far, this is not applied to the matrix architecture, as it would be difficult for the decision unit to adjust the behaviour of nested tasks.

Figure 12 illustrates which activities of the plug-in behavior are handled by the decision unit and the tasks.

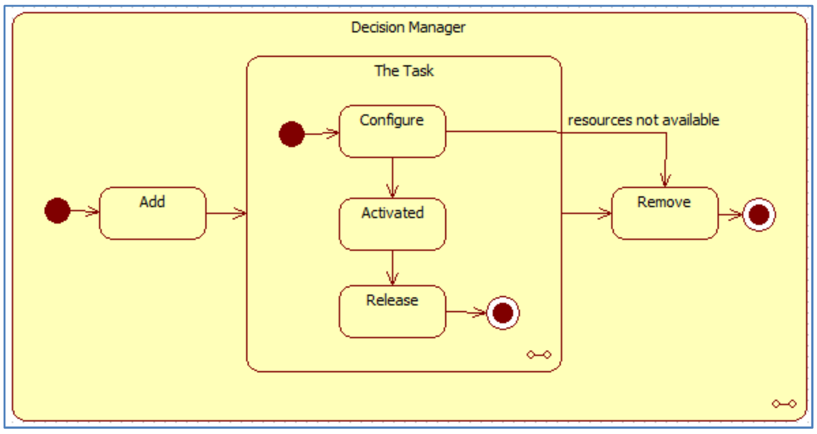


Figure 12: The activities of the plug-in behavior

- Add: the decision unit locates and creates the task and initializes its activation.
- Configure: the task receives instructions from the decision unit and verifies if all the resources it needs are present and available. If resources are not present or available, the task must instruct the decision unit. The decision unit then prepares the task to be removed.

- Activated: the task sets up resources for receiving information from the environment and instructions from the decision unit.
- Release: the task releases any resources used.
- Remove: the decision unit removes the task from the architecture.

In this way a framework is provided, controlling the life of the task. The responsibility of the decision unit is to create and remove tasks. The tasks must then ensure that the resources they need are available, before they start receiving information from the environment.

5.2.3 The Structure of a Task

In a matrix system the tasks should contact a single decision unit, if they cannot figure out what to do. They must not begin to interact with each another to resolve a high-level goal. The central decision unit ensures that the high-level goals are met. After the tasks are started, they receive information in real time directly from the services of the device.

This description of the behaviour, the responsibility of a task, and its relationships towards the decision unit, comes close to the description of the agent view mentioned in [17, page 193]. Here communication between agents (the tasks and the decision unit) is achieved through asynchronous message channels.

By seeing the decision unit as an agent, it makes sense to enable interaction between the tasks and the decision unit, as interactions between agents in MAS systems. In this way there exists a peer-to-peer relationship between the decision unit and the tasks, as it is the decision unit which connects services and tasks, while maintaining the high-level goals. However, neither the decision unit nor the tasks are truly agents. In the matrix system, the relationship between the decision unit and the tasks may be seen as a master – slave relationship. The decision unit provides responsibility to the tasks and the tasks tries to obey it. In a MAS system, the relationship between the agents is more like a master – master relationship. There is no centralized decision unit and each agent may take the lead to start a communication.

A task must therefore provide communication channels for:

- Receiving information from services: During run-time the tasks receives information from the environment through services provided by the device.
- Receive information from the decision unit: After the task is created, it receives instructions from the decision unit. Also after the task is activated, it can receive instructions from the decision unit to adjust its behavior.
- Send information to the decision unit: If information from services deviates from what is expected, the task must notify the decision unit.

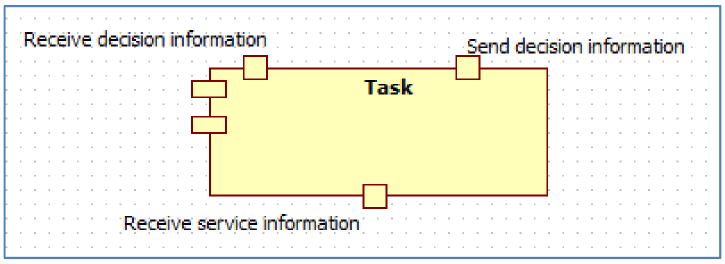

Receive decision information
Send decision information
Task
Receive service information

Figure 13: The design of a task

Since each task is a separated unit, they can still benefit from the techniques, mentioned in section 3.3.4, when modeling its internal structure. In this way, the strategy and paradigm best suited for the purpose, can be used. And further, keeping it in a separate unit, also supports the idea that a task can handle a specific kind of concern. This task can then have its own set of high-level goals required for this task to operate. These high-level goals must then be managed by the decision unit. In this way, the world is kept simple, as suggested in section 4.1.

In all of section 5.2 the decision unit handles many different responsibilities, such as determine when a new task must be created, the life-cycle of a task and send / receive information to / from the tasks. In the real system, the decision unit may consist of multiple collaborative units each with its own responsibility. However, for now it has been best to only address all responsibilities to a single unit for simplicity.

5.3 What Is the Difference

Now that the concept of the matrix architecture style has been presented, it is time to describe how it benefits from the approach in section 3 and how it differs from the three layer reference model in Figure 1.

Starting with the reference model, a self-manageable system may be seen as a three layer system. As presented in this thesis, the matrix system may be seen as just a two layered system. The reason for this is that (1) some of the responsibility from the *Change Management* layer of the reference model, will be performed by the tasks, and (2) because of this, the *Change Management* and the *Goal management* layers has merges into the decision unit of the matrix architecture. By also including MAS systems into the comparison, completes the differences. A MAS system may be seen as a one layered system, as each agent handles all three layers of the reference model. I.e. monitoring the environment and making both high-level and low-level decisions.

Reference model	Matrix architecture	MAS system
Goal management	Decision unit	Agents
Change management		
Component Control	Tasks	

Figure 14: Comparison of three similar systems

Because the reference model has three layers, does not mean that the system must be realized with three layers. The advantage of using a three layer reference model is to keep responsibilities separated, in order to get a clean picture of the system. However, many layers in a system which are deployed in a diverse environment, may reduce performance. Especially when implemented using a closed architecture[11]. The same is seen in the ISO OSI reference model, which describes a protocol stack of seven layers. However, there is no official supported implementation that implements all the layers separately. Some of the layers are merged, either to increase performance or the responsibilities have been closely related. Furthermore, to boost the performance just a bit more, the ISO OSI model is being implemented with an open architecture[12] and not as previously, with a closed architecture. ISO OSI implemented this way is an interested topic in the field of distributed systems. The reference model, therefore only describes a guide to how the system may be structured, what it should contain and how components are related. It does not describe how the system should actually be realised and deployed. The matrix architecture merges the two top most layers, because they are closely related and suggest an open architecture to increase the performance.

Looking at the actions from the feedback loop from section 3.3; some of the actions are handled by multiple components in the semi-decentralized feedback loop. This is accomplished by dividing the responsibility between the decision unit and the tasks. They both handles the collecting data, analysis and decision actions of the feedback loop, which suggest the decentralized approach, as opposed to the centralized approach described in section 3.3. By dividing the responsibility this way, the decision unit may be able to handle more high-level goals from multiple tasks, as the tasks takes care of low-level goals. Hence it handles scalability better. The decision unit is furthermore responsible for constructing the dynamic architecture in the act action, by adding and removing tasks (as mentioned at the end of section 5.2, in a real system, the responsibility of the decision unit may by divided between multiple collaborative units).

The MADAM project, mentioned in 3 Context-Aware Self-Adaptable Systems, also uses plug-in components. The benefit, compared with this project, is first that the responsibility is not divided between the *adaption manager* and the *components*. Hence, the MADAM project then uses a central controlled feedback loop, which reduces the scalability. Second, the *context manager* assigns environment information to the properties of the components. In the semi-decentralized approach, the tasks have direct access to this information, through the services of the device. Hence, there is not overhead in getting the environment information. And if the same information is communicated to different tasks, the tasks analyses the information differently as they may be used for different purposes. Then there is no reason to have a central component assigning this information to different components. This approach provides a better performance.

[11] A closed architecture is where layer N only uses services of layer N+1 and only communicates back to layer N-1.

[12] An open architecture is where layer N can use services of any layer below it-self and it can communicate back to any layers above it-self. Open architectures are also referred to a “cross layered communication”.

6 Implementation

This section gives a detailed architecture description of the implementation of the matrix architecture style using the semi-decentralized feedback loop approach. First, some architectural decisions are explained, which among other addresses the design challenges from section 5.2. Then the system is described in a top-down approach, starting with the model view which explains the static structures of the system. Then follows the component & connector view to describe the dynamic structures of the system.

6.1 Architectural Decisions

This section describes some problems which are found both before and during implementation of the system.

6.1.1 Implementation Language

A prototype of the system has been implemented using Microsoft .Net Framework and C#. The choice of using .Net and C# is mainly because it has a good support for threads and message queues and support of rapid development, which makes it good for prototyping.

6.1.2 Message Queues

As mentioned in section 3.3, several techniques can be used when communicating between concurrent components. Not all are suitable for a system like this, when all of the components executes on the same device and must be dynamically added and removed to the architecture. For instance, Remote Method Invocation (RMI) requires the presence of both a proxy and stub, which are probably impossible to create at runtime. However, RMI is best suited for distributed communication anyway. This leaves back event-based and message-passing communication. The choice of using a message-passing approach over an event-based approach is because message-passing requires a queue to store messages. If it should be the case that a component does not have time to handle a message, the message is stored in the queue, for later retrieval. Had it been event-based, the event would either be lost or the receiving component should store the event itself. Using a messaging-passing approach makes the component less fragile, as the queue stores the messages. This makes the component easier to support the plug-in feature.

Therefore, to support a loose coupling between the components of the system and at the same time asynchronous communication between components, message queue has been chosen. Further, the loose coupling between components supports the plug-in feature mentioned in section 5.2.2.

6.1.3 Managing the Global View

A complete implementation of context management is not supported. However, in the current implementation the task components only support simple low-level goals.

During the creation of a task, the decision unit supplies the task with a data range[13]. When a task receives data from a service, which is outside this data range, the task sends a notification to the decision unit using a message queue. The decision unit can then notify the task to adjust the data range.

6.1.4 Design of a Task Component

To support a seamless plug-in feature, each task must implement an interface, to get a uniform handle of tasks. Second, the use of message queues makes good support for loose coupling between components of the system.

This means that when the decision unit creates a task, it can use the interface to configure and activate a task. During the configuration step, the task can prepare itself to be activated, by setting up threads for listing for incoming data from services and notifications from the decision unit. Later, when the task must be removed, the decision manager also uses the interface to release the task. Here the task can stop listing for service data and notifications from the decision unit.

Further, because a task must be able to listen for incoming data from a service and notifications from the decision unit, while at the same time be able to send notifications to the decision unit, the task needs three queues.

6.1.5 Support of the Plug-In Feature

To support the dynamic established connections between components of the system, the decision unit provides the task with three message queues. One queue is used by every task, to communicate with the decision unit. The two others are unique to the task. One is used to connect the task to one or more services, so the task can receive data from the multiple services. Just as in the Observer pattern, here the subject is the service and the task is the observer. The other unique message queue is used to connect the task to the decision unit, so the decision unit can send messages to a specific task. Again, here the decision unit is the subject and the task is the observer.

The reason to make the decision unit communicate with a task using a unique queue, is because otherwise every task would receive every message send out by the decision unit. If every task receives other tasks messages, it would use precious time consuming irrelevant messages. The overhead at the tasks, would then far exceed the overhead, required by the decision unit to manage a queue for every task, which would decrease the overall response time of the system.

6.2 Model View

The system consists of two layers; a *Decision Unit* and a *Component* layer which are implemented in the *Decision Unit* and *Component* packages, respectively. A last

[13] Due to the time limit of the thesis, it has not been possible to implement full support of context management. In order to simulate context management, each task is associated with a data range. The data range can be expanded, when necessary, if a task receives data outside its data range.

package, the *Service* package, contains wrappers of sensors provided by the device, such as GPS, accelerometer, camera etc. However, services must be able to communicate with either the *Decision Unit* layer or any tasks contained in the *Component* layer. Hence, a wrapper to a service needs to be provided to integrate it into the system. All packages are shown above in Figure 11.

The following subsection shortly explains the content of each of the packages.

6.2.1 Decision Manager Package

This implementation of the *DecisionManager* is very simple. It only contains a *DecisionManager* class which is responsible for:

- Listing for notifications from services and tasks.
- Create tasks that know how to handle specific services. E.g. if there is no tasks to consume GPS data, the *DecisionManager* must create a task that understands GPS data.
- Create services that support a specific task. E.g. if the user activates a camera task and the camera service has not been activated already, the *DecisionManager* must create the camera service.
- Decide what to do if a task sends a notification. A task sends a notification to the *DecisionManager* when it receives data it does not know how to handle. The *DecisionManager* then estimates what to do. The current implementation only instructs the task to adjust its data range. It does not remove a task or create a new.
- Control the life cycle of tasks and services.

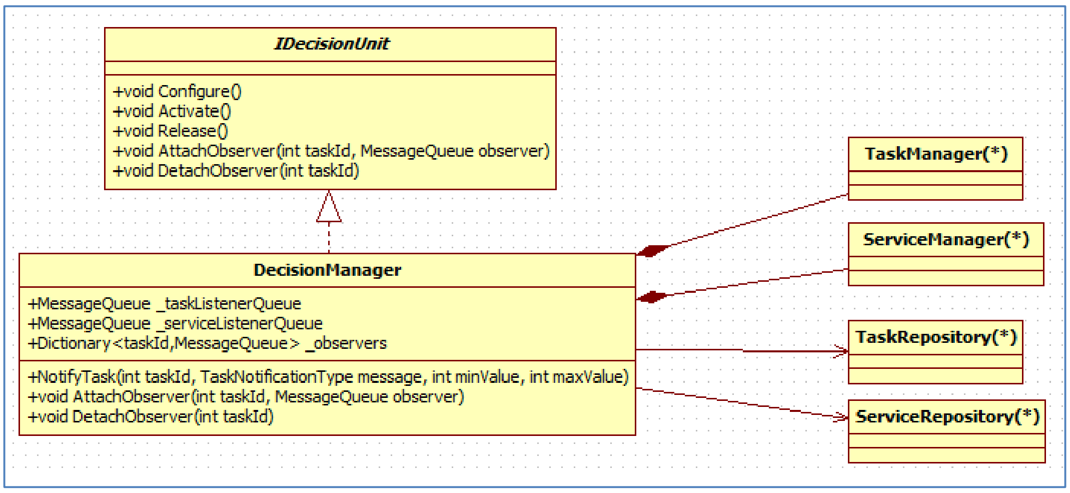

IDecisionUnit
+void Configure()
+void Activate()
+void Release()
+void AttachObserver(int taskId, MessageQueue observer)
+void DetachObserver(int taskId)
DecisionManager
+MessageQueue _taskListenerQueue
+MessageQueue _serviceListenerQueue
+Dictionary<taskId,MessageQueue> _observers
+NotifyTask(int taskId, TaskNotificationType message, int minValue, int maxValue)
+void AttachObserver(int taskId, MessageQueue observer)
+void DetachObserver(int taskId)
TaskManager(*)
ServiceManager(*)
TaskRepository(*)
ServiceRepository(*)

Figure 15: Class diagram of the Decision Unit package

The *DecisionManager* uses the *TaskManager* and *ServiceManager* to create tasks and services respectively, and further, to setup communication between services, tasks and the *DecisionManager*. When tasks and services are created, they are added to the *TaskRepository* and *ServiceReposity*, respectively. The *TaskManager*, *ServiceManager*, *TaskRepository* and *ServiceRepository* are explained in the following subsections.

6.2.2 Component Control Package

The Component Control package contains several classes used for handling tasks. Each task must support the plug-in feature, mentioned in section 5.2.2. This is done by implementing the *ITask* interface. Further, a task must be able to communicate with both the *DecisionManager* and the services, which requires the three message queues mentioned in section 6.1.4. When a task is activated, it creates two threads; one for listening for service data and the other for listening for instructions from the *DecisionManager*.

It is not possible to create or remove tasks directly, but only through the *TaskManager* class. The *TaskManager* class is also responsible for connecting a task with its services and the *DecisionManager*. When a task is created, it is placed in the *TaskRepository*, which implements the Singleton pattern.

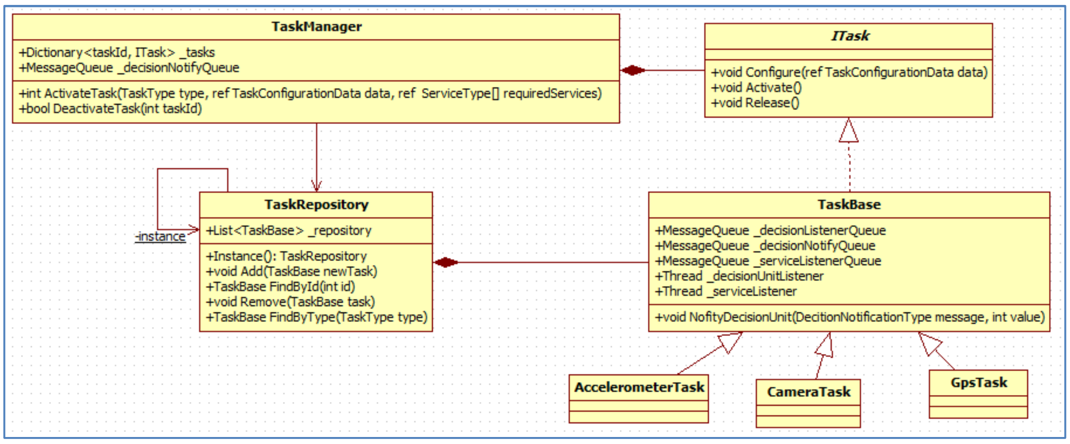

TaskManager
+Dictionary<taskId, ITask> _tasks
+MessageQueue _decisionNotifyQueue
+int ActivateTask(TaskType type, ref TaskConfigurationData data, ref ServiceType[] requiredServices)
+bool DeactivateTask(int taskId)
ITask
+void Configure(ref TaskConfigurationData data)
+void Activate()
+void Release()
TaskRepository
-instance
+List<TaskBase> _repository
+Instance(): TaskRepository
+void Add(TaskBase newTask)
+TaskBase FindById(int id)
+void Remove(TaskBase task)
+TaskBase FindByType(TaskType type)
TaskBase
+MessageQueue _decisionListenerQueue
+MessageQueue _decisionNotifyQueue
+MessageQueue _serviceListenerQueue
+Thread _decisionUnitListener
+Thread _serviceListener
+void NofityDecisionUnit(DecitionNotificationType message, int value)
AccelerometerTask
CameraTask
GpsTask

Figure 16: Class diagram of the Component Control package

6.2.3 Service Package

As in the Component Control package, the Service package contains several classes used for handling services. The structure between the classes, is the same as the classes in the Component Control package. To get a uniform handling of services, each services implements the *IService* interface. Each service can further be connected to several tasks, which is why they must have a list of observers, to which they can send data. When a service is activated, it starts a thread, which continually sends data to the observers, until the service is taken down.

It is not possible to create or remove services directly, but only through the *ServiceManager* class. The *ServiceManager* class is also responsible for connecting a service with the *DecisionManager* and attaching tasks (observers) to it, whenever a task is created. When a service is created, it is placed in the *ServiceRepository*, which implements the Singleton pattern.

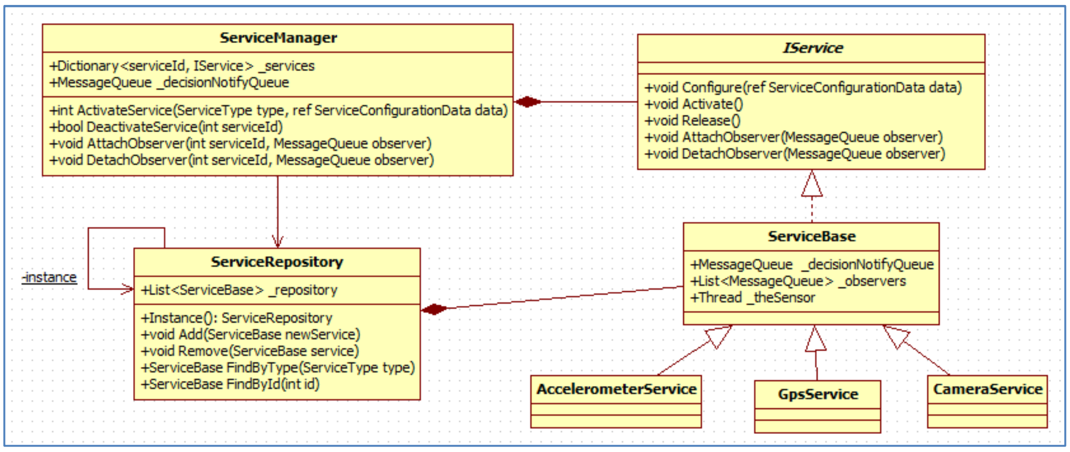

ServiceManager
+Dictionary<serviceId, IService> _services
+MessageQueue _decisionNotifyQueue
+int ActivateService(ServiceType type, ref ServiceConfigurationData data)
+bool DeactivateService(int serviceId)
+void AttachObserver(int serviceId, MessageQueue observer)
+void DetachObserver(int serviceId, MessageQueue observer)
IService
+void Configure(ref ServiceConfigurationData data)
+void Activate()
+void Release()
+void AttachObserver(MessageQueue observer)
+void DetachObserver(MessageQueue observer)
-instance
ServiceRepository
+List<ServiceBase> _repository
+Instance(): ServiceRepository
+void Add(ServiceBase newService)
+void Remove(ServiceBase service)
+ServiceBase FindByType(ServiceType type)
+ServiceBase FindById(int id)
ServiceBase
+MessageQueue _decisionNotifyQueue
+List<MessageQueue> _observers
+Thread _theSensor
AccelerometerService
GpsService
CameraService

Figure 17: Class diagram of the Service package

6.3 Component & Connector View

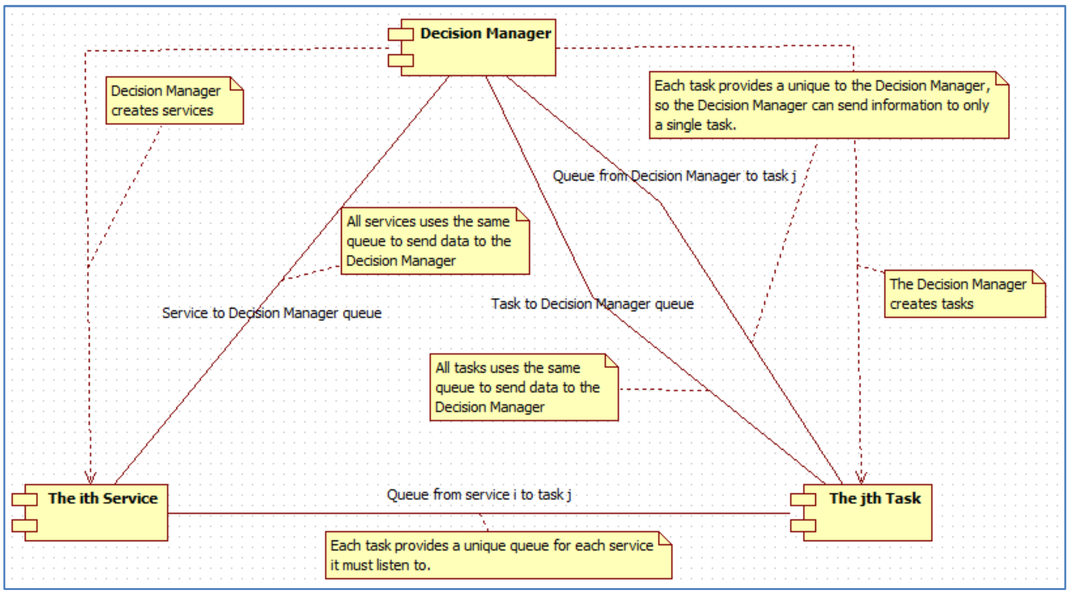
During system activity the components of the system must communicate with each other to support the semi-decentralized approach. This is done by sending messages using message queues, which connects the components. This section first shows how message queues are used to connect the components. Then follows some specific aspects such as how the services and tasks are created. And also how tasks communicate with the *DecisionManager*, both during normal execution, but also when they receive data from services they do not know how to handle. Appendix A contains a log showing a five second execution of the system.

6.3.1 Communication Flow

As explained previously, the communication between the components of the system is done using message queues. All services are connected to the *DecisionManager* using the same message queue. This is because the *DecisionManager* must be able to get notifications from a service, if there are no tasks attached to it. The *DecisionManager* can distinct the services by the service id, which is included in the message.

All tasks must also communicate with the *DecisionManager*, if they encounter data from the services that are outside of its data range. The *DecisionManager* can distinct the tasks by the task id, which is included in the message.

However, some of the communication channels require a unique queue between the components. Because a task can be connected to several services, each of these services must be connected to the task, using a queue provided by that task. This suggests the use of the Observer pattern, where a service is the subject and a task is the observer. The same relationship exists between the *DecisionManager* and a task. The *DecisionManager* must be able to communicate with a specific task, when that task must be notified with new information. This is why a task must provide a message queue for the *DecisionManager*. Here the *DecisionManager* is the subject and the task is the observer.

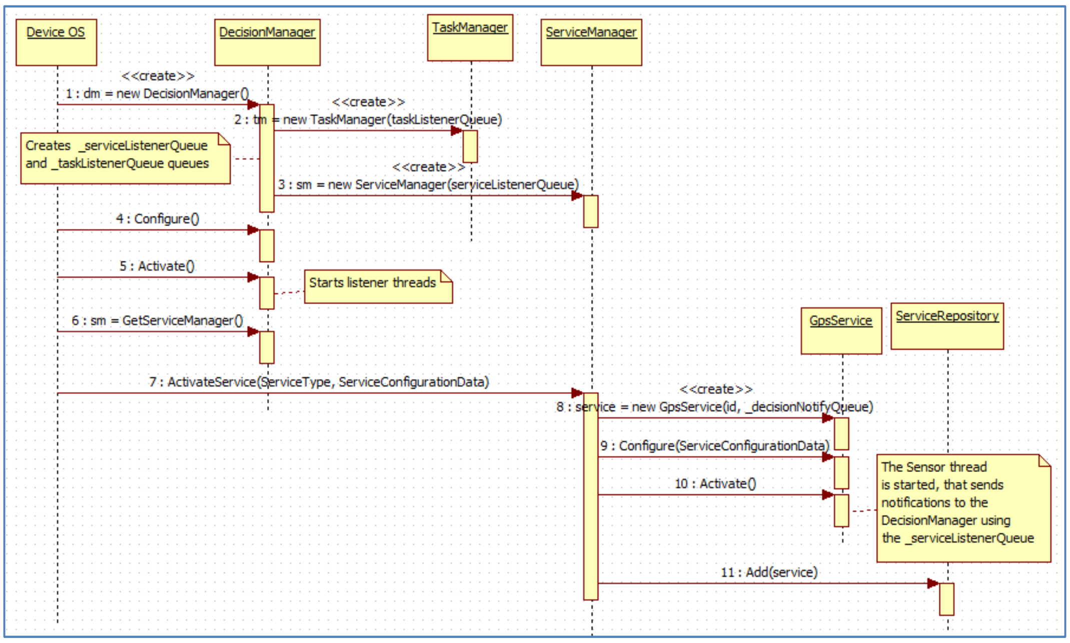


Decision Manager
Decision Manager creates services
Each task provides a unique to the Decision Manager, so the Decision Manager can send information to only a single task.
Queue from Decision Manager to task j
All services uses the same queue to send data to the Decision Manager
The Decision Manager creates tasks
Service to Decision Manager queue
Task to Decision Manager queue
All tasks uses the same queue to send data to the Decision Manager
The ith Service
Queue from service i to task j
The jth Task
Each task provides a unique queue for each service it must listen to.

Figure 18: Components of the system and how they are connected

6.3.2 Creating Services and Tasks

To startup the system, the device must create an instance of the *DecisionManager* and activate it. If the device is exposed to implicit actions, as mentioned in section 5.2.1, it can use the *DecisionManager*, to obtain an instance of the *ServiceManager* (as in step 6 of Figure 19). The *ServiceManager* can then be used to create services, using the *ServiceManager::ActivateService()*. Services are then added to the *ServiceRepository*.



Device OS
DecisionManager
TaskManager
ServiceManager
<<create>>
1 : dm = new DecisionManager()
<<create>>
2 : tm = new TaskManager(taskListenerQueue)
Creates _serviceListenerQueue and _taskListenerQueue queues
<<create>>
3 : sm = new ServiceManager(serviceListenerQueue)
4 : Configure()
5 : Activate()
Starts listener threads
6 : sm = GetServiceManager()
GpsService
ServiceRepository
7 : ActivateService(ServiceType, ServiceConfigurationData)
<<create>>
8 : service = new GpsService(id, _decisionNotifyQueue)
9 : Configure(ServiceConfigurationData)
10 : Activate()
The Sensor thread is started, that sends notifications to the DecisionManager using the _serviceListenerQueue
11 : Add(service)

Figure 19: Sequence diagram of how to create a service

The current implementation of the system does not support explicit action. The reason is that the *DecisionManager* does not lookup services, required by the system, when a new task needs to be created. The *DecisionManager* only supports the creation of a task, when it is needed by a service – an implicit action. This is shown here in Figure 20.

Here the *DecisionManager* receives a notification from a service, when there are no tasks attached to the service. The *DecisionManager* uses the *ServiceRepository* to find out what type of service it has received a notification from. Based on the type of service, the *DecisionManager* creates a task to handle the service. After the task is activated, it is added to the *TaskRepository* and then connected to both the service and the *DecisionManager*.

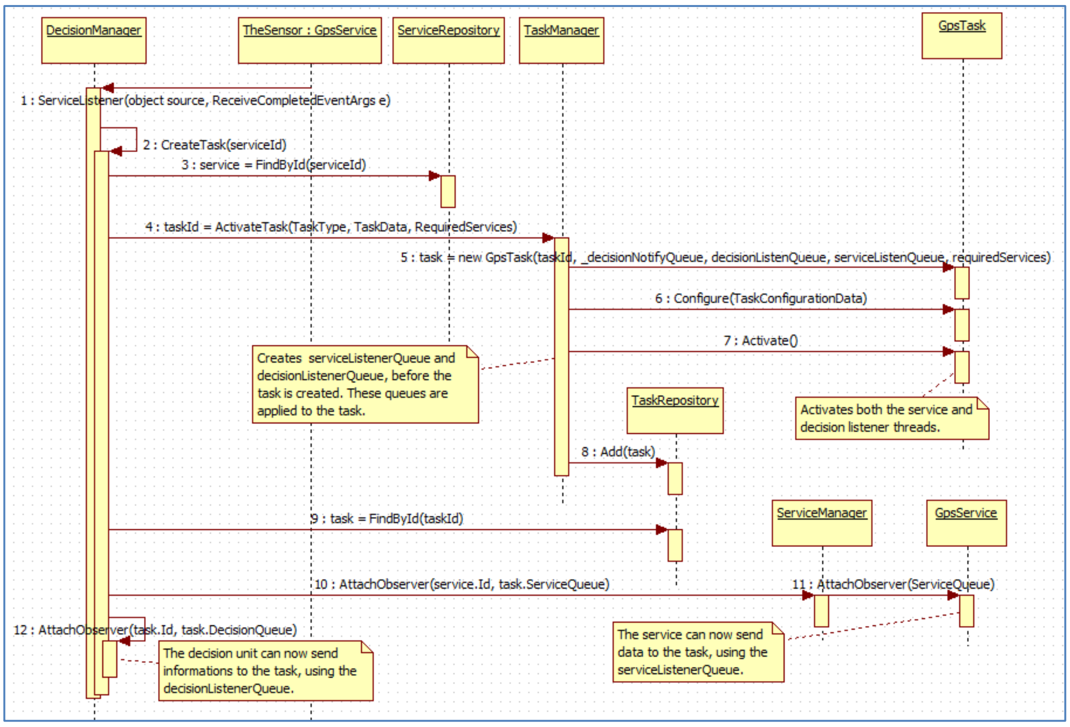

DecisionManager
TheSensor : GpsService
ServiceRepository
TaskManager
GpsTask
1 : ServiceListener(object source, ReceiveCompletedEventArgs e)
2 : CreateTask(serviceId)
3 : service = FindById(serviceId)
4 : taskId = ActivateTask(TaskType, TaskData, RequiredServices)
5 : task = new GpsTask(taskId, _decisionNotifyQueue, decisionListenQueue, serviceListenQueue, requiredServices)
6 : Configure(TaskConfigurationData)
7 : Activate()
Creates serviceListenerQueue and decisionListenerQueue, before the task is created. These queues are applied to the task.
TaskRepository
Activates both the service and decision listener threads.
8 : Add(task)
ServiceManager
GpsService
9 : task = FindById(taskId)
10 : AttachObserver(service.Id, task.ServiceQueue)
11 : AttachObserver(ServiceQueue)
12 : AttachObserver(task.Id, task.DecisionQueue)
The decision unit can now send informations to the task, using the decisionListenerQueue.
The service can now send data to the task, using the serviceListenerQueue.

Figure 20: Sequence diagram of how a task is created

6.3.3 Tasks Communicating With the Decision Unit

When a task receives data from a service, which is outside its data range, the task notifies the *DecisionManager* about this. In the current implementation of the system, the *DecisionManager* does not have any logic concerning the global view. Therefore, it just retransmits the same data back to the task. The task then expands its data range, either by lowering the minimum value or raising the maximum value of the data range.

All communication between the service, the task and the *DecisionManager*, is done using the unique queues associated with the task during the creation of the task.

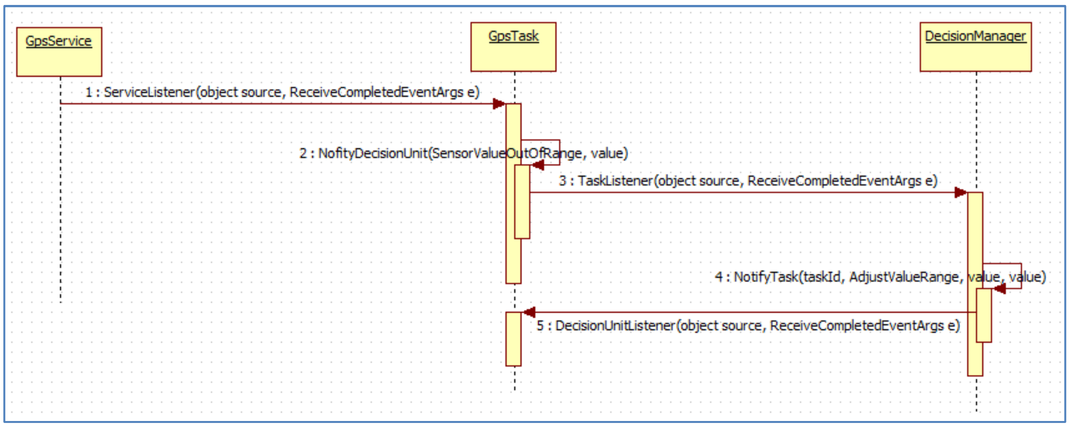

GpsService
GpsTask
DecisionManager
1 : ServiceListener(object source, ReceiveCompletedEventArgs e)
2 : NofityDecisionUnit(SensorValueOutOfRange, value)
3 : TaskListener(object source, ReceiveCompletedEventArgs e)
4 : NotifyTask(taskId, AdjustValueRange, value, value)
5 : DecisionUnitListener(object source, ReceiveCompletedEventArgs e)

Figure 21: A task notifies the *DecisionManager* if the received data from a service is outside of the data range

6.3.4 Releasing Resources

When the system needs to be removed, the *DecisionManager* releases its resources by calling the *ServiceManager* and the *TaskManager* on every service and task it has created.

When a service is removed, it stops its internal thread, which sends data to any task connected to it. Similar, when a task is removed, it stops its two internal threads which are listening for information from the *DecisionManager* and data from services connected to the task. Finally, the *DecisionManager* deletes the queues that connected the task.

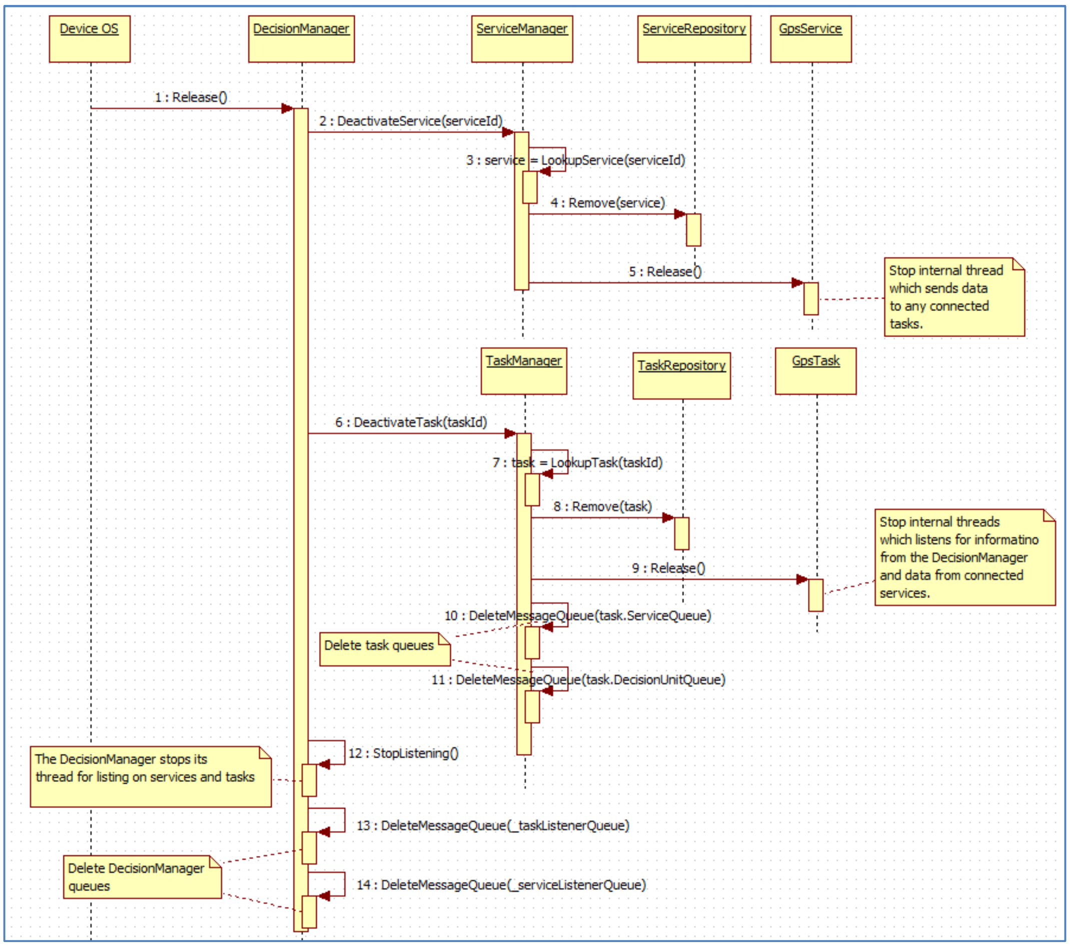

Device OS
DecisionManager
ServiceManager
ServiceRepository
GpsService
1 : Release()
2 : DeactivateService(serviceId)
3 : service = LookupService(serviceId)
4 : Remove(service)
5 : Release()
Stop internal thread which sends data to any connected tasks.
TaskManager
TaskRepository
GpsTask
6 : DeactivateTask(taskId)
7 : task = LookupTask(taskId)
8 : Remove(task)
9 : Release()
Stop internal threads which listens for informatino from the DecisionManager and data from connected services.
10 : DeleteMessageQueue(task.ServiceQueue)
Delete task queues
11 : DeleteMessageQueue(task.DecisionUnitQueue)
12 : StopListening()
The DecisionManager stops its thread for listing on services and tasks
13 : DeleteMessageQueue(_taskListenerQueue)
Delete DecisionManager queues
14 : DeleteMessageQueue(_serviceListenerQueue)

Figure 22: The DecisionManager releases all its resources

It might also be the case that a task needs to be removed. This event may either come from the device, if the user closes the task or if the *DecisionManager* decides to do so. Here the *DecisionManager* first detaches the task from its services, so the services no longer can send data to the task. Afterwards the task is deactivated, which will stop the task from sending any notifications to the *DecisionManager*. And, finally, the task is removed as an observer of the *DecisionManager*.

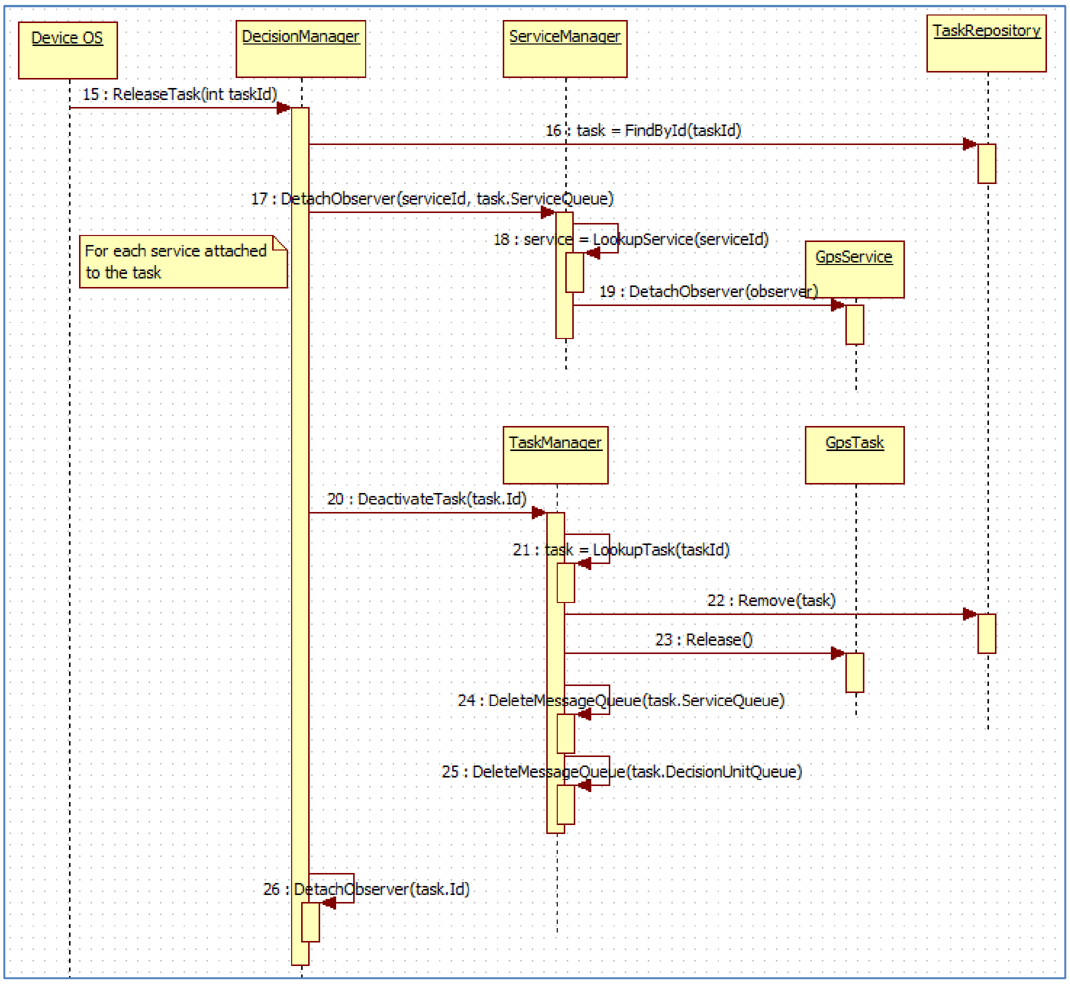

Device OS
DecisionManager
ServiceManager
TaskRepository
15 : ReleaseTask(int taskId)
16 : task = FindById(taskId)
17 : DetachObserver(serviceId, task.ServiceQueue)
18 : service = LookupService(serviceId)
For each service attached to the task
GpsService
19 : DetachObserver(observer)
TaskManager
GpsTask
20 : DeactivateTask(task.Id)
21 : task = LookupTask(taskId)
22 : Remove(task)
23 : Release()
24 : DeleteMessageQueue(task.ServiceQueue)
25 : DeleteMessageQueue(task.DecisionUnitQueue)
26 : DetachObserver(task.Id)

Figure 23: DecisionManager releases task resources

Releasing the resources for a service may happen if the user does not want to use this service any more. The process is quite simple, compared to releasing resources for a task. When the service is deactivated, it does not send any data to any tasks – and that is it. If this was the only service a task was listening to, then the queue, used for listening for service data, will time out if it does not receive any data in a specified amount of time. This event would then activate a notification to the *DecisionManager*, which could find out if the service is still running or not. If the service is not running, the *DecisionManager* should inform the task about it. Another way is for the service to notify the *DecisionManager* that it is being removed. The *DecsionManager* could then notify any tasks attached to the service. However, neither of these approaches has been implemented in the current version of the system.

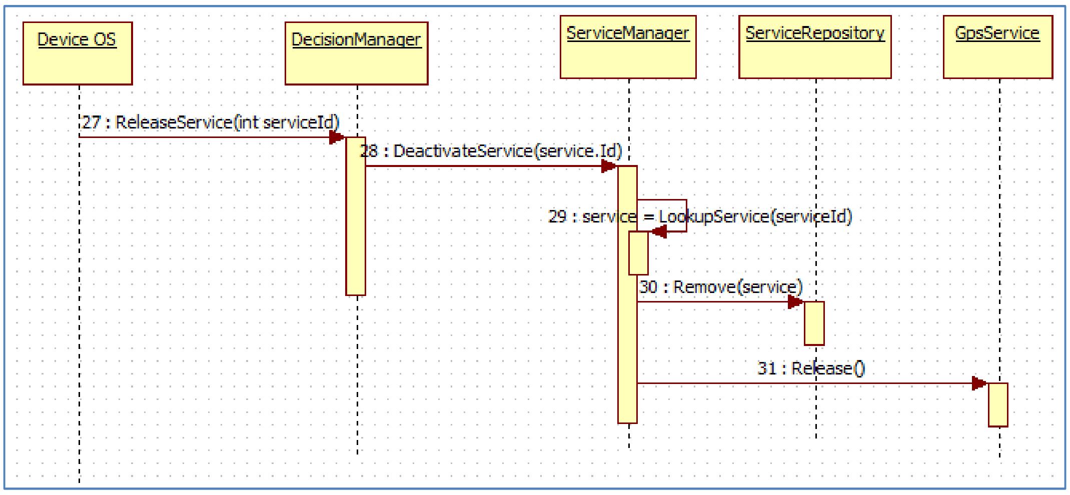

Device OS
DecisionManager
ServiceManager
ServiceRepository
GpsService
27 : ReleaseService(int serviceId)
28 : DeactivateService(service.Id)
29 : service = LookupService(serviceId)
30 : Remove(service)
31 : Release()

Figure 24: DecisionManager releases service resources

7 Evaluation and New Challenges

Now that the system has been implemented, it is time to evaluate it. The evaluation is described in section 7.1 and focuses on the design goals mentioned in section 5.2 (which addresses the limitations and challenges from section 4).

Even though the new matrix architecture style addresses limitations and challenges from previous approaches, it also provides challenges of its own. These are mentioned in section 7.2.

7.1 Evaluation

The evaluation of the system is conducted by writing messages to the console window, during execution of the system and saved in log files (See Appendix B). This sequence of messages is used to observe the communication between the components of the system. Section 7.1.1 first evaluates the overall system using the sampled data from Appendix B. Then section 7.1.2, 7.1.3 and 7.1.4 evaluates the design goals from section 5.2.

7.1.1 Evaluation of Sample Data

This section shows results from different test runs of the system. Each test run focuses on a different aspect of the system and is executed three times.

Test Run 1

All tasks are created with a correct data range and associated with a correct type of service. This setup simulates a normal execution where all tasks are correctly activated. It is therefore used as a reference.

	Execution 1	Execution 2	Execution 3	Average
Total calls from services to decision unit	87 *)	88 *)	89 *)	88
Total calls from tasks to decision unit	0	0	0	0
Total calls from decision unit to tasks	0	0	0	0
Total calls from services to tasks	950	995	866	937

Figure 25: All tasks are created correct

*) During the activation of a service, it notifies the decision unit when it has been configured and when there is no tasks attached to it. Therefore each service notifies the decision unit twice during activation. If the number is odd, then one of the services was properly removed before it notified the decision unit the second time.

Since each task is configured with a correct data range and associated with the correct service, none of them needs to notify the decision unit, of invalid data received from a service.

Test Run 2

All tasks are configured with a wrong data range, but they are associated with the correct type of service. This setup illustrated how much communication is required between the decision unit and the tasks, to correct the data range.

	Execution 1	Execution 2	Execution 3	Average
Total calls from services to decision unit	97	90	84	90,33
Total calls from tasks to decision unit	306	238	353	299
Total calls from decision unit to tasks	227	175	289	233,33
Total calls from services to tasks	943	857	949	916,33

Figure 26: All tasks are created with a wrong data range

As it can be seen, the tasks send more notifications to the decision unit, than the decision unit sends notifications back to the tasks. This is because a task is able to send more notifications to the decision unit, before the decision unit instructs it to adjust its data range.

The number of calls from services to the tasks remains roughly the same, when compared to Test run 1. However, the implementation to handle this behavior doesn't make any fancy decisions or computing, which is why the number of "*Total calls from tasks to decision unit*" is low, compared to Test run 3.

Test Run 3

All tasks are associated with a wrong type of service, but they are configured with a correct data range. This setup illustrates how much communication is required between the decision unit and the tasks, to associate the tasks with a correct service.

	Execution 1	Execution 2	Execution 3	Average
Total calls from services to decision unit	96	90	91	92,33
Total calls from tasks to decision unit	1135	957	765	952,33
Total calls from decision unit to tasks	0	0	0	0
Total calls from services to tasks	4980	3272	4019	4090,33

Figure 27: All tasks are associated with a wrong service

The number of "*Total calls from tasks to decision unit*" is the largest among the test runs. This has to do with the time it takes the decision unit to find a new service to a task. During this time, a task seems to have plenty of time to send many notifications to the decision unit. However, the decision unit does not notify a task, when it has associated a new service to the task, which is why the "*Total calls from decision unit to tasks*" is zero.

The number of "*Total calls from services to task*" is also very high. This has something to do with the message queues. Messages in a message queue are not removed from the queue. This means, when a task is associated to a new service, the task receives all the messages produced so far by that service. Therefore a service first sends messages to the wrong task, because this test run associates a wrong type of service to a task. Then all the same messages are send to the correct task. However, half the number of "*Total calls from services to tasks*" is still about twice the size compared to the other test runs. There is no logical explanation for that at the moment.

7.1.2 The Global View

Even though managing the global view is not on focus in this thesis, it does have an effect on the scalability of the system. As can be seen from section 7.1.1, if a task is associated with a wrong service or the data range is not properly initialized by the decision unit, it requires more communication between the task and the decision unit. Limiting the communication on these channels increases the systems capabilities to scale, as more resources are given to the decision unit. These resources can then be used to make more accurate decision and managing the life of services and tasks.

Section 7.1.1 also shows that the most communication is performed between the services and the tasks. This is exactly the idea of the semi-decentralized feedback loop. There is no reason to involve the decision unit, as long as a task is associated with correct services and receives data within the data range.

Therefore, the right split of responsibility between the decision unit and the tasks, is a challenge. It is also very important for the overall system performance.

7.1.3 Self-Adaptability

As it can be seen from the section 7.1.1 and the sequence diagrams from section 6.3.2 and 6.3.4; both tasks and services are seamlessly added and removed dynamically.

In order for two components to be easily connected, they must be loose coupled and only communicate using a connector. The connector should handle the communication protocol, so the components only implement a minimum of logic to handle the connector and send and receive messages.

This approach has been implemented in the prototype. Because the decision unit handles the creation of both the services and tasks, and connects them, they are unaware of each other's existents. This means that a task does not need to know how to find the services it needs, nor how to connect to services. It only has to concentrate on its task, which makes it a compact component that can easily be added and removed.

The use of messages queues (the connector) requires only a minimum of logic implemented in both the service and the task. The message queue handles the communication protocol and stores any messages internally.

The cocktail of loose coupling and message queues supports the concept of the matrix architecture style and strongly supports self-adaptability.

The other part of self-adaptability, which is not addressed in this thesis, is for the decision unit to be able to make the right decision and configure the tasks with accurate data.

7.1.4 The Tasks

Even though the decision unit both handles the creation and removing of tasks and services, most of the activity goes on between the services and the tasks, as can be seen in section 7.1.1. This makes tasks a very important component. It not only relieves the decision unit from a lot of communication from services, but it also takes some low level decisions itself.

Data from a service is pushed out to a task through a message queue. This works fine, as long as the task is associated with the correct service and data is within the data range. However, when data is outside the data range, the data is still pushed out with the same frequency. In the current implementation of the prototype, the task notifies the decision unit on every occasion it received an invalid data. If many consecutive invalid data arrives at the task, this overloads the decision unit, which still must be able to handle notifications from other tasks.

One solution would be to stop the task from, either notifying the decision unit or listing for more data from the service, while it awaits a decision from the decision unit. Another approach would be to use a pull strategy between the service and task, instead of the current implemented push strategy. With a pull strategy, a task collects data from the service at its own frequency.

Using a pull strategy, a service could update its data when it differs from the previous value by a certain threshold. Tasks can then fetch the data whenever they want. However, the connector between a task and a service must still support a loose coupling, to support the self-adaptability feature mentioned in section 7.1.3. Otherwise a blackboard pattern between the services and the tasks may be used. Here each service uses a push strategy to push data onto the blackboard, where a task can use the pull strategy to collect data. This approach, however, may just complicate the design too much and decrease the performance of the overall system. Nevertheless, it is a feasible solution.

Besides the issues of receiving invalid data at a task, the design and implementation of a task works well with the concept of a matrix architecture style. And, again, the better the decision unit is to configure a task with accurate data, the lesser the possibility is for the task to receive invalid data and notify the decision unit about it.

7.2 New Challenges

As always, architectures have challenges which must be met and the matrix architecture style is no different. This section describes some of the new challenges the matrix architecture style introduces, when it is used for a context-aware self-adaptable system.

7.2.1 Notification of the Decision Unit

When should a task notify the decision unit, that it can't use data from a service? Should it be on the first capture of invalid data or should it wait to see if consecutive data also are invalid?

In the prototype implementation of the system, the task notifies the decision unit on the each occurrence of invalid data. This means that the decision unit uses resources on each notification. In a situation where the occurrence only happens once or rarely, this approach might not be the best.

Another approach would be to use a technique similar to catching fish! When you want to catch a fish, the fish must be hocked before you pull out of the water. Fish are smart creatures. When they see the bait, they first taste it by taking small bits of the bait. You may not catch the fish, if you don't know when to hock it. Compared to the task of receiving data, this is similar to receiving invalid data once in a while. But when do you know, when this invalid data may become relevant. In the case of the fish, you observe what is going on either by looking at the cork or touching the fish line. If the cork starts bouncing up and down on the surface or you feel the fish tugging the fishing line, you need to wait and see (or feel), if this behavior continues. If it does, then it is time to hock the fish. Otherwise the fish just tastes you bait and didn't like it. Compared to the task receiving data, this is similar to receiving invalid data. When the first occurrence happens, wait a while to see if more invalid data arrives. If it does, notify the decision unit. Otherwise this is just one of the rare occurrences and the decision unit does not need to be informed.

However, fish can also be very hungry. Sometimes you are lucky and the fish takes the bait and the hock in one mouthful. When this happens, there is no doubt that you have caught the fish. Comparing the task receiving data, this would be the case where the task continuously receives invalid data and they deviate a lot from what is expected. Then it is time to notify the decision unit.

There might be other ways to estimate when to notify the decision unit. Finding the right time is important, otherwise it would overload the decision unit, as seen from the test runs from section 7.1.1.

The topic of notifying the decision unit about invalid data, goes back to the subject of keeping the world simple, as mentioned in section 4.1. The simpler the world gets, the easier it is to interpret the data from the environment, as there are fewer kinds of data to comprehend. And the fewer kinds of data there is, the better the decision unit is to configure the task and the better the task can make proper low-level decision about invalid data.

7.2.2 Low Communication

It may be difficult for the decision unit to configure a task properly. This means that the task must notify the decision unit when it encounters invalid data from a service. If this happens too often, the system tends to behave like a MAS system (more communication than computation). The more autonomous the tasks are, the more effective the system becomes.

The fishing technique mentioned in section 7.2.1 may be used to keep communication low.

7.2.3 Service Knowledge

It is important for a service to know when its data has been used, otherwise it must notify the decision unit to create a task for the purpose. If the decision unit can't find any receivers for the data, the service can be removed.

In the current prototype implementation, a service holds a collection of observers. If the collection is empty, the decision unit is notified. However, this approach only works with a push strategy, as mentioned in section 7.1.4. If a pull strategy is used, a service does not know if there are any receivers for its data.

The advantage of using a pull strategy is that a task can retrieve data at its own rate. However, a pull strategy provides problems for a service to find out if its data has any receivers. The blackboard pattern, mentioned earlier, may also be used to solve this problem. If there are no tasks to retrieve service data from the blackboard, the blackboard will have to remove the service itself. In this way the decision unit have delegated some responsibility to the blackboard. Another approach is for the blackboard to notify the decision unit to remove a service. And, as before, adding the blackboard pattern may just complicate the design too much and decrease the performance of the overall system.

The advantages of using the push strategy, is that it keeps the design and communication simple.

8 Summary

The purpose of this thesis has been to design a new architecture style for context-aware self-adaptable systems, which improves performance and scalability of the system. The approach to solve these challenges may be many, especially since the context-aware self-adaptable systems combine two large research fields – context-aware and self-manageable. Attempts to construct self-manageable systems, has originated from both the robotic research and biologic research. The robotic approach tends to utilize the approach presented in section 3, while the biologic approach is more in the direction of MAS systems. Because of the many different approaches, it is likely that there has already been attempted improvement upon performance and scalability, within the field of context-aware self-adaptable system.

However, due to the time limit of this thesis, it has not been possible to conduct a thoroughly research, in finding other projects addressing this matter and / or constructing a similar architecture.

Nevertheless, the matrix architecture style presented in this thesis, is constructed around the concept of a matrix. It takes advantages of providing the components of the system to be added and removed dynamically, by using a plug-in mechanism. Further, the responsibility of managing the global system view has been divided between the components and a central decision unit. This division suggests a semi-decentralized feedback loop, as opposed to the traditional central controlled feedback loop.

In order to find weaknesses about the traditional approach, the thesis starts with a survey of context-aware self-manageable systems. These systems are built using three main parts; the architecture, a context model and a feedback loop. Only the architecture and feedback loop has been addressed in this thesis.

In the traditional approach the architecture has three layers, each with a well defined responsibility. The lower layer (*component control*), monitors the environment and reports changes to the *change management* layer just above it. It also receives instructions from the *change management* layer, to adjust its components or create some new. The middle layer (*change management*), has rules for managing the current set of components. If the changes from the *component control* layer can't be met, the upper layer (*goal management*) must be notified. The *change management* receives new rules from the *goal management* layer, when new plans are introduced into system. The upper layer (*goal management*), uses the current state of the system, combined with a specification of high-level goals, to attempt to achieve new plans. The new plans are given to the *change management* layer.

The context model is responsible for modelling the environment. It helps the system to understand the context in which it is being used. Previous systems often focus on static context, such as location and information about sights. However, recent systems further includes dynamic context as well, such as device runtime status and service execution time. Another recent approach is to use Semantic Web techniques, such as ontologies, to model the context. Other approaches also exist, such as set theory, directed graphs, key-value pairs and object-oriented models.

The feedback loop ensures that the system is continuously monitoring the environment and makes some reasoning about the information collected from the environment. This is done by ensuring the execution of four actions; collect, analyze, decide and act. The *collect* action is used for monitoring the environment, by ensuring that input from the user or sensors are propagated into the system. Then follows the *analyze* action, which analyzes the input data and detects any changes. The system must then make some reasoning about the data, to find out if the architectures of the system must be changed. This part is performed by the *decide* action. Finally, if the architecture needs to be changed, it is the job for the *act* action.

Even though systems being build, using this traditional approach, are stable, they do have some limitations and challenges. These limitations and challenges are either directly or indirectly related to the performance and scalability of the system. Three such issues have been addressed in this thesis, to suggest a new architecture style.

The first topic addresses the subject of keeping the environment simple. The amount of data in the environment is huge and it must be processed fast and with accuracy, to support a context-aware self-adaptable system. Previous systems tend to go in the direction of monitoring everything in the environment, in order to support a truly self-adaptable system. That is, a system that by it-self knows when it must adapt to a new situation, to complement human activities. However the more diverse data the system receives from the environment, the more complicated rules are needed to narrow down the intention of the user. Instead it should focus on a smaller subset of the environment, and hence focus on a smaller subset of critical high level goals. Therefore, by providing non-overlapping tasks which each handles a specific aspect of the environment, the environment gets simpler and easier to manage. And, further, it should only monitor what is required to fulfil the current set of active tasks and make support for explicit user actions as well.

The second topic addressed, is how to be self-adaptable. Self-adaptability requires two things; a unit that can make a decision when to adapt to a new situation and support for adaption at both the architecture and component level. The first requirement, the decision unit, is not covered in this thesis. However, adaptability is. The techniques used, must support adaptability at run-time. At the architecture level, this thesis suggests a framework supporting a plug-in feature, which manages the life of components and allow components to be added and removed at run-time. The component may then take advantages of techniques used to build product line applications. Each component support the first topic addressed; to support a specific aspect area of the environment and therefore are none-overlapping.

A framework that supports a plug-in feature where components can be added and removed at run-time, requires a unit to decide when a component must be added and removed. The third topic addressed, is to provide a feedback loop different from the centralized controlled feedback loop, which had a scalability issue. This thesis suggests a new approach, to prevent scalability issues. The new approach is a semi-decentralized feedback loop, which splits the responsibility of make decisions, among the components and the decision unit. The decision unit is responsible for adding and removing components to the architecture. When a component has been added to the architecture, it receives some responsibility from the decision unit, before it starts listing to the environment. The component then receives environmental information directly from sensors, analyzes the information and makes local decisions. In this way, components handle low-level decisions, by taking over some of the responsibility from the decision unit. The decision unit then has more resources to handle high-level critical goals.

To combine the solution of these three subjects required a new architecture. This new architecture supports the concept of a matrix where components can be added and removed dynamically at run-time. A decision unit is used to handle high-level critical goals and to decide when to add and remove components. Components then receive environmental information directly from sensors, which they analyze. The responsibility delegated from the decision unit, is used to make low-level decisions, based on the analysis of the environmental information. In this way, the matrix architecture style is a two level architecture, instead of the traditional three layered architecture. Here the *goal management* and *change management* layers, from previously, have merged into a *decision unit* layer. The *component control* layer remains the same, except that the components are now able to make low-level decisions.

This matrix architecture style addresses the performance and scalability limitations from the traditional approach. Scalability is partly achieved by the semi-decentralized feedback loop, which suggests splitting the responsibility between a central decision unit and the components. The split reduces the overhead at the decision unit as it should only concentrate on high-level goals. It does not deal with tasks that can be carried out by the components, such as receiving and analysing information and make low-level decision. The other part responsible of achieving scalability, is the plug-in feature and because components are none-overlapped. Components which are none-overlapped can be added and removed independently of each other.

The performance issues are partly achieved, because the components not only receive environmental information directly from the sensors, but they also analyses them and makes local decisions. This suggests a higher throughput, as opposed to the traditional approach, where a central decision unit must make all decisions based on information from all active sensors.

Because the components receives information directly from the sensors, while managing the responsibility provided from the decision unit, the decision unit has more resources to manages critical high-level goals.

A prototype implementing the matrix architecture style has been made to evaluate the new approach. The evaluation is based on the results of different executions of the system, where components are automatically added and removed. These executions show that under normal circumstances, where components are provided with correct responsibilities, the components consumed a large amount of information. The decision unit were never notified by any of the components. If the components were provided with invalid responsibilities, the amount of notifications to the decision unit grew rapidly.

Even though the matrix architecture style addresses the performance and scalability issues from pervious approaches, it does introduce some new challenges. These challenges were both found during design and implementation, but also during testing of the system.

One of the challenges is to decide when a component should notify the decision unit, if it can’t manage the responsibility provided by the decision unit. Should it be on the first occurrence? And what if it only happens rarely, should it then be based on percentage or ratio or something else? If the decision unit is notified on every occurrence and the component does not stop receiving information from the environment, the component notifies the decision unit with the same frequency as they receive invalid information. This means that the decision unit gets overloaded with notifications, which lowers the scalability and performance of the system. This was what happened, when the components were provided with invalid responsibilities.

In a system like this, where there is communication between the decision unit and the components, it is generally a challenge to minimise the communication. Too much communication degrades the system’s ability to perform, as communication is slower than computation.

The current implementation of the matrix architecture style does have some issues, when components are provided with invalid responsibilities. However, an overall evaluation of the new matrix architecture style, suggest a better performance and scalability.

9 Conclusion

The architecture presented in this thesis, addresses some limitations and challenges associated with the traditional approach of constructing context-aware self-adaptable systems. The main limitation is to be able to consume large amount of diverse data in the environment, which must be processed fast and with accuracy. The challenges are to construct an architecture and feedback loop to support this.

In this thesis, a matrix architecture style has been designed for context-aware self-adaptable systems, which improves performance and scalability of the system. This architecture is constructed around the concept of a matrix. It takes advantages of providing the components of the system to be added and removed dynamically, by using a plug-in mechanism. Furthermore, the responsibility of managing the global system view has been divided between the components and a central decision unit. This division suggests a semi-decentralized feedback loop, as opposed to the traditional central controlled feedback loop.

Scalability is partly achieved by the semi-decentralized feedback loop, which suggests splitting the responsibility between a central decision unit and the components. The split reduces the overhead at the decision unit as it should only concentrate on high-level goals.

The performance issues are partly achieved, because the components not only receive environmental information directly from the sensors, but they also analyses them and makes local decisions. This suggests a higher throughput, as opposed to the traditional approach, where a central decision unit must make all decisions based on information from all active sensors.

A prototype implementing the matrix architecture style has been made to evaluate the new approach. The evaluation is based on the results of different executions of the system, where components are automatically added and removed. These executions show that under normal circumstances, where components are provided with correct responsibilities, the components consumed a large amount of information. The decision unit were never notified by any of the components. If the components were provided with invalid responsibilities, the amount of notifications to the decision unit grew rapidly.

The current implementation of the matrix architecture style does have some issues, when components are provided with invalid responsibilities. However, an overall evaluation of the new matrix architecture style, suggest a better performance and scalability.

Appendix A – An Execution of The System

This appendix shows a five second execution of the system, obtained by writing messages to the console window. Lines that belong together have been colored the same color. And a line number has been added to each line.

Since this system consists of several concurrent threads communicating in an asynchronous manner, the sequence of messages written in the console, may be misleading to the actual executing of the system.

The most interesting track to follow is the one with the **black bold** color:

- Line 1 – 5: This is what is explained in section 6.3.2 “*creating services and tasks*”
- Line 18 – 19 (and 11): This is explained in section 6.3.3 “*Tasks communicating with the Decision Unit*”
- Line 30 and 35: Services are removed before tasks. If they were removed the other way around, then services would try to send messages to a non-existing queue. This would cause an exception.

1 Added serviced: 0

2 A task is needed for service id 0 with name 'Camera'

3 A task is needed for service id 0 with name ' Camera'

4 Decision Unit from Task Id: 0, state: Configured, message: None, value: -1, name: CameraTask

5 Added task id: 0, minValue: 1, maxValue: 9, name: CameraTask

6 A task is needed for service id 1 with name 'Accelerometer'

7 Added serviceId: 1

8 A task is needed for service id 1 with name 'Accelerometer'

9 Decision Unit from Task Id: 1, state: Configured, message: None, value: -1, name: AccelerometerTask

10 Added task id: 1, minValue: 1, maxValue: 9, name: AccelerometerTask

11 Decision Unit from Task Id: 0, state: Activated, message: SensorValueOutOfRange, value: 13, name: CameraTask

- has received invalid value: 13

12 A task is needed for service id 2 with name 'Accelerometer'

13 A task is needed for service id 2 with name 'Accelerometer'

14 Added serviceId: 2

15 A task is needed for service id 2 with name 'Accelerometer'

16 Decision Unit from Task Id: 2, state: Configured, message: None, value: -1, name: AccelerometerTask

17 Added task id: 2, minValue: 1, maxValue: 9, name: AccelerometerTask

18 Decision Unit from Task Id: 0, state: Activated, message: SensorValueOutOfRange, value: 13, name: CameraTask

- has received invalid value: 13

```
19 Task id 0 - Notification from Desicion Unit - message type: AdjustValueRange, minValue: 13, maxValue: 13, Task: Decision Unit
20 A task is needed for service id 3 with name 'Accelerometer'
21 A task is needed for service id 3 with name 'Accelerometer'
22 Added serviceId: 3
23 Decision Unit from Task Id: 3, state: Configured, message: None, value: -1, name: AccelerometerTask
24 Added task id: 3, minValue: 1, maxValue: 9, name: AccelerometerTask
25 A task is needed for service id 4 with name 'Camera'
26 Added serviceId: 4
27 A task is needed for service id 4 with name 'Camera'
28 Decision Unit from Task Id: 4, state: Configured, message: None, value: -1, name: CameraTask
29 Added task id: 4, minValue: 1, maxValue: 9, name: CameraTask

30 removed service id: 0
31 removed service id: 1
32 removed service id: 2
33 removed service id: 3
34 removed service id: 4
35 Removed task id: 0
36 Removed task id: 1
37 Removed task id: 2
38 Removed task id: 3
39 Removed task id: 4
```

Appendix B – Evaluation Data

This appendix shows different executions of the system, to measure the communication between the components of the system. In all executions, random tasks and services are added and removed throughout a single execution. Each execution runs for 20 sec.

Execution 1

Purpose: Normal execution to be used as a reference. All tasks are created with a correct data range. Below is sample data from one execution.

```
Creating service id 0, type: 2
Creating service id 1, type: 1
Created task id 2 to service id 0
Creating service id 2, type: 3
Creating service id 3, type: 1
Creating service id 4, type: 2
Created task id 5 to service id 1
Creating service id 5, type: 1
Removed service id: 3
Removed service id 3
Removed task id: 2
Removed task id 2
Service id 6 not found
Creating service id 6, type: 2
Created task id 8 to service id 2
Creating service id 7, type: 2
Creating service id 8, type: 1
Creating service id 9, type: 2
Removed service id: 4
Removed service id 4
Removed task id: 8
Removed task id 8
Created task id 13 to service id 1
Creating service id 10, type: 2
Creating service id 11, type: 2
Creating service id 12, type: 3
Created task id 16 to service id 2
Service id 13 not found
Creating service id 13, type: 1
Removed service id: 8
Removed service id 8
Removed task id: 13
Removed task id 13
Creating service id 14, type: 2
Created task id 19 to service id 6
Creating service id 15, type: 2
Creating service id 16, type: 2
Creating service id 17, type: 2
Removed service id: 9
Removed service id 9
Removed task id: 16
Removed task id 16
Created task id 24 to service id 12
Creating service id 18, type: 3
Creating service id 19, type: 1
```
Continue on next column →

```
Created task id 27 to service id 6
Creating service id 20, type: 1
Creating service id 21, type: 3
Removed service id: 10
Removed service id 10
Removed task id: 19
Removed task id 19
Creating service id 22, type: 2
Created task id 30 to service id 15
Creating service id 23, type: 3
Creating service id 24, type: 3
Created task id 34 to service id 14
Creating service id 25, type: 1
Removed service id: 22
Removed service id 22
Service id 26 not found
Creating service id 26, type: 2
Removed task id: 24
Removed task id 24
Created task id 36 to service id 20
Creating service id 27, type: 1
Creating service id 28, type: 3
Created task id 40 to service id 20
Service id 29 not found
Creating service id 29, type: 3
Removed service id: 12
Removed service id 12
Removed task id: 36
Removed task id 36
Creating service id 30, type: 1
Creating service id 31, type: 2
Created task id 42 to service id 28
Creating service id 32, type: 3
Service id 33 not found
Creating service id 33, type: 2
Creating service id 34, type: 2
Removed service id: 23
Removed service id 23
Removed task id: 27
Removed task id 27
Creating service id 35, type: 3
Created task id 47 to service id 16
Creating service id 36, type: 3
Creating service id 37, type: 1
Created task id 52 to service id 35
Service id 38 not found
Creating service id 38, type: 3
Removed service id: 17
Removed service id 17
Removed task id: 42
Removed task id 42
Creating service id 39, type: 3
Created task id 54 to service id 30
Creating service id 40, type: 2
```

Evaluation Data: this table contains sample data from three executions.

	Execution 1	**Execution 2**	**Execution 3**	**Average**
Total calls from services to decision unit	87 *)	88 *)	89 *)	88
Total calls from tasks to decision unit	0	0	0	0
Total calls from decision unit to tasks	0	0	0	0
Total calls from services to tasks	950	995	866	937

*) A service notifies the decision unit, when it is configured and when there is no tasks attached to it.

This is sampled data from one of the executions.

Number of service calls to task *i*	Number of notification calls from the decision unit to task *i*	Number of notification calls from task *i* to the decision unit
task id 0: 24	0	0
task id 1: 50		
task id 2: 1		
task id 3: 16		
task id 5: 47		
task id 6: 1		
task id 7: 49		
task id 9: 23		
task id 10: 23		
task id 11: 3		
task id 12: 2		
task id 13: 2		
task id 14: 4		
task id 15: 22		
task id 16: 1		
task id 17: 4		
task id 18: 43		
task id 19: 3		
task id 20: 21		
task id 21: 21		
task id 22: 21		
task id 23: 9		
task id 24: 2		
task id 25: 13		
task id 26: 42		
task id 27: 7		
task id 28: 42		
task id 29: 13		
task id 30: 18		
task id 32: 2		
task id 33: 12		
task id 34: 16		
task id 35: 39		
task id 36: 2		
task id 37: 37		
task id 38: 18		
task id 39: 12		
task id 40: 33		
task id 41: 38		
task id 42: 1		
task id 43: 18		
task id 44: 11		
task id 45: 12		
task id 46: 18		
task id 47: 12		
task id 48: 18		
task id 49: 12		
task id 50: 12		
task id 51: 39		
task id 52: 12		
task id 53: 12		
task id 54: 29		
task id 55: 19		
task id 56: 12		

Execution 2

Purpose: All tasks are configured with a wrong data range, but connected to the correct service. This requires a task to notify the decision unit. Then the decision unit notifies the task with other minimum and maximum values for the data range. Below is sample data from one execution.

```
Service id 15 not found
Creating service id 15, type: 1
Creating service id 16, type: 2
Removed service id: 10
Removed service id 10
Creating service id 17, type: 3
Removed task id: 10
Removed task id 10
Created task id 21 to service id 14
Creating service id 18, type: 1
Creating service id 19, type: 2
Creating service id 20, type: 1
Service id 21 not found
Creating service id 21, type: 3
Service id 22 not found
Creating service id 22, type: 3
Removed service id: 7
Removed service id 7
Created task id 28 to service id 3
Removed task id: 21
Removed task id 21
Creating service id 23, type: 2
Creating service id 24, type: 2
Created task id 31 to service id 11
Creating service id 25, type: 3
Service id 26 not found
Creating service id 26, type: 1
Removed service id: 19
Removed service id 19
Removed task id: 13
Removed task id 13
Creating service id 27, type: 3
Created task id 35 to service id 21
Creating service id 28, type: 3
Service id 29 not found
Creating service id 29, type: 3
Created task id 39 to service id 26
```

Continue on next column →

```
Removed service id: 21
Removed service id 21
Removed task id: 35
Removed task id 35
Creating service id 30, type: 3
Creating service id 31, type: 1
Creating service id 32, type: 2
Created task id 42 to service id 25
Removed service id: 23
Removed service id 23
Removed task id: 31
Removed task id 31
Service id 33 not found
Creating service id 33, type: 3
Creating service id 34, type: 3
Created task id 46 to service id 12
Creating service id 35, type: 3
Creating service id 36, type: 3
Removed service id: 9
Removed service id 9
Removed task id: 39
Removed task id 39
Creating service id 37, type: 1
Created task id 50 to service id 22
Service id 38 not found
Creating service id 38, type: 1
Creating service id 39, type: 1
Created task id 54 to service id 5
Service id 40 not found
Creating service id 40, type: 2
Removed service id: 15
Removed service id 15
Removed task id: 16
```

Evaluation Data: this table contains sample data from three executions.

	Execution 1	Execution 2	Execution 3	Average
Total calls from services to decision unit	97	90	84	90,33
Total calls from tasks to decision unit	306	238	353	299
Total calls from decision unit to tasks	227	175	289	233,33
Total calls from services to tasks	943	857	949	916,33

This is sampled data from one of the executions.

Number of service calls to task *i*

task id 0: 16
task id 1: 16
task id 4: 24
task id 7: 1
task id 8: 48
task id 9: 15
task id 10: 3
task id 11: 2
task id 13: 16
task id 14: 11
task id 15: 15
task id 16: 9
task id 17: 22
task id 18: 14
task id 19: 14
task id 20: 22
task id 22: 23
task id 23: 22
task id 24: 15
task id 25: 48
task id 26: 2
task id 27: 47
task id 28: 15
task id 29: 3
task id 31: 2
task id 32: 21
task id 33: 14
task id 34: 14
task id 36: 45
task id 37: 14
task id 38: 14
task id 40: 14
task id 41: 45
task id 42: 12
task id 43: 14
task id 44: 22
task id 45: 14
task id 46: 12
task id 47: 14
task id 48: 14
task id 49: 15
task id 50: 10
task id 51: 48
task id 52: 47
task id 53: 47
task id 55: 46
task id 56: 22

Number of notification calls from the decision unit to task *i*

task id 0: 2
task id 1: 2
task id 4: 2
task id 8: 6
task id 9: 1
task id 10: 2
task id 11: 1
task id 13: 8
task id 14: 3
task id 15: 2
task id 16: 4
task id 17: 5
task id 18: 4
task id 19: 3
task id 20: 10
task id 22: 4
task id 23: 10
task id 24: 3
task id 25: 13
task id 26: 1
task id 27: 13
task id 28: 7
task id 29: 2
task id 32: 5
task id 33: 3
task id 34: 5
task id 36: 12
task id 37: 3
task id 38: 4
task id 40: 3
task id 41: 9
task id 42: 3
task id 43: 3
task id 44: 5
task id 45: 4
task id 46: 5
task id 47: 3
task id 48: 2
task id 49: 3
task id 50: 3
task id 51: 10
task id 52: 10
task id 53: 11
task id 55: 10
task id 56: 3

Number of notification calls from task *i* to the decision unit

Task id 0: 10
Task id 1: 3
Task id 4: 9
Task id 7: 1
Task id 8: 7
Task id 9: 4
Task id 10: 3
Task id 11: 2
Task id 13: 9
Task id 14: 5
Task id 15: 3
Task id 16: 5
Task id 17: 6
Task id 18: 6
Task id 19: 7
Task id 20: 11
Task id 22: 5
Task id 23: 11
Task id 24: 5
Task id 25: 14
Task id 26: 2
Task id 27: 14
Task id 28: 8
Task id 29: 3
Task id 31: 2
Task id 32: 6
Task id 33: 5
Task id 34: 7
Task id 36: 13
Task id 37: 4
Task id 38: 5
Task id 40: 5
Task id 41: 10
Task id 42: 5
Task id 43: 5
Task id 44: 8
Task id 45: 5
Task id 46: 6
Task id 47: 5
Task id 48: 4
Task id 49: 5
Task id 50: 4
Task id 51: 11
Task id 52: 11
Task id 53: 12
Task id 55: 11
Task id 56: 4

Execution 3

Purpose: All tasks are configured with a wrong kind of service, but the data range is correct. This requires the task to notify the decision unit. Then the decision unit detaches the wrong service from the task and finds a correct service. The task is then attached to the correct service. This process requires more computation by the decision unit. Below is sample data from one execution.

```
Creating service id 3, type: 2
Removed service id: 1
Removed service id 1
Creating service id 4, type: 1
Created task id 4 to service id 2
Creating service id 5, type: 1
Service id 6 not found
Creating service id 6, type: 3
Created task id 8 to service id 0
Service id 7 not found
Creating service id 7, type: 2
Removed service id: 2
Removed service id 2
Service id 8 not found
Creating service id 8, type: 3
Removed task id: 4
Removed task id 4
Created task id 9 to service id 5
Creating service id 9, type: 2
Creating service id 10, type: 1
Creating service id 11, type: 3
Created task id 14 to service id 6
Removed service id: 4
Removed service id 4
Service id 12 not found
Creating service id 12, type: 1
Removed task id: 2
Removed task id 2
Creating service id 13, type: 2
Creating service id 14, type: 3
Created task id 18 to service id 5
Service id 15 not found
Creating service id 15, type: 2
Removed task id: 9
Removed task id 9
Creating service id 16, type: 2
Created task id 23 to service id 14
Creating service id 17, type: 3
Service id 18 not found
Creating service id 18, type: 2
Removed service id: 15
Removed service id 15
Removed task id: 8
Removed task id 8
Creating service id 19, type: 2
Created task id 26 to service id 6
Creating service id 20, type: 1
Creating service id 21, type: 3
Created task id 31 to service id 10
Service id 22 not found
Creating service id 22, type: 2
Removed service id: 8
Removed service id 8
Removed task id: 31
Removed task id 31
```

Continue on next column →

```
Creating service id 23, type: 1
Creating service id 24, type: 2
Created task id 33 to service id 14
Creating service id 25, type: 2
Service id 26 not found
Creating service id 26, type: 1
Removed service id: 5
Removed service id 5
Removed task id: 18
Removed task id 18
Created task id 37 to service id 14
Creating service id 27, type: 3
Creating service id 28, type: 1
Creating service id 29, type: 1
Created task id 41 to service id 26
Service id 30 not found
Creating service id 30, type: 2
Service id 31 not found
Creating service id 31, type: 2
Removed service id: 11
Removed service id 11
Removed task id: 14
Removed task id 14
Creating service id 32, type: 2
Created task id 43 to service id 16
Creating service id 33, type: 1
Service id 34 not found
Creating service id 34, type: 3
Created task id 48 to service id 30
Service id 35 not found
Creating service id 35, type: 3
Removed service id: 18
Removed service id 18
Removed task id: 26
Removed task id 26
Creating service id 36, type: 2
Created task id 50 to service id 19
Creating service id 37, type: 2
Creating service id 38, type: 3
Service id 39 not found
Creating service id 39, type: 1
Created task id 55 to service id 30
Removed service id: 22
Removed service id 22
Removed task id: 37
Removed task id 37
Creating service id 40, type: 3
```

Evaluation Data: this table contains sample data from three executions.

	Execution 1	**Execution 2**	**Execution 3**	**Average**
Total calls from services to decision unit	96	90	91	92,33
Total calls from tasks to decision unit	1135	957	765	952,33
Total calls from decision unit to tasks	0	0	0	0
Total calls from services to tasks	4980	3272	4019	4090,33

This is sampled data from one of the executions.

Number of service calls to task *i*	Number of notification calls from the decision unit to task *i*	Number of notification calls from task *i* to the decision unit
task id 0: 254	0	Task id 0: 16
task id 2: 12		Task id 2: 3
task id 3: 16		Task id 3: 1
task id 5: 29		Task id 5: 6
task id 6: 191		Task id 6: 23
task id 7: 42		Task id 7: 19
task id 8: 15		Task id 8: 3
task id 9: 8		Task id 9: 5
task id 10: 159		Task id 10: 22
task id 11: 244		Task id 11: 15
task id 12: 162		Task id 12: 24
task id 13: 77		Task id 13: 48
task id 14: 39		Task id 14: 6
task id 15: 17		Task id 15: 2
task id 16: 88		Task id 16: 5
task id 17: 74		Task id 17: 47
task id 18: 16		Task id 18: 11
task id 19: 141		Task id 19: 23
task id 20: 73		Task id 20: 47
task id 21: 246		Task id 21: 15
task id 22: 124		Task id 22: 23
task id 23: 245		Task id 23: 15
task id 25: 244		Task id 25: 15
task id 26: 25		Task id 26: 5
task id 27: 106		Task id 27: 23
task id 28: 52		Task id 28: 6
task id 29: 74		Task id 29: 47
task id 30: 227		Task id 30: 15
task id 32: 77		Task id 32: 47
task id 33: 200		Task id 33: 12
task id 34: 44		Task id 34: 6
task id 35: 90		Task id 35: 23
task id 36: 82		Task id 36: 23
task id 37: 17		Task id 37: 3
task id 38: 79		Task id 38: 46
task id 39: 169		Task id 39: 15
task id 40: 82		Task id 40: 48
task id 41: 76		Task id 41: 44
task id 42: 81		Task id 42: 48
task id 43: 50		Task id 43: 15
task id 44: 65		Task id 44: 22
task id 45: 67		Task id 45: 23
task id 46: 59		Task id 46: 23
task id 47: 81		Task id 47: 48
task id 48: 58		Task id 48: 21
task id 49: 55		Task id 49: 22
task id 50: 40		Task id 50: 15
task id 51: 92		Task id 51: 14
task id 52: 53		Task id 52: 22
task id 53: 88		Task id 53: 14
task id 54: 84		Task id 54: 14
task id 55: 48		Task id 55: 19
task id 56: 76		Task id 56: 44
task id 57: 67		Task id 57: 14

References

[1] Semantic Web ontologies for Ambient Intelligence – runtime monitoring of Semantic Component Constraints
Klaus Marius Hansen, Weishan Zhang, Joao Fernandes and Mads Ingstrup
Department of Computer Science, University of Aarhus

[2] Using Mixin Layers for Context-Aware and Self-Adaptable Systems
Brecht Desmet, Jorge Vallejos Vargas, Stijn Mostinckx and Pascal Costanza
Programming techonology Lab – Vrije Universiteit Brussel

[3] Towards Self-managed Pervasive Middleware using OWL/SWRL ontologies
Weishan Zhang, Klaus Marius Hansen
Department of Computer Science, University of Aarhus

[4] Software Engineering for Self-Adaptive Systems: A Research Roap Map (Draft version)
Betty H.C. Cheng, Rogério de Lemos, HOlger Giese, Paola Inverardi, Jeff Magee
Dagstuhl Seminar Proceedings 08031

[5] Self-Managed Systems: an Architectural Challenge
Jeff Kramer and Jeff Magee
Future of Software Engineering (FOSE'07)

[6] Issues in Developing Context-Aware Computing
Jason Pascoe, Nick Ryan and David Morse
HUC'99, INCS 1707, PP. 208-221

[7] Understanding and using context
Anind K. Dey
Personal and Ubiquitous Computing (2001) 5:4-7

[8] A survey of Context-Aware Mobile Computing Research
Guanling Chen and David Kotz
Department of Computer Science, Dartmouth College

[9] Categorization and Modeling of Quality in Context Information
M.A. Razzaque, Simon Dobson and Paddy Nixon
School of Computer Science and Information
University College, Dublin IE

[10] Using Architecture Models for Runtime Adaptability
Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, ketil Lund, Eli Gjørven
IEEE Software, 2006
0740-7459/06

[11] Implementing Self-Adaptability in Context-Aware Systems
Boris Mejias and jorge Vallejos
Catholique de Louvain University of Belgium
Vrije University of Belgium

[12] Software Architecture in Practice (Second Edition)
Bass et al.
ISBN: 0321154959

[13] Specification of the MADAM Core Architecture and Middleware Services)
MADAM Consortium
http://www.intermedia.uio.no/download/attachments/9675/Deliverable+D2.3+Specification+of+the+MADAM+Core+Architecture+and+Middleware+Services.pdf?version=1
Downloaded the 7th of Marts 2010.

[14] Expanding the Possibilities for Enterprise Computing: Multi-Agent Autonomic Architectures
Gilda pour
San Jose State university

[15] Multiagent Systems and Software Architeture
Proceedings of the Special Track at Net.ObjectDays Erfurt, Germany, September 19, 2006
Danny Weyns and Tom Holvoet (Eds.)

[16] Multiagent Systems
Katia P. Sycara
AI Magazine, Volume 19 No 2, Summer 1998, page 79-92

[17] Applying Multi-agent Concepts to Dynamic Plug-in Architectures
Lawrence Cabac, Michael Duvigneau, Daniel Moldt and Heiko Rölke
University of Hamburg, Department of Computer Science
Agent-Oriented Software Engineering VI, LNCS 3950, pp. 190-204, 2006