university of groningen

faculty of mathematics and natural sciences

Concurrent execution of automatically generated plans in Smart Homes

Master's thesis

August 2011

Student: E. Lazovik

Primary supervisor: Prof. dr. ir. M. Aiello

Secondary supervisor: Prof. dr. ir. P. Avgeriou

ABSTRACT

An application area where heterogeneity is a norm is that of pervasive systems, where thousands of autonomous heterogeneous devices live together and need to interoperate. In particular, domotics is concerned with technology that pervades the home in order to make it more pro-active and aware with the final goal of increasing security and comfort of its inhabitants.

This thesis focuses on Smart Homes, that is, homes that contain heterogeneous interactive and pro-active devices, that adapt their behavior to the needs of the home inhabitant through extensive interoperation and user interaction. For example, a movie may be automatically paused when the user leaves the room, and then launched again when s/he is back; windows are automatically opened to regulate the air condition or as a reaction to gas leak, and so on. One of the greatest challenges in developing middleware for smart homes is that this kind of systems are in need for the orchestration of the heterogeneous services within complex environment. Therefore, a special orchestration engine is needed to be implemented in order to provide efficient concurrent execution of complex scenarios. For every scenario from different users a special plan is generated automatically. These plans should be executed concurrently in the most efficient way.

The thesis is dedicated to the implementation of the orchestration engine capable of controlling concurrent execution of automatically generated plans for complex scenarios within Smart Homes and discusses the advantages of the chosen approach.

CONTENTS

1 INTRODUCTION 1
2 RELATED WORK 3
  2.1 Projects in the field of Smart Homes 3
  2.2 Concurrent Execution in Pervasive Systems 6
3 SM4ALL PROJECT 9
  3.1 Architecture of SM4ALL 11
4 ORCHESTRATION MODEL FOR THE SM4ALL PROJECT 15
  4.1 Responsibilities 15
  4.2 Dependencies 15
  4.3 Formal orchestration model 16
5 ORCHESTRATOR IMPLEMENTATION 21
  5.1 Event-Based Actor Scheme 21
  5.2 Process model 22
  5.3 Basic Components 22
  5.4 Communication with Environment 29
6 THE SCENARIOS AND ORCHESTRATOR RUNNING 31
7 TESTING AND EVALUATION OF THE SOLUTION 37
  7.1 Critical section test 37
  7.2 Dining Philosophers test 38
  7.3 The Evaluation of the Solution 41
8 CONCLUSIONS AND DISCUSSIONS 47
A APPENDIX 49
B APPENDIX 53
C APPENDIX C 59

LIST OF FIGURES

Figure 1 Global view of the system and interaction with environment. 9
Figure 2 Architecture of the SM4ALL platform [21]. 12
Figure 3 Planner within the system. 14
Figure 4 Execution of the plans locking independent resources. 32
Figure 5 Execution of the plans locking one type of the unique resource. 33
Figure 6 Execution of the plans locking the same type of the unique resource for step 2. 34
Figure 7 Execution of the plans locking the unique resources in reversed order. 35
Figure 8 Maximum number of conflicts depending on the number of working threads. 42
Figure 9 Maximum number of locks conflicts depending on the number of working threads. 43
Figure 10 Percentage of CPU use depending on the number of working threads. 44
Figure 11 Percentage of CPU use depending on the number of working threads through Windows panel. 44
Figure 12 Total memory used for execution depending on the number of working threads. 45
Figure 13 Execution time depending on the number of concurrent threads. 45

LIST OF TABLES

Table 1 Maximum number of conflicts depending on the number of working threads. 41
Table 2 Maximum number of the concurrent conflicts depending on the number of working threads. 42
Table 3 Maximum number of the concurrent conflicts depending on the number of working threads (in %). 43
Table 4 Total memory used depending on the number of working threads. 43

Table 5 Time of execution depending on the number of working threads. 45

1 INTRODUCTION

Due to the ever increasing availability of cheap sensors and actuators, homes are becoming more technological [9]. This goes well beyond the 'gadgetification' of the house of the early adopters and wealthy families, as domotic solutions are becoming massively accessible for creating more secure and comfortable living spaces. Such trend is welcomed by the general public in as much as by the people who have special needs such as limited mobility or invalidity. The current shift does not simply provide for homes with hundreds of sensors and actuators, but also for a different way of controlling the home and even interacting with it. If in the past one had direct command-effect interactions or, at most, simple feedback loops, now we are going towards smart pro-active homes [8]. The idea is that quality of life will improve when not only we have means to mechanize domestic activities, but when we also are immersed in an environment that is aware of its inhabitants, of the user activities and adapts itself to best support us. This can be done by resorting to software solutions and embedded systems specifically tailored for home deployment. The software is mostly responsible of coordinating devices and actuators that are dynamically available in the home, while coupling the coordination with sensing activities.

In the following chapters the presented work focuses on dealing with the issues of concurrent execution of the automatically generated plans for domotics. This work is carried out as a part of the EU STREP Project FP7-224332 Smart Homes for All [21]. Its main contribution to the European Project is providing a solution for an orchestration engine for the heterogeneous devices in Smart Home environments. A mere fact of having many technological devices at home does not still makes the house "smart". Only by means of special middleware capable of the composition of the services and turning the devices' actions into complex plans can we achieve a really smart environment able to support us in every ady life. However, some of the devices or service are of limited number within the house. When more than one person wants to achieve his/her goal and the different plans involve the same resource, some measures should be taken in order to resolve possible plans conflicts and allow the plans smoothly proceed. Therefore, there is a need of a special orchestration engine which can execute complex plans simultaneously even facing the limited resources conditions. Thus, orchestrator is one of the most important parts of the system. Therefore, the main objective of the presented work is to provide a valid solution to the problem of concurrent execution of complex plans. Additionally, the investigation how the Actors scheme can be used to incorporate event-based scenarios into the smart home is conducted. Finally, a prototype of the orchestration engine is developed.

In Chapter 2 we present a related work in the field of Smart Home environment and concurrent execution of the programs in pervasive systems. There is some research performed in these and close areas.

Chapter 3 is dedicated to the general overview of the SM4ALL project and to its architecture.

As the orchestration engine is a part of the SM4LL middleware and takes responsibility for execution of complex plans, the Chapter 4 provides a general survey on the Orchestrator module including its responsibilities and dependencies from other modules of the middleware with pointing out the formal model for the orchestration of the plans.

Chapter 5 demonstrates the basic components of the orhcestration engine in action and provides explanation to the particular details of the concurrent algorithm for the resources of limited number. The issues of coordination of work between the different executor machines and communication with the environment are also covered in this Chapter.

The correct solution for the orchestration engine leads to the impossibility of having the deadlocks and race conditions among the concurrently executed plans. Chapter 6 focuses on possible scenarios for the orchestrator concerning the limited type of the resources and demonstrates that the system is able to avoid the obstacles.

To verify and validate the system a special evaluation could be performed through the testing experiments. Chapter 7 shows the results of the orchestration testing and provides an explanation for the behavior of the system using the number of concurrent threads as a basic metric.

Finally, Chapter 8 represents the main conclusions for this work and proposes the directions for future development in the area of concurrent execution within Smart Homes environment.

2 RELATED WORK

The related work could be approached from two different points. There are some projects that are similar to the SM4ALL. The orchestration of the services is at the center of such projects because without the composition it is impossible to use web services simultaneously.

At the same time pervasive systems are much more elaborated in order to compose the services provided by the physical devices for their concurrent execution. However, such solutions almost never were applied to the Smart Homes environments.

Therefore, it is necessary to take into consideration the development in the both areas.

2.1 PROJECTS IN THE FIELD OF SMART HOMES

Nowadays, the Internet has become an indispensable part of our lives, especially in developed countries. A vast number of daily services have found a counterpart on the Internet. For example, activities such as shopping, reading newspapers, booking hotels are available online and people actively use them. Such services can also be accessed by other services, not only directly by humans. This makes it possible to create services which are actually nothing more than a combination of other ones, e.g., travel portals use a number of finer-grained services, such as hotel and flight booking, car rentals, etc.

These scenarios are not possible to realize unless standardized protocols are developed. Protocols are often presented as stacks of related protocols taking care of different aspects, in the spirit of the famous ISO/OSI Networking stack reference model. For Web interoperation, there are a number of such protocol stacks. The most famous ones are REST [14] and Web services [20, 12]. These protocol stacks are usually formed by several layers, starting from low-level transport protocols, e.g., SOAP [24], up to complex composition [13] and transaction [27] languages. The newest development of such protocols is a JSON (JavaScript Object Notation) wich is a lightweight data-interchange format[17]. JSON is a text format that is completely language independent what expands its application to many different domains.

The emerging success of service-oriented computing brings back services from Internet to real life. Everyday devices, such as mobile phone and media players, and even fridges and TV become smarter and smarter, and often provide their functionalities in form of embedded services. These services can be accessed through standardized API, e.g., web services.

Nowadays, the web services architecture is widely used as an architecture for creating different distributed computer systems. There are some initiatives to present different frameworks for the orchestration of web services.

One of the projects that concentrates on Smart Homes is a Duke Smart Home Program which is the project of the Pratt school of Engineering[2]. The goal of the Duke Smart Home Program is to offer a research and educational program that emphasizes energy efficient, sustainable and 'smarter' living. Smarter living is defined as using technology for automation in a way that encourages behavior the users want and need to achieve the values of energy efficient, sustainable living. The program operates and manages The Home Depot Smart Home as an evolving resource purposefully used to inform our ideas about sustainable, energy efficient, smart living. Many students have composed the teams and developed different projects for use within Smart Homes. The majory of the projects focuses on tracking people or transport movement in stdent campus by using different types of sensors. All the projects do not deal with the very complex scenarios, and, therefore, a little interest was paid to the orchestration of the services. However, this question will arise for them in the future as they plan to merge some projects together to provide better services using orchestration concept. Respect to this project SM4ALL is in advantage because it already tackles complex scenarios.

One of the main objective of Smart Homes is to help inhabitants with the different types of disabilities. A special organization is created to help blind and partially sighted people as a part of Tiresias organization[1] wich is an assosiation of many hospitals and some IT-companies as a special organization in the United Kingom. The Digital Accessibility Team (DAT) is involved in influencing the research and development of new technologies for the future benefit of blind and partially sighted people and in providing standards and research to improve the design of equipment to ensure it is accessible by people with sight problems. DAT is continuing its work in the areas of Human Computer Interaction (HCI) by participating in a number of British Standards Institute (BSI) groups including ICT/6 ICT Accessibility Coordination Panel, IST/45 Web Accessibility Committee and PH/9 Applied Ergonomics Committee. Along with this, DAT is also involved in a number of research initiatives and other projects in these areas. One of their projects is "Ambient Intelligence System of Agents for Knowledge-Based and Integrated Services for Mobility Impaired Users (ASK-IT)"[3]. The research was expected to be conducted the orchestration of the services for the blind and partially sighted people. However, the project focused more on sensor activities and dealing with them independently. For such kind of project the orchestration is not needed. A simple job scheduler can deal with that type of work. However, nowadays people want not only to be able to control the devices independently, but to compose the services into complex plans with the ultimate goals to satisfy their needs. That requires a smart planner module together with the orchestration engine.

Another project for support of the independent living of ageing people is Netcarity [6]. Netcarity is a European project researching and testing technologies which will help older people to improve their wellbeing, independence, safety and health at home. The project is investigating how new and existing technologies can be integrated cost effectively into people's homes, making them feel more comfortable about remaining in this familiar environment. It is developing and testing a new technology infrastructure for homes, with systems that enhance communication with friends, family and care givers; support everyday living and promote a sense of social inclusion. It will encourage older people to live independently and inspire them to be more socially active. Netcarity's goal is to turn older peoples' homes into supportive environments which include them in society and postpone or avoid the expensive and traumatic move into care homes. There are some products already ready to use from this project. Howeverer, thr most of them is concentrated on the home automation without further integration of services. In other cases when the integration is provided it is assumed that the person lives alone, and therefore, all the requests from the inhabitant are executed in a sequential order without bothering about concurrent requests for the same resources.

One of the close to the SM4ALL project in terms of orchestration and distributed communication is the project DAFFIE of the Boston university [4]. DAFFIE is a software package and network architecture for building distributed/collaborative tele-immersive environments from pre-existing data which supports dynamic behaviors through the use of networked agents. DAFFIE provides participant representations by avatars, multiple virtual spaces, animated models, 3D localized sound using 2-8 audio channels, and network-based telephony. It currently runs on SGI workstations and supports ImmersaDesks, CAVEs, SGI graphics workstations, and head-mounted displays. DAFFIE is being ported to run on a number of other Unix and NT platforms. DAFFIE is designed to run on high-speed, low-latency networks, such as the vBNS and Internet. A running DAFFIE world includes an event server, immersive viewer clients, sound (e.g. telephony) generator and sound player clients. Additional sites may participate using viewer clients, sound clients, and autonomous agents, joining or leaving at any time. Here it is possible to see the network of agents that are connected with each other. This project is close to SM4ALL project in the terms of network agents communicating with other agents by sending messages. However, there is also a big difference for orchestration of the provided services. It does not include any web service and any other interaction between different participants except their messages through special hardware. Every request to the system is independent from the others. The requested resources could be limited, but if the user does not receive the resource, the next request for this resource is considered to be new. There is no history for the requests, and the orchestration of the request is off the picture. There some ideas that may be useful from that world including interaction between avatars, the sound system and the system of messaging. As the objectives of the system are different from domotics, the problems of embedded web services within smart homes are on the backstage of this system.

Some other projects could be found where Smart Homes are involved through Internet. Nowadays, there is a huge amount od projects of such type. However, any of them could provide a reliable and effective solution for the concurrent execution of the complex plans, especially with the existing of limited types of resources. Smart Homes For All European project [21] is the first project tackling this issue. The presented work is part of this project, therefore, the prject is described further in the work.

2.2 CONCURRENT EXECUTION IN PERVASIVE SYSTEMS

There are some products which support concurrent execution of different processes that are presented at the market nowadays. They propose solution for the different users to launch their processes simultaneously.

An example of such products is Pervasive SQL Client Running on Citrix Server[23]. This solution assumes that there are pervasive clients that could simultaneously access database via SQL interface. This solution is good for the enterprises with the centralized server and database. However, in distributed system envorionment we deal with the possibly huge amount of devices and servers. That means, that Citrix server solution could not be completely adapted in Smart Homes.

Another example of the product currently in market is Btrieve[22]. Btrieve is a transactional database (navigational database) software product. It is based on Indexed Sequential Access Method (ISAM), which is a way of storing data for fast retrieval. It is scalable solution, however, it is again developed for the fast retrieval of the data from database. In Smart Homes we have many devices that want to write their information to one of the databases simultaneously. It is much more likely, that we have more writes than reads. Therefore, this solution is still not completely suitable for solving the problem of concurrent execution of complex scenarios in Smart Homes.

Some research has been done by different pervasive communities in order to solve the task of concurrent execution in ever changing environment. One the teqniques is static slicing for pervasive programs[15]. It helps to achieve the clearer definition of the states of the system considering the state of the every device in the environment. However, it does not completely solve the problem of concurrent execution of the processes in the Smart Homes middleware. The focus of the article is how to capture the state of the system to make the decisions for plan. The execution itself is not taken into consideration.

One of the main challenges imposed on context consistency checking by asynchronous environments is how to interpret and detect concurrent events. To this end, Nanjing pervasive community proposed the paper about the Concurrent Events Detection for Asynchronous consistency checking (CEDA) algorithm[16]. An analytical model, together with corresponding numerical results, is derived to study the performance of CEDA. They also conducted extensive experimental evaluation to investigate whether CEDA is desirable for context-aware applications. Both theoretical analysis and experimental evaluation show that CEDA accurately detects concurrent events in time in asynchronous pervasive computing environments, even with dynamic changes in message delay, duration of events and error rate of context collection. It is a very interesting work, however, it does not deal with the execution of concurrent processes in conditions of having limited amount of resources.

There are some other research activities in area of pervasive systems, however, almost all of them are focused on fast access to centralized databases or on context consistency of the changing pervasive environment. There is not any research done in sphere of concurrent processes dealing with the limited number of the resources. That is why this work proposes a solution to this problem.

3 SM4ALL PROJECT

The European project SM4ALL - Smart Homes For All is the project where an embedded middleware platform for pervasive and immersive environments is being developed. This innovative middleware platform is designed for the interaction of smart embedded devices within the home environments as it is presented in Figure 1.

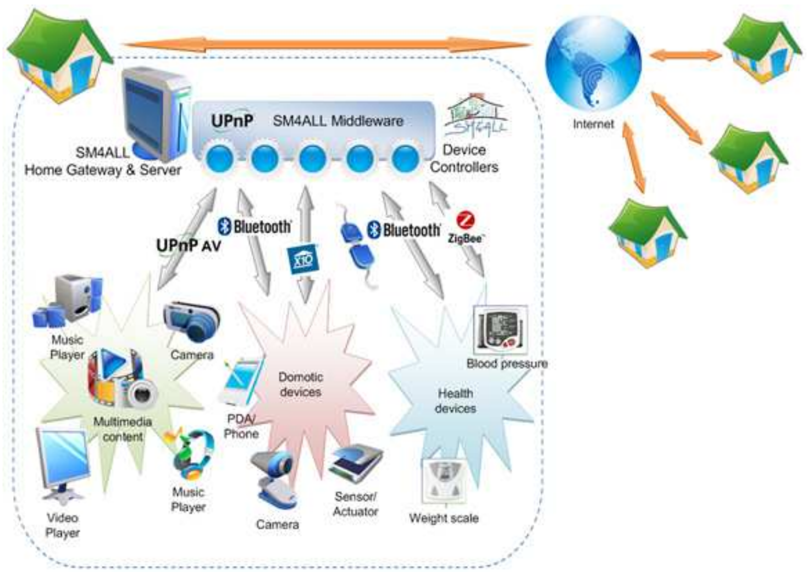

Figure 1: Global view of the system and interaction with environment.

Embedded systems are specialized computers used in machines and other equipment as the controllers of the behaviour of that machines. Within the framework of this European project every embedded program represents web service interacting with human user through the special interface. From the point of view of web service that interface is the client who can ask for some actions to be performed.

The way of interaction is presented through compositional and semantics techniques for dynamic services reconfiguration. The platform is largely scalable and robust because of the use of Peer-to-Peer technologies. At the same time it preserves person privacy and enhances the security of the whole environment. This platform is applied to the homes of the persons with different capabilities: from young healthy people to the aged and disabled.

The project is aimed to integrate devices to simplify the access to the services provided by these devices and to compose web services in order to give a simple access to complex functionalities for the users within smart home environment.

The input for the platform is expected from the user and the response to the input is the change in home environment. Some actions will be performed to achieve the most common task which will be taken from user input. As an example, one could imagine that user wants to take a shower. The response from the system will be warming of the bathroom, turning on the water and warming it.

All favourite user values for the temperature, gas and other devices can be stored at the User Preferences. User Preferences is a database containing pairs of key-value, where key is the name of the preference (e.g., preferred water temperature) and the value is what user likes.

Some functions should be executed by the system on the permanent basis without the request of the user. This is concerned to the temperature inside the house, monitoring of the possible gas leakage. System is responding automatically in case of lowering the home temperature or of gas leakage. It is performing some actions to resolve the problem. In case of gas leakage it opens the window immediately, then presents the notification on the screen for user, etc. In case of lowering temperature it automatically turns on boiler to warm the house to reach the preferred temperature. As these actions are independent from the user personality, another database is considered which is called Rule Preferences. User can insert preferred values for such permanent actions and monitoring. For example, s/he can add preferred permanent value for the temperature of the house.

This way there are two different databases for the preferences of the user. One of them contains the preferences of the concrete user concerning her/his special needs. And the second database contains information about the preferred values of users for monitoring the home environment on the permanent basis.

As it has been said before, the system is designed for the wide range of users. Therefore, it will have many different interfaces for the users of different capabilities. In one case it can be device with the keyboard to print the input, in other case it can be device which recognizes voice commands of the user.

The house is the dynamic environment as people like to change the position of furniture and devices through the time. System is designed to support such changes as well as discover new devices with the functionality of web services. In order to reach these goals the system needs to be largely scalable.

Context awareness is another important feature for the system of such type. People are not typically at the same place inside house for a long time during the day. System executes some activities on permanent basis, and if the location of the person is unknown there is a possibility to execute some action which will be damaging to person. For example, person is near the window and the system decides to open the window to get fresh air for the room. The person can be traumatized by opening window. To avoid such damaging coincidences the system is being designed to take into account the context awareness.

Security of the system and privacy of the person are key requirements to be dealt with. Nobody wants their neighbour to know the daily activities inside the home. Therefore, the platform is being designed in order to deny any unauthorized access to the system data.

The middleware needs to be tested, validated and verified. However, to have a real house with the installed devices, huge amount of sensors and a middleware is very expensive. For this reason a special visualization and simulation environment (ViSi)[19, 7] was developed by using the Google SketchUp tool[5].

3.1 ARCHITECTURE OF SM4ALL

Different technologies of wide range of devices are expected to be part of the SM4All architecture and interoperate through the SM4All middleware. Due to this fact there is an abstract communication layer supporting standard Peer-to-Peer as one component of the global platform architecture. This component is called Peer-to-Peer Home Gateway. The gateway is an interface between pervasive layer and platform middleware. The UPnP (Peer-to-Peer) approach to the system has many advantages:

- compatibility with many communication standards;
- discovery of the new devices having web service functionalities;
- management of failures of the smart home web services;
- simple way to remove unnecessary devices.

As it can be seen in Figure 2 the middleware contains a set of logical components distributed among three different layers:

- User layer;
- Composition layer;
- Pervasive layer.

3.1.0.1 *User layer*

User layer is the interaction layer. It is dedicated to the interaction of the system with different types of users. Due to the different communication technologies it is realized through different User Interfaces. From the point of global view one can see this layer as one logical component: User Interface. It is the component built to get user input which will become the service invocation at the level of lower layers.

Such interfaces will be used by users to establish new goal to achieve, to set their individual preferences or to put preferable rules for the automatic monitoring and control of the home on the permanent basis.

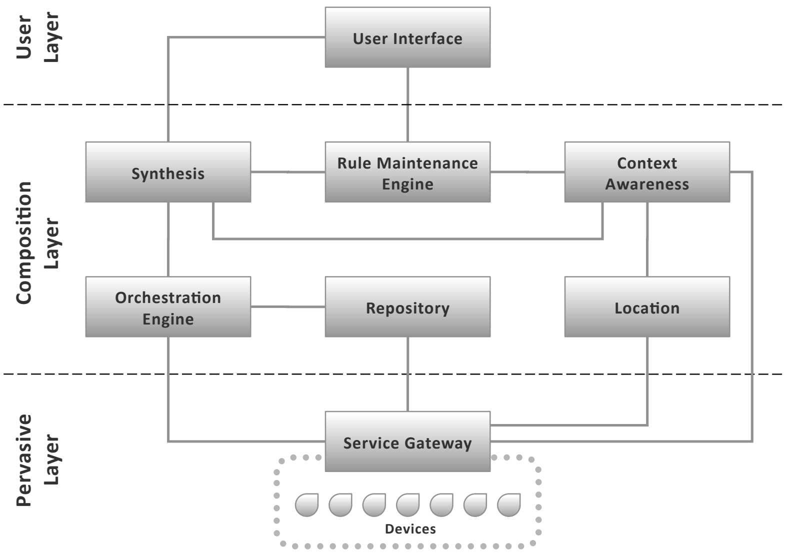

Figure 2: Architecture of the SM4ALL platform [21].

3.1.0.2 *Composition layer*

The main goal of the composition layer of the middleware is to receive user input on the high level language from user interface, to fulfill goals to achieve and to control the execution of the services provided by smart home devices deployed within the middleware platform.

Following logical elements constitute the composition layer:

- Repository. It is a general repository for descriptors of services, different ontologies and other types of data;

- Synthesis. This logical component receives user input from the upper User layer or from the Rule Maintenance Engine and composes concrete Plans. Synthesis is purposed to translate a high-level complex goal into the sequence of more simple actions that can be assigned to different devices having corresponding web service functionalities. The translation of the high-level goal is conducted according to the information from the Context Awareness logical component;

- Orchestration Engine. This engine receives the Plan from the Synthesis logical component and constructs the set of the web services available from devices deployed within the middleware platform. Thus, Plan can be executed with actual services. The orchestration is executed while interacting with the Repository data containing web services descriptors.

- Rule Maintenance Engine. This engine is constructed in order to maintain automatic actions of the system inside smart home environment. It activates some functionalities when special determined conditions are hold. It means that Rules are activated depending on the Conditions. The Conditions are inserted through User layer as Rule Preferences for smart home. The Plans constructing depends on the Context Awareness and Location logical components.

- Context Awareness. This component collects data from devices, processes and stores the resulted values in order to provide up-to-date information about real environment and current status of the system. These results represent the context for all layers of the middleware. User preferences and Rule preferences are the parts of the context too.

- Location. This component is constructed to contain an important information for the whole context. It contains locations of the objects and people inside the smart home environment where middleware is running. Specific logical component was elaborated for such kind of information because complex mechanisms are involved in calculating the locations of people and objects within the home.

Logical components Context Awareness and Location are permanently interacting with Pervasive layer in order to receive up-to-date data from physical devices with embedded web services systems.

To deal with complex scenarios within smart home a planning system is a necessary element of the Composition layer. It synthesizes plans on-the-fly based on goals given by home inhabitants[18].

The goal is specified through one of the possible interfaces, be it brain-computing interface, mobile device, voice-recognition, or, possibly, other software. Given a goal, the planner collects the information about the current state of the house, e.g., available services and their current states, through the context module, which may possibly prefetch the data for better performance.

Synthesized plan is then given to the orchestration component (executor) which is responsible for a plan execution. It also includes simple reasoning capabilities for simple failure recovery and service instantiation. To execute a particular action, the orchestration component finds one of the possible services that implements the desired action. Some extra constraints may be associated in this case to reduce possible instantiations. For example, alarm action may instantiate corresponding implementation which is close to the user, e.g., by showing a message on TV screen if the user is watching it, or by invoking alarm in the alarm clock if the user is sleeping in his bed.

User himself is represented as one of the services at the pervasive layer. That is, whenever interaction with the user is needed according to a plan, orchestration component simply invokes one of the services that represent the user.

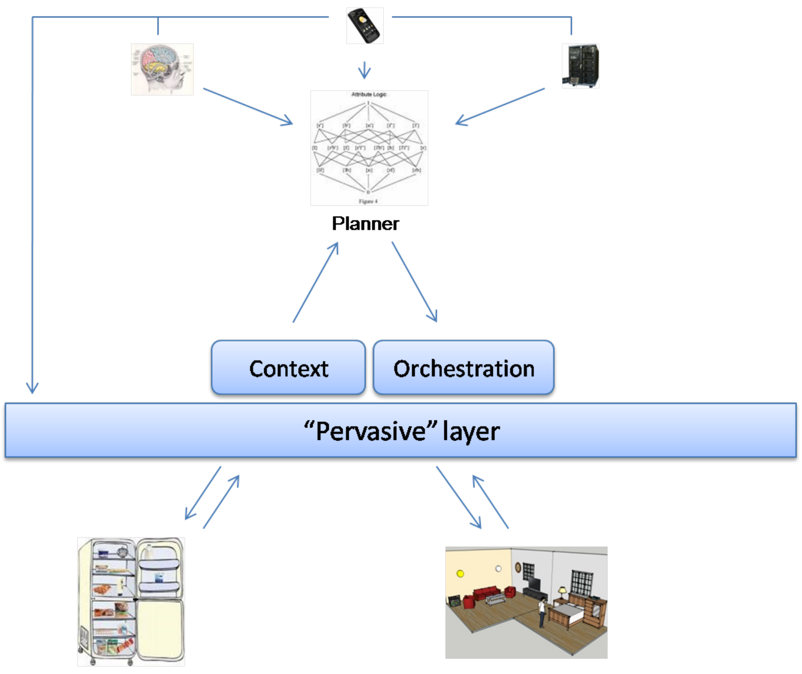

Figure 3: Planner within the system.

Figure 3 presents a general view of the system and a place and role of the planner and orchestration engine in it.

3.1.0.3 *Pervasive layer*

Pervasive layer of the middleware is a physical layer of the system which is represented by single logical component: Service Gateway.

Service Gateway is a component which lets physical devices to interact with the other layers components by presenting the descriptors of the embedded web services, executing service instances and working as a specific middleware between user communication devices and the platform, etc. The component can be viewed as an abstract wrapper for all devices which are presented in house.

4 ORCHESTRATION MODEL FOR THE SM4ALL PROJECT

The orchestrator is an independent component of the infrastructure of the SM4ALL project. It is a module that receives the plans to execute and decides on what to do in order to execute them in an efficient way. Since the current state of the environment constantly changes, and these changes may interfere with the process in execution, the orchestrator should be able to receive feedback about the status of the system to drive the execution of each invocation accordingly.

4.1 RESPONSIBILITIES

The main responsibility of an orchestrator is to execute of the incoming plans and user commands. The devices are invoked when there is a need in their performance. Every plan should be executed in an efficient way. There could be many plans in a simultaneous execution. Every user can have more than one plan in the process of execution or many user can require to execute their plans at the same time. Every plan is executed independently. However, there may be plans which use the same resources from the same location. Therefore, an orchestrator is responsible for checking whether the plans can be executed together without intersection. If the plans share one or more steps concerning different users, it is the responsibility of an orchestration engine to resolve the conflict. Therefore, an orchestration engine is responsible for:

- execution of the plans;
- effective execution of the user commands;
- simultaneous execution of the plans for the same or different users;
- dealing with the conflicts concerning limited resources.

4.2 DEPENDENCIES

An orchestrator engine is dependent from the following modules:

- Context Awareness;
- Repository;
- Planner;

- Pervasive layer interface for the physical devices.

Context Awareness module, as it was described before, is the module responsible for the constant monitoring of the Smart Home environment and updating the information about the changes in physical devices, services, locations. This information is very important for the orchestration engine, since the orchestrator needs to know the current state of the environment in order to execute plans efficiently. Repository contains the list of all resources which can be in use the execution of plans. When the plan arrives, orchestrator should check whether it is possible to execute the plan with the resources available from the Repository. Since the list of the resources could be changed, an orchestrator should communicate with the Repository for every plan in motion. Planner is a module that creates plans to execute. An orchestration engine needs plans from the planner to start their execution. If the plan cannot be executed for some reason (for example, if there are no resources available), an orchestrator notifies Planner about an issue. In that case, the plan should be revised by Planner. The Pervasive layer interface is needed for the retrieving the needed physical devices, for the communication and controlling them. An orchestrator needs to send commands and receive the responses through the special gateway for the physical devices.

4.3 FORMAL ORCHESTRATION MODEL

The formal orchestration model provides the formal description of the plans incoming from planner, the problems that an orchestrator is facing and model of the orchestration engine.

4.3.0.4 *Formal statements for the plans*

Each action represents basic atomic operations that are supported by the target environment. For example, in the context of smart homes, some possible actions are *turnOnLight, turnOffLight, pauseTv, ...*

Definition 1 (Action). *An action* a *is defined by a tuple* $\langle p, e \rangle$*, where*

- $p : \mathcal{D} \rightarrow \{\top, \bot\}$ *is a* precondition *function that for each possible global variable assignment it defines whether the action is applicable or not.*
- $e : \mathcal{D} \rightarrow \mathcal{D}$ *is an* effect *function, that for each global variable assignment it defines a new value if the action is executed.*

To distinguish between precondition and effect functions for different actions, we write $a.p$ *and* $a.e$ *to explicitly refer that* p *and* e *belong to action* a*.*

Therefore, each action is described in terms of preconditions and postconditions (effects).

Definition 2 (Planning Domain). *Planning domain $\mathcal{P}$ is defined by a tuple $\langle \mathcal{V}, \mathcal{A}, d_0 \rangle$, where:*

- $\mathcal{V} = \{v_1, \ldots, v_n\}$ *is a set of variables over arbitrary domains* $\mathcal{D} = \mathcal{D}_0 \times \ldots \times \mathcal{D}_n$;
- $\mathcal{A}$ *is a set of actions;*
- $d_0 \in \mathcal{D}$ *is an initial state (variable assignment, or context).*

Definition 3 (Plan). *Plan π for a planning domain $\mathcal{P}$ is defined as a sequence of actions $a_0, \ldots, a_n$, such that $\forall a \in \pi : a \in \mathcal{A}$. Plan is* valid *for $d \in \mathcal{D}$ if :*

- $\forall i : a_i.p(a_{i-1}.e(a_{i-2}.e(\ldots a_0.e(d)))) = \top$, *that is, each action is applicable at each step of the plan.*

Plan π is valid for $\mathcal{P}$ if it is valid for the initial state d_0 of $\mathcal{P}$.

Definition 4 (Reduced Planning Domain). *Reduced planning domain with respect to a plan π is a tuple $\mathcal{P}_\pi = \langle \mathcal{V}_\pi, \mathcal{A}_\pi, d_\pi \rangle$, where*

- $a \in \pi \Leftrightarrow a \in \mathcal{A}_\pi$;
- *if variable v is affected by plan π then it is in $\mathcal{V}_\pi$;*
- d_π *is a projection of* d *to* $\mathcal{V}_\pi$.

Definition 5 (Independent Plans). *Two plans π_1 and π_2 are independent if their reduced planning domains do not overlap, that is:*

- $\mathcal{V}_{\pi_1} \cap \mathcal{V}_{\pi_2} = \emptyset$.

Otherwise, plans are non-independent.

Definition 6 (Instantaneous Plan). *A plan π of length n is* fast *iff $\forall i = 0..n-1 : a_i$ is instantaneous.*

Note that the last action of an instantaneous plan is not required to be instantaneous by itself. That also mean, that each plan may be represented as a sequence of one or more chunks of instantaneous plans.

4.3.0.5 *Orchestration process model*

The main purpose of an orchestrator is to control the execution of the plans incoming from Planner using the information from the Repository module. Therefore, the main problem is to execute many plans in parallel. The plans that are overlapping by using the same resources should be synchronized.

Definition 7 (Plan Synchronization Problem). *A problem of merging two non-independent instantaneous plans π_1 and π_2 is to find a tuple $\langle \pi_1^*, \pi_2^* \rangle$, such that:*

- *all common ordered sub-sequence of actions are represented only once in either of the plans;*
- *all critical sections are properly guarded;*
- *the plans are deadlock free, that is for any possible execution of plans, it will always be no deadlock.*

There are many plans concurrently in execution that are synchronized. When the new plan arrives, it should also be synchronized with the others.

Definition 8 (New Synchronization Problem). *Given a merged structure π_1, to merge a new plan π_2 into the structure means to find a tuple $\langle \pi_1^*, \pi_2^* \rangle$, such that:*

- *all common ordered sub-sequence of actions are represented only once among the plans;*
- *all critical sections are properly guarded;*
- *the plans are deadlock free, that is for any possible execution of plans, it will always be no deadlock.*

It is assumed that there is a special layer between the orchestrator and the planner which transforms plans according to the needs of the orchestration engine. It converts all plans into the list of Activities since the orchestrator operates the plans in such terms. Every Activity possesses the list of the resources that are needed to complete the task. A term Process is interchangeable with the term Plan, but it is considered to be Activity.

```
case class Process(main: Activity) extends EmbeddedActivity(main)
```

Activity can be:

- Structured;
- Embedded;
- represented by one Invoke procedure.

```
case class Resource(name: String)
  override def toString = name


abstract class Activity
  def supervisedResources: List[Resource] = List.empty[Resource]
```

Definition 9 (Structured Activity). *Structured Activity is an Activity in form of Sequence or Flow.*

```
abstract class StructuredActivity(val list: List[Activity])
 extends Activity
 override def supervisedResources = list.flatMap(_.supervisedResources)
```

Definition 10 (Sequence Activity). *Sequence Activity is represented by the flat Map of the Activities with the resources attached to them.*

```
abstract class EmbeddedActivity(val activity: Activity)
 extends StructuredActivity(List(activity))
```

Definition 11 (Flow Activity). *Flow Activity is a list of Activities in a form of a tree.*

```
abstract class EmbeddedActivity(val activity: Activity)
 extends StructuredActivity(List(activity))
```

Definition 12 (Embedded Activity). *Embedded Activity is the Activity that is nested within the other Activity.*

```
abstract class EmbeddedActivity(val activity: Activity)
 extends StructuredActivity(List(activity))
```

The example of the Embedded Activity is a Loop Activity within which many Activities can be nested.

Definition 13 (Loop Activity). *Loop Activity is an Activity that repeats specified number of times and can contain the other Activities inside itself.*

```
case class  Loop(times: Int, loopActivity: Activity)
 extends EmbeddedActivity(loopActivity)
```

Definition 14 (Invoke). *Invoke Activity represents a single domain operation.*

```
case class  Invoke(name: String) extends Activity
```

There are some special types of Activities:

Definition 15 (Empty Activity). *Empty Activity represents the Activity that does nothing.*

```
case object Empty extends Activity
```

Definition 16 (Terminate Activity). *Terminate Activity is an Activity that allows to terminate currently running process.*

```
case object Terminate extends Activity
```

Definition 17 (Critical Section). *Critical section is a special type of Activity that is formed by the intermediary layer in case when at least one type of the resource needed to complete a task is limited in number. A special Software Transactional Memory from the Akka library [10] is used for the execution of the Critical section.*

```
case class  CriticalSection(resources: List[Resource], cs: Activity)
 extends EmbeddedActivity(cs)
 override def supervisedResources = resources ++ futureResources
 def futureResources = cs.supervisedResources
```

5 ORCHESTRATOR IMPLEMENTATION

The orchestrator is implemented in Scala programming language, and by using the IDEA Intelligence framework and Maven tool. Orchestrator is a part of the SM4ALL middleware which is responsible for the proper execution of user commands. It is a part of the composition layer of the system. It represents a component that provides an execution of the plans incoming from planner using the Repository as a reference.

5.1 EVENT-BASED ACTOR SCHEME

Event-driven programming or event-based programming is a programming paradigm in which the flow of the program is determined by events–i.e., sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads[25].

Event-driven programming could be a good choice for the environment such as Smart Homes where there are many heterogeneous devices producing the events which should be dealt with.

A special library is elaborated for the communication between the different actors within one cluster of servers for Scala and Java languages. It is Akka project[10]. Akka is chosen as one of the basic libraries for the project because it represents the platform for the next generation event-driven, scalable and fault-tolerant architectures based on the JVM (Java Virtual Machine). It provides Actor Model together with Software Transactional Memory what raises the level of abstraction and provides a better platform to build correct concurrent and scalable applications.

Actor in Akka is an abstraction that helps the developer not to deal with the explicit locking and thread management, making it easier to write correct concurrent and parallel systems. Usage of Actors provides an asynchronous, non-blocking and highly performant event-driven programming model.

Software Transactional Memory allows to have a transactional dataset with begin/commit/rollback semantics. It is very important for the application since it helps to share the data structures across actors. For example, for the counter of the processes currently at execution it is indispensable.

There are different policies for the sending the messages from one Actor to another. According to the Akka documentation, the messages are sent to an Actor through one of the 'send' methods. 'Tell' means "fire-and-forget", e.g. send a message asynchronously and return immediately. ′SendRequestReply′ means "send-and-reply-eventually", e.g. send a message asynchronously and wait for a reply through a Future. Here a timeout may be specified. This method throws an ′ActorTimeoutException′ if the call timed out. ′SendRequestReplyFuture′ sends a message asynchronously and returns a ′Future′[11]. Therefore, for various purposes one can choose different policies which would be more adequate to the situation. For the purposes of the concurrent execution within Smart Homes environment different policies were chosen for different purposes. Where the reply is needed in order to continue correct execution (i.e., the process needs a unique identifier at the starting point), the "send-and-reply-eventually" variant is chosen. When the reply is not so important, a "fire-and-forget" policy is applied within the application.

Actors also provides the abstraction for transparent distribution and the basis for truly scalable and fault-tolerant applications. Akka is Open Source and available under the Apache 2 License.

Akka is chosen for the management of the concurrent execution of the plans, high-level scalability of the solution and event-driven approach since it provides all abstractions that are needed in order to alleviate from the low-level controlling of the parallel schemes and it is an Open Source project.

5.2 PROCESS MODEL

The *Process model* describes the content of the plans in terms of orchestration engine needs. The complex plans that are received by the Orchestrator from the Planner are converted into the list of Activities. Every action of the plan is represented as an independent activity to execute. Every Activity has a list of resources attached to it for the correct execution of the step of the plan.

5.3 BASIC COMPONENTS

The basic components of the orchestrator are the *process server* consisting of the *Process Manager, Ticket Manager and Counter* and *Process Model* which was described in the previous subsection. *Process server* side is responsible for the receiving the plans from the planner and dealing with them. *Process manager* is responsible for the parsing and executing the plans. *Ticket manager* is a special type of Actor which holds the list of the available groups of the resources and provides the processes with the permission to use the resources evading deadlock problems and race conditions. *Process model* describes the possible activities that could be executed. For more information, please, refer to the subsection "Formal statements".

Process server is started with the start of the whole SM4ALL middleware. It works till the user shuts down the whole system. The server is able to receive the messages containing plans, give them to the process manager and send the activities containing in the plan to the execution.

When the server starts, the special types of Actors are started as well. The first type of an Actor is a *Counter* which issues the process identifiers. Process for the further notes is counted as an interchangeable word for the plan. The second type of an Actor is a *Ticket Manager* which should know at the start if there are any processes to execute in the system.

When the server stops, it closes all Actors dependable from it. The code for the main functionality of server in Scala is following:

```
 class ProcessServer
  def start(): ActorRef =
    val counter        = actorOf[Counter].start()
    val ticketManager  = actorOf(new TicketManager(counter)).start()
    actorOf(new ProcessManager(ticketManager, counter)).start()


  def stop()
    Thread.sleep(2000)
    registry.shutdownAll()
```

Every process should receive the identifier in order to be executable. It is done because the server needs to know the identity of every process in execution for managing the resources and security reasons. To give the identifier to every process a special type of Actor is used which is *Counter*. For the purpose of the project it is implemented as just the increment of the non-negative integers. However, for the security reasons it can be easily expanded to the more complicated scheme.

When the Counter receives the request for identifier through the special Counter message, it increments the counter and replies with the new counter:

```
 case object CounterMessage

class Counter extends Actor
  var counter: Long = 0;

  def receive =
    case CounterMessage =>
      counter = counter + 1
      self.reply(counter)
```

When server receives the plan from the planner, the process is dealt with by *Process manager* part of the server. Every process is executed independently.

The *Process manager* needs the information about how to find *Ticket Manager* and the *Counter* in order to execute process from its starting point. Therefore, the references for the ticket manager and the counter are given at the beginning of the system running.

For the easy monitoring of the system and controlling its behavior the log is created where every step of the running orchestration system is written.

The messages for the process manager could be of two different types. The first type is a new plan to execute. The second type is the request of counting how many processes are in execution at the moment.

When the Process manager receives a message, it should check whether this message contains a process or the process count request. If the message contains a new plan to execute, the process manager sends the message to the Counter in order to receive the new identifier for the plan.

If the identifier is granted by Counter, the function *execute* is called together with the incrementing the number of the processes in execution. The incrementing of the processes in execution is done by using atomic conditions to keep the count of processes always valid. Atomic condition allows to change the value of the variable by only one process at the moment. When the execution of the plan is finished, the counter of the processes in execution is decremented by using atomic condition.

If the identifier is not granted by *Counter*, a special message is written to the log about the stopping of the execution of the process because of the failed returning of the process identifier.

In the case when the message received by the process manager contains the request for the counting the plans currently in execution, it replies with the requested counter.

```
case class  ProcessContext(processId: String)
case object ProcessCount

class ProcessManager(val ticketManager: ActorRef,
 val counter: ActorRef)
 extends Actor
 val log = LoggerFactory.getLogger(classOf[ProcessManager])

 val ref = Ref(0)

 def receive =
   case Process(main) =>
     counter !! CounterMessage match
       case Some(processId) =>
         spawn
           atomic
             ref alter (_ + 1)
```

```
          execute(ProcessContext(processId.toString), main)
          atomic
            ref alter (_ - 1)


        case x => log.error("Counter failed to return processId ,
         process stopped!", x)

    case ProcessCount =>
      self.reply(atomic
        ref.get
      )
```

The execution of every plan is performed by calling the function *execute* which is originally the part of the *Process Manager*. The process context containing the general information over the process (including process identifier) and a plan itself as a list of Activities is passed to the function for the execution.

As it was described previously in *Process model* section, Activity may contain many Activities in itself as a Sequence, Flow or the Loop. The function *execute* in the case of receiving the Sequence or the Flow of Activities for each item containing in the list invokes itself recursively. That means that at the end every plan consists of the sequential ordered list of invocations of single Activities. In the case of receiving the Loop Activity, the function *execute* invokes itself recursively as many times as it is needed.

If the function *execute* receives the call for the single Activity, the case *Invoke* is chosen. At the beginning of the execution the message reporting the start of the Activity is written to the log. After that the Activity is executed. At the end of the execution the reporting message about finishing the execution is written to a log.

A special case is a *Critical section* occurrence. A Critical section is assigned for dealing with the limited resources groups. Some resources involve short type of action and they can be triggered at any moment as many times as user wants. The action following the last user command would be taken into consideration in this case. The example of such activity could be turn on/ turn off the lamp. It does not matter how many times the activity is performed. The result will be the last user command. However, some resources could be limited in number or even exclusive. The example of such resource could be the only bathroom in the apartment. If one user occupied it, another user could not go in there for some time. For such type of the resources a special resolving solution should be procured.

The solution that is proposed concerns the exclusive or limited resources. First, every process should acquire a special ticket from the *Ticket Manager* as a permission to use a resource. If resource is available, the process receives the ticket and could proceed with the execution. If the resource is not available, the process should wait until the resource is available and then after a definite amount of time to request the ticket again. For the concurrent execution without the deadlocks a special scheme is running for the *Ticket Manager* which is described in a section "The Scenarios and Orchestrator Running".

In the case of the receiving the message that does not contain neither Sequence, Flow, Loop, or Invoke, nor Critical Section, the message is ignored and a warning is written to the log.

The code for the function *execute* demonstrates the procedure of the executing of one plan:

```
def execute(ctx: ProcessContext, a: Activity): Unit = a match
   case Sequence(list) => list foreach (execute(ctx, _))
   case Flow(list)     => list foreach (execute(ctx, _))
   case Loop(times, activity) =>
     if (times > 0)
       execute(ctx, activity)
       execute(ctx, Loop(times - 1, activity))

   case Invoke(name)   =>
     log.info(ctx.processId + ": Invoke started: ", name)
     Thread.sleep(3000)
     log.info(ctx.processId + ": Invoke finished: ", name)
   case cs: CriticalSection =>
     // get ticket
     ticketManager !! GetTicket(ctx.processId, cs.resources,
       cs.futureResources) match
       case Some(Ticket(id)) =>
         execute(ctx, cs.cs);
         // release ticket
         ticketManager ! ReleaseTicket(ctx.processId, id,
          cs.resources)
       case Some(TicketNotAvailable(reason)) =>
         log.debug(ctx.processId +
          ": Conflict:  while asking for ",
          reason, cs.resources)
         // wait and retry
         Thread.sleep(1000)
         execute(ctx, cs)
       case x => log.error("Unexpected message received
        from TicketManager: ", x)

   case x => log.error("Unknown process element,
    ignoring: ", x)
```

For dealing with the tickets for the limited resources a special *Ticket Manager* is developed. Ticket Manager is a special Actor within the system. There is only one Ticket Manager for the whole orchestration engine. However, regards the results of testing it is not a bottleneck for the system. For more information about the testing evaluation it is recommended to refer to the Chapter "Testing and Evaluation of the Solution".

When starting, Ticket Manager launches the log and two Map structures. One Map structure is responsible for the storage of data about the processes to which the tickets for the resources are given. The other Map structure is responsible for the data storage of the resources to which the tickets are presented.

Ticket Manager is able to receive messages and to respond on them. The types of messages it may receive are:

- request to get ticket for the limited type or the resource;
- request for release of the ticket for the limited type of resource;
- all other messages are marked as unknown and are not parsed.

When Ticket Manager receives the request for getting the ticket for the limited resource, it, at first, checks whether the ticket is allowed to be issued. It checks the availability of the resource by looking through the Map "Resource - Ticket". If there are no anymore resources of the requested type, it sends the message to the process-sender that the ticket is unavailable. If the resource is still available, Ticket manager checks dependencies for the variables representing future resources. Future resource notion is provided as by checking future resources needs of the process comparing it with the other processes concurrently in execution it is possible to avoid cyclic dependencies. If the future resources contain some resources that are already in use, the ticket is not issued. Otherwise, the ticket is granted.

In case of the receiving message containing the request to release ticket it removes ticket note from both Map structures, unlocks all resources that are mentioned in message and sends a notification that ticket is successfully removed.

```
abstract class TicketMessage
case class Ticket(ticketId: String) extends TicketMessage
case class TicketNotAvailable(reason: String) extends TicketMessage

case class GetTicket(processId: String, resources: List[Resource],
 future: List[Resource]) extends TicketMessage
case class ReleaseTicket(processId: String, ticketId: String,
resources: List[Resource])
extends TicketMessage

class TicketManager(val counter: ActorRef) extends Actor
  val log = LoggerFactory.getLogger(classOf[TicketManager])

  var tickets     = Map[Ticket, String]()
  var res_tickets = Map[Resource, Ticket]()


  def receive =
    case GetTicket(processId, resources, future) =>
      // check if ticket has to be issued:
      // 1. check if the resource has already been taken
      if (resources exists(r => res_tickets.contains(r)))
        self.reply(TicketNotAvailable("One of the
         requested resources is locked: " + resources))
       // 2. check if the process future depends on locked vars
        //this condition is actually a bit strong: we only care
        //if it leads to cyclic dependencies
      else if (future exists(r => res_tickets.contains(r)))
        self.reply(TicketNotAvailable("My future resources
         are locked: " + future))
       else
        // issue ticket
        self.reply(issueTicket(processId, resources, future))

    case ReleaseTicket(processId, ticketId, resources) =>
      // 1. remove from tickets
      tickets -= Ticket(ticketId)
      // 2. unlock resources
      resources foreach (r => res_tickets -= r)

      log.debug(processId + ": Ticket released: ", ticketId)
      log.trace(processId + ": TICKETS: ", tickets)
      log.trace(processId + ": RES_TIC: ", res_tickets)
    case x => log.info("Received unknown message: .
     Ignoring it!", x);


  def issueTicket(processId: String, resources: List[Resource],
     future: List[Resource]):
      Ticket = counter !! CounterMessage match
      case Some(ticketId) =>
        val ticket = Ticket(ticketId.toString)
        // 1. keep the process id for each ticket
        tickets += ticket -> processId
        // 2. for each resource add ticket that keeps it locked,
        //consider exclusive resources first
        resources foreach (r => res_tickets += r -> ticket)

        log.debug(processId + ": Ticket issued:  for ",
         ticket, resources)
        log.trace(processId + ": TICKETS: ", tickets)
        log.trace(processId + ": RES_TIC: ", res_tickets)

        return ticket
      case None => throw new IllegalStateException
       ("Unknown response from counter,
       cannot issue a ticket!")

    Ticket((counter !! CounterMessage get).toString)
```

The process of issuing ticket is performed by adding to the Map structure "Ticket - Process identifier" a needed information and by mapping the resource to the ticket in other Map structure. At the end of the process a notification is sent to the sender with the response to the request.

5.4 communication with environment

Communication of an orchestrator with the environment could be performed in different ways. Since the orchestrator is an independent module of the middleware working as a "black box", every possible solution may be chosen in order to connect to other modules. It could use the HTTP protocol and TCP/IP stack. There is a possibility to use the web services protocols of communication, would it be SOAP messages[24] or REST interface of services[14].

The other possible solution is to use JSON library[17]. JSON (JavaScript Object Notation) is a lightweight data-interchange format. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

Everyone could choose which protocol he/she likes to use for an orchestrator communication with the other modules or even directly with the outside environment as well in cases when it is needed.

6 THE SCENARIOS AND ORCHESTRATOR RUNNING

After that, the context is going to be extracted from the middleware platform, and is put into the repository for the orchestrator. All available devices are contained there in order to know for the orchestrator whether they are available or not. If the resources are exclusive (means - unique) and not available - they are in use by one of the users at the house, and others should wait for it.

While the orchestrator runs, it may execute many plans simultaneously. Every plan is executed independently. A special Actor is created to deal with every plan. The plans incoming from the Planner module are usually quite complex. The structure of such plans consists of many steps, or in terms of the orchestrator - Activities. Every Activity may need to lock the resources in order to successfully complete the stage. Some of the resources are limited. Therefore, it could be impossible to lock the unique resource, if it is already in use by some other process.

The solution is developed in a way to provide deadlock-free execution without race conditions of all processes. Every process is able to achieve the final goal if it is executed by using the proposed orchestration model. Such successful concurrent execution is provided by using the '*Ticket Manager* Actor.

For the limited resources groups it is possible to have different scenarios regarding whether some processes needs to lock the same resource or not. For the simplicity it is assumed that all plans are began to be executed at the same point of time.

The first scenario demonstrates the situation where every process needs to lock different limited resources, as it is demonstrated at Figure 4. In this scenario there are no overlapping plans. The execution of every plan proceeds smoothly without any problems. Every plan can lock the resources it needs for the completing the task.

The second scenario provides an overlook for the situation where the first plan needs the unique resource of the type a, the second plan needs the same resource a and after that - resource b, and the third plan needs resource of type c as it is shown at Figure 5. For such type of scenario there are two possible developments. First of all, the third plan does not overlap with any other plan. It is executed independently. However, the first and second plans are in need of the same unique resource a.

The first possibility of further development is where the first plan is successful in obtaining the ticket for the resource a. In this case the Actor responsible for the execution of the second plan waits for the

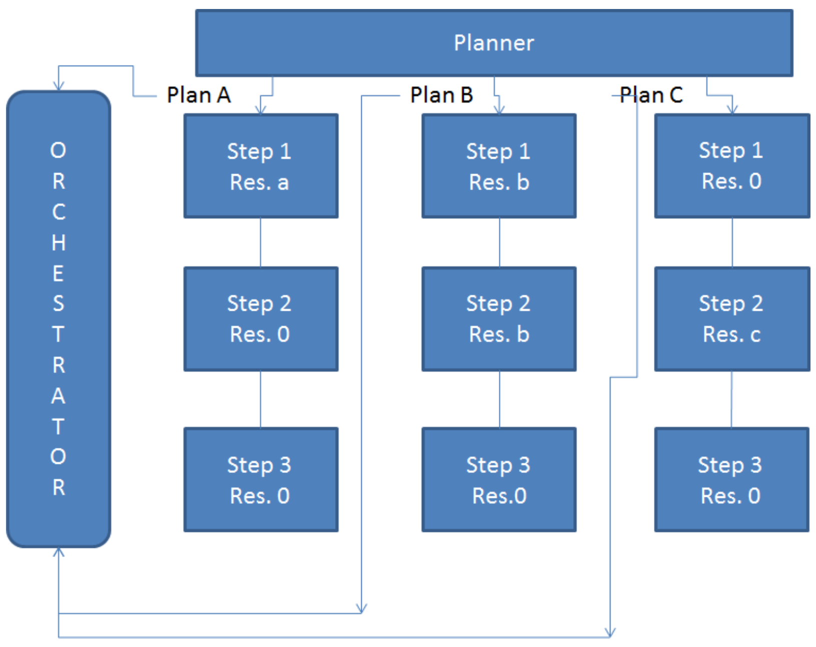

Figure 4: Execution of the plans locking independent resources.

release of ticket from the first plan. The resource b is considered to be the future resource, however, the ticket for it cannot be obtained because the plan is the sequence of actions where step 1 should be completed before step 2. When the first plan releases the resource a, second plan obtains the ticket for that resource and can continue execution without any problems. After completing the first step, second plan requests for the ticket for the resource of type b and receives it without any problem.

The second possibility of the situation development is where the second plan obtains the ticket for the unique resource a. In this case first plan should wait the release of the ticket for that resource. After obtaining the ticket for the resource a first plan could be executed till it achieves the final goal.

The third scenario demonstrates the situation where the first plan needs the unique resource of the type a and after that - resource b, the second plan needs the resource b and after that - again the resource b, and the third plan needs resource of type c as it is shown at Figure 6. The third plan again does not overlap with any other plan. It is executed independently. However, the first and second plans are in need of the same unique resource b. There two possible scenarios for this type of situation.

First plan needs to lock the unique resource of type a for the completing the first step. Second plan needs the unique resource of type b at the same time for the first step. Both plans are successful in obtaining the ticket for the execution of the first steps.

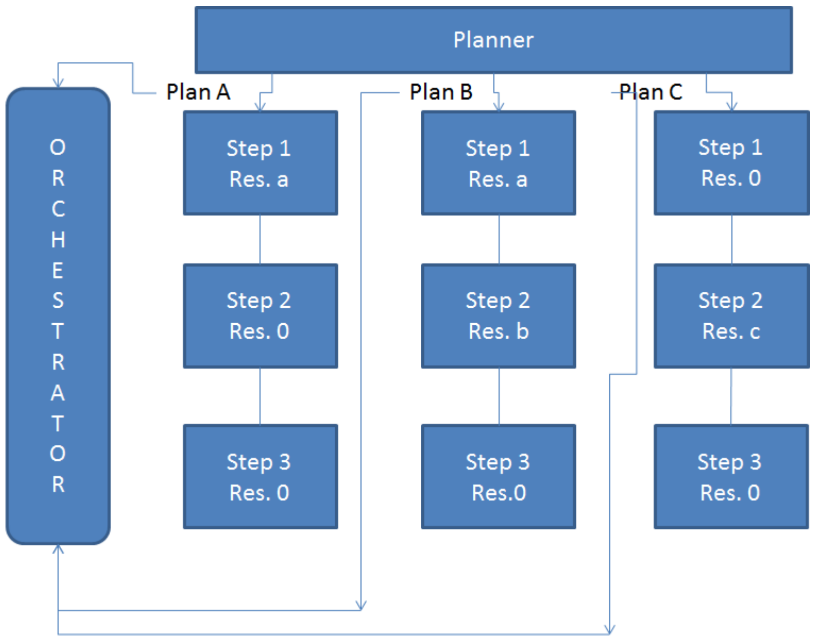

Figure 5: Execution of the plans locking one type of the unique resource.

After the completing of the first step every plan is in need for the resource b. The ticket for the resource b should be released by the plan after finishing the step. Therefore, the situation can develop in two ways. A new ticket for the unique resource b should be requested by both plans. One of them obtains the ticket (in FIFO mode), the other should wait until the releasing of the ticket for the resource. After that, the ticket could be successfully received by the remaining plan. Such solution is developed in order to avoid the possible starvation of the processes.

Forth scenario deals with the case where two of three plans are in need to lock the same resources, but the order of the lock is reversed. That means that first plan needs the resource of type a and then - b, and the second plan needs at first resource b and after that - a, as it is shown at the Figure 7. The third plan again does not overlap with any other plan. It is executed independently. However, the first and second plans are in need of the same unique resources a and b for the same step, but for two different critical sections.

In this case there could be a problem of the deadlock between the two processes. For that reason a special notion of future resources is introduced. Ticket manager saves the information for the every process not only about the resources the process needs at this point of time, but also about the resources that the process will need in the nearest future to complete the task.

Therefore, let us assume that the first process is the first to obtain the ticket for the resource of type a, and is subscribed for the resource of type b as a future resource. The Ticket Manager receives the request

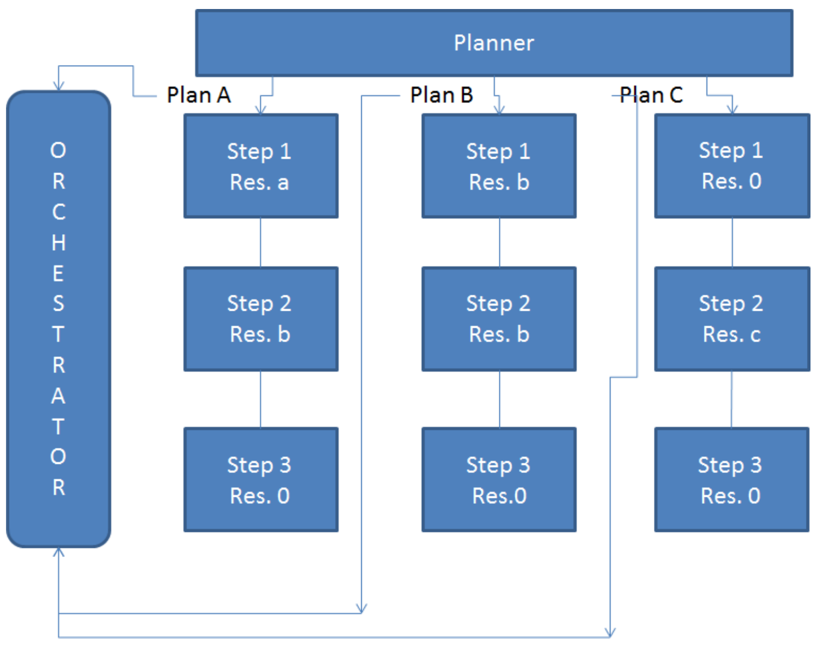

Figure 6: Execution of the plans locking the same type of the unique resource for step 2.

for the resource b from the second plan as a needed resource and for the resource a as a future resource. Ticket Manager check all plans already in execution for the availability of the resources. It finds that the resource a is requested by the first plan and it is not available. Furthermore, resource b is needed in future for the first plan to complete the task. In this case second plan is not able to obtain neither of the resources. It should wait the release of the ticket for the resource a to subscribe to it as a future resource, and the release of the ticket for the resource b to continue execution.

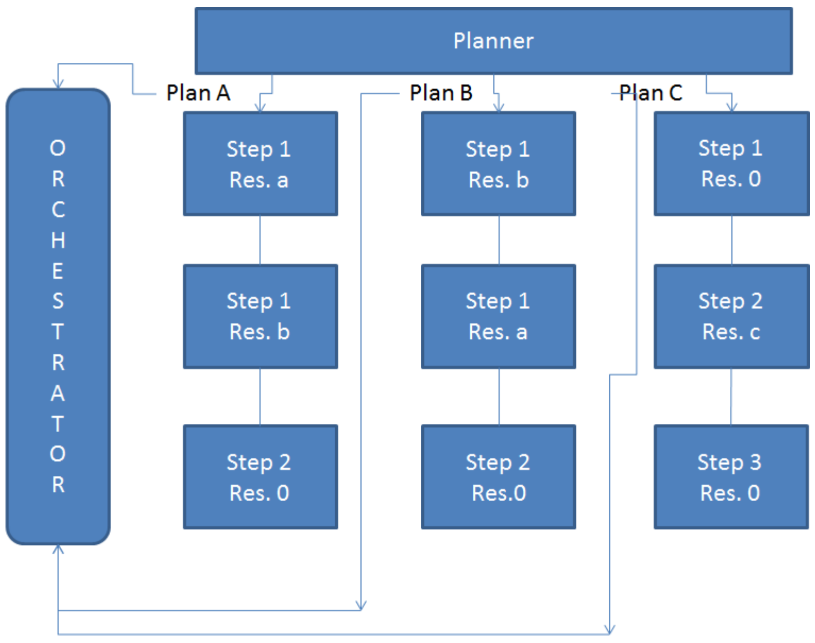

Figure 7: Execution of the plans locking the unique resources in reversed order.

7 TESTING AND EVALUATION OF THE SOLUTION

To check the feasibility and correctness of the solution the special tests are needed to be elaborated and executed. Moreover, it is possible to check the scalability and the percentage of usage of CPU. The tests were conducted to control the behavior of the system and to check different parameters in order to demonstrate high quality of the proposed solution.

7.1 CRITICAL SECTION TEST

The *Critical section* is the most important point of the system. If it performs well, that means that the application works correctly.

A critical section test contains the execution of two different plans. First plan is executed two times (as a Loop Activity), and it needs to lock the unique resource x, then - y, after that - z. Every request for the resource is made out to be a Critical section. Second plan is also executed two times (as a Loop Activity), and it needs to lock the unique resource p, then - z, after that - y. The main objective of the test is to demonstrate that all processes accomplished their tasks successfully. That means that at the end there should be zero processes currently in execution.

```
class CriticalSectionTest extends SpecificationWithJUnit

 "ProcessServer" should
   "execute all processes without deadlocks" in
     val ps = new ProcessServer().start()

     val x = List(Resource("x"))
     val y = List(Resource("y"))
     val z = List(Resource("z"))
     val p = List(Resource("p"))
     val cs_1 = new Process(Loop(2,
      new CS(x,
      new CS(y,
      new CS(z,
      new Invoke("Invoke 1"))))))
     val cs_2 = new Process(Loop(2,
      new CS(p,
      new CS(z,
      new CS(y,
      new Invoke("Invoke 2"))))))

     ps ! cs_1
     ps ! cs_2
```

```
    val res = wait(ps)

    ps.stop()

    res must_== 0



  @tailrec final def wait(ps: ActorRef): Int =
    Thread.sleep(1000)
    (ps !! ProcessCount) match
      case Some(x) => if (x != 0) wait(ps) else 0
      case None => 0
```

Test has been run over 20 times and every time it was finished successfully. The log of the console output is provided in the Appendix A.

7.2 DINING PHILOSOPHERS TEST

The dining philosophers is one of the famous problems in the field of concurrent execution of the processes. The problem statement for the dining philosophers from Wikipedia [26] is the following: Five silent philosophers sit at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. Eating is not limited by the amount of spaghetti left: assume an infinite supply. However, a philosopher can only eat while holding both the fork to the left and the fork to the right (an alternative problem formulation uses rice and chopsticks instead of spaghetti and forks).

Each philosopher can pick up an adjacent fork, when available, and put it down, when holding it. These are separate actions: forks must be picked up and put down one by one.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that each philosopher won't starve, i.e. can forever continue to alternate between eating and thinking.

This problem is often used to check the deadlock-free execution of the systems. For this reason it was chosen to check the correctness of the concurrent algorithm. There are two slightly different tests were elaborated for the dining philosophers problem:

- where the philosopher claims two forks simultaneously to eat;
- where the philosopher claims one fork at a time and can wait for another.

For the first test the ultimate goal is to feed to philosophers three times. Every philosopher should be able to complete the task successfully by the end of the test.

```
class PhilosophersTest extends SpecificationWithJUnit

  "ProcessServer" should
    "feed all philosophers 3 times" in
      val ps = new ProcessServer().start()

      val p_1 = List(Resource("f1"), Resource("f2"))
      val p_2 = List(Resource("f2"), Resource("f3"))
      val p_3 = List(Resource("f3"), Resource("f4"))
      val p_4 = List(Resource("f4"), Resource("f5"))
      val p_5 = List(Resource("f5"), Resource("f1"))
      val cs_1 = new Process(Loop(3,
       new CS(p_1,
       new Invoke("P1 eats"))))
      val cs_2 = new Process(Loop(3,
       new CS(p_2,
       new Invoke("P2 eats"))))
      val cs_3 = new Process(Loop(3,
       new CS(p_3,
       new Invoke("P3 eats"))))
      val cs_4 = new Process(Loop(3,
       new CS(p_4,
       new Invoke("P4 eats"))))
      val cs_5 = new Process(Loop(3,
       new CS(p_5,
       new Invoke("P5 eats"))))

      ps ! cs_1
      ps ! cs_2
      ps ! cs_3
      ps ! cs_4
      ps ! cs_5

      val res = wait(ps)

      ps.stop()

      res must_== 0



  @tailrec final def wait(ps: ActorRef): Int =
    Thread.sleep(1000)
    (ps !! ProcessCount) match
      case Some(x) => if (x != 0) wait(ps) else 0
      case None => 0
```

The test was executed over 30 times and demonstrated that the solution is correct for such type of the problem. The output on the debugging console of the test could be found in Appendix B.

For the second test the ultimate goal is to feed to philosophers two times. While one fork could be claimed, the other one should be claimed as future resource. Every philosopher should be able to complete the task successfully by the end of the test.

```
class TruePhilosophersTest extends SpecificationWithJUnit

  "ProcessServer" should
    "feed real philosophers 2 times" in
      val prs = new ProcessServer().start()

      val f_1 = List(Resource("f1"))
      val f_2 = List(Resource("f2"))
      val f_3 = List(Resource("f3"))
      val f_4 = List(Resource("f4"))
      val f_5 = List(Resource("f5"))

      val phil_1 = Process(Loop(2, CriticalSection(f_1,
       CriticalSection(f_2, Invoke("Phil_1 eats")))))
      val phil_2 = Process(Loop(2, CriticalSection(f_2,
       CriticalSection(f_3, Invoke("Phil_2 eats")))))
      val phil_3 = Process(Loop(2, CriticalSection(f_3,
       CriticalSection(f_4, Invoke("Phil_3 eats")))))
      val phil_4 = Process(Loop(2, CriticalSection(f_4,
       CriticalSection(f_5, Invoke("Phil_4 eats")))))
      val phil_5 = Process(Loop(2, CriticalSection(f_5,
       CriticalSection(f_1, Invoke("Phil_5 eats")))))

      prs ! phil_1
      prs ! phil_2
      prs ! phil_3
      prs ! phil_4
      prs ! phil_5

      val rs = wait(prs)

      prs.stop()

      rs must_== 0



  @tailrec final def wait(prs: ActorRef): Int =
    Thread.sleep(1000)
    (prs !! ProcessCount) match
      case Some(x) => if (x != 0) wait(prs) else 0
      case None => 0
```

The test was executed over 30 times and demonstrated that the solution is correct for such type of the problem. The output on the debugging console of the test could be found in Appendix C.

7.3 THE EVALUATION OF THE SOLUTION

While running the tests and scenarios, the system behavior could be checked using different metrics in various situations.

One of the main issues for the distributed systems is always high level of performance. In software architecture practice, performance testing is such kind of testing that is done to determine how fast some aspect of a system performs under a particular workload. It can also serve as a checkpoint for the verification and validation of the other quality attributes of the system, such as scalability, reliability and resource usage.

The tests chosen for the demonstration of the high performance level capability of the system include the metrics:

- the number of processes in execution simultaneously;
- the maximum number of conflicts for the lock of the limited resources in the process of execution;
- the maximum number of concurrent lock conflicts;
- the load of CPU while processing the tests;
- the amount of the computer memory involved;
- the time of the running the tests.

Firstly, the experimental testing has been conducted to check how many conflicts appeared at maximum while executing the test for the dining philosophers problem depending on the number of the processes concurrently in execution. The number of working threads for the different number of processes has been limited to different numbers for the tests. As one of the examples, for the execution of 70 processes the limit of 16 concurrent execution threads is chosen, etc. The results are presented in Table 1. In the graphical representation

Processes	70	140	210	280	350	420	490	560	630	700	770
16 WT	94	176	250	322	399	461	551	622	694	779	844
128 WT	790	2797	3198	3795	4212	4572	4968	5764	6052	6354	6943
1024 WT	794	3231	7321	13060	20450	29491	40180	50774	64139	78087	92025

Table 1: Maximum number of conflicts depending on the number of working threads.

shown at the Figure 8 it is obvious that the limit of the simultaneously working threads should not be equal to the number of processes.

When there are a big number of processes in execution, the maximum number of locks conflicts increases practically exponentially. At the same time with the reasonable limit for working threads it is possible to achieve a good performance from the system. The next experiment

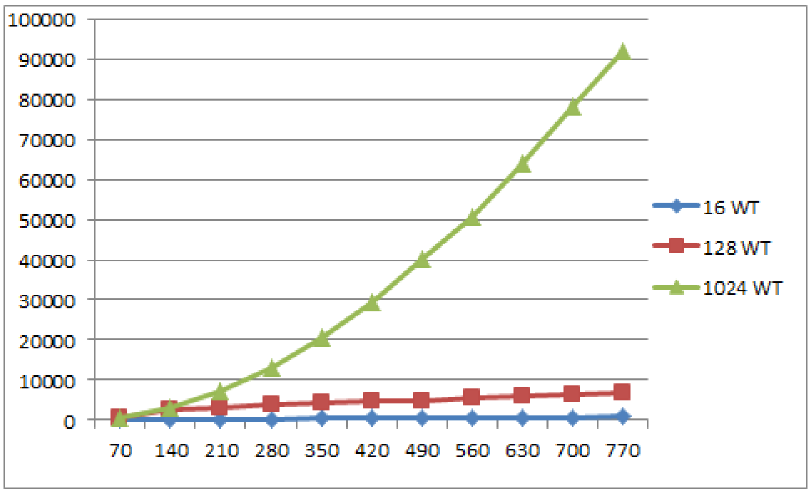

Figure 8: Maximum number of conflicts depending on the number of working threads.

is based on the number of working threads and it calculates the maximum amount of simultaneous locks. The results are presented in the Table 2.

Processes	70	140	210	280	350	420	490	560	630	700	770
16 WT	16	16	16	16	16	16	16	16	16	16	16
128 WT	67	128	128	128	128	128	128	128	128	128	128
1024 WT	67	137	207	277	347	417	487	556	626	696	766

Table 2: Maximum number of the concurrent conflicts depending on the number of working threads.

The chart at Figure 9 shows that for such amount of concurrent working threads it takes almost all permitted slots. However, there is a slight trend for the increasing amount of working threads to expect bigger deviation from the linear increase for the decreasing the number of slots. Such big amount of concurrent working threads is not what we could expect for Smart Homes environment. However, for the big Smart Offices buildings this metric could be quite interesting.

The next experiment represents the persentage of the CPU use regards the number of concurrent working threads. The results are provided in Table 3. The results of the testing processor use are quite interesting. At the chart 10 a jump could be seen at the beginning of the execution. It is expectable since the program starts to run and it needs initialize all variables in the environment. By the end of the

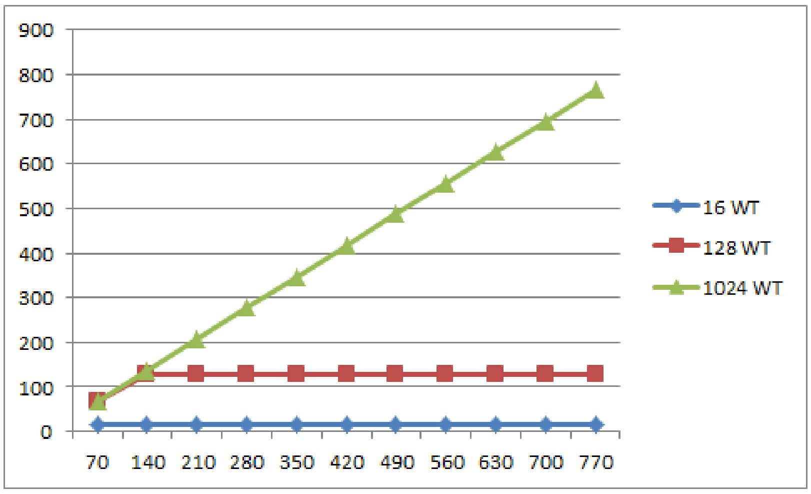

Figure 9: Maximum number of locks conflicts depending on the number of working threads.

Processes	70	140	210	280	350	420	490	560	630	700	770
16 WT	0.273	0.17	0.117	0.09	0.195	0.184	0.068	0.05	0.008	0.007	0.006
128 WT	0.165	0.065	0.142	0.04	0.032	0.067	0.018	0.021	0.036	0.024	0.024
1024 WT	0.125	0.09	0.118	0.019	0.037	0.036	0.016	0.015	0.013	0.02	0.028

Table 3: Maximum number of the concurrent conflicts depending on the number of working threads (in %).

execution it becomes more and more decreasing. It is the result that you could expect from the system of high quality of performance.

The check of processor use can be confirmed by demonstrating CPU use through Windows control panel. The check is done on the laptop with the 8 processor cores and demonstrated at the Figure 11.

Next testing experiment helps to calculate how much virtual memory (in MB) is used in total to finish the execution of all processes. The results are presented in Table 4. The chart 12 demonstrates that the

Processes	70	140	210	280	350	420	490	560	630	700	770
16 WT	24	2.7	1.4	3	3.7	4.1	8.2	11.7	8.2	11.5	1
128 WT	1	10.2	5.9	9.3	14.9	10.9	15.8	17	6.8	20.1	22.2
1024 WT	1	21.5	44.4	44.4	36.1	44.7	39.6	17.6	30.4	14.6	18.6

Table 4: Total memory used depending on the number of working threads.

big number of threads leads to the overload of the virtual memory in use by the program. Therefore, it is necessary to decide on rather low number of threads if the memory is critical for the execution.

The last experiment shows the total time of an execution (in sec) of concurrent plans depending on the limiting number of the concurrent threads. The results are demonstrated in Table 5. The chart 13 demonstrates that the big number of threads leads to the much longer

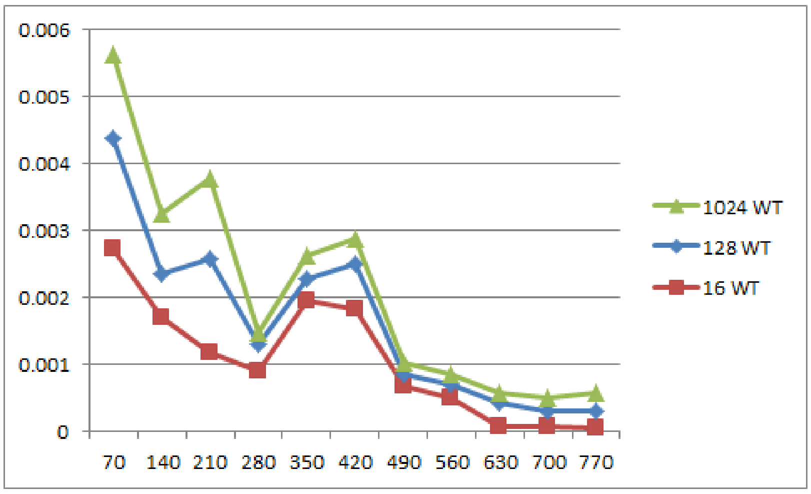

Figure 10: Percentage of CPU use depending on the number of working threads.

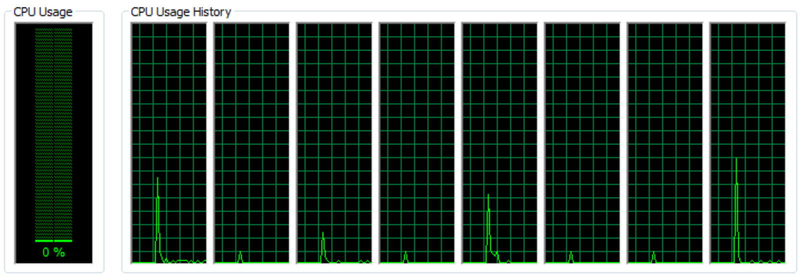

Figure 11: Percentage of CPU use depending on the number of working threads through Windows panel.

execution of the program. Even more, while having 1024 threads, there are some drops shown, but in general the tendency concerning very big number of threads is not good enough. Therefore, it is necessary to decide on rather low number of concurrent threads if the time is critical for the performance.

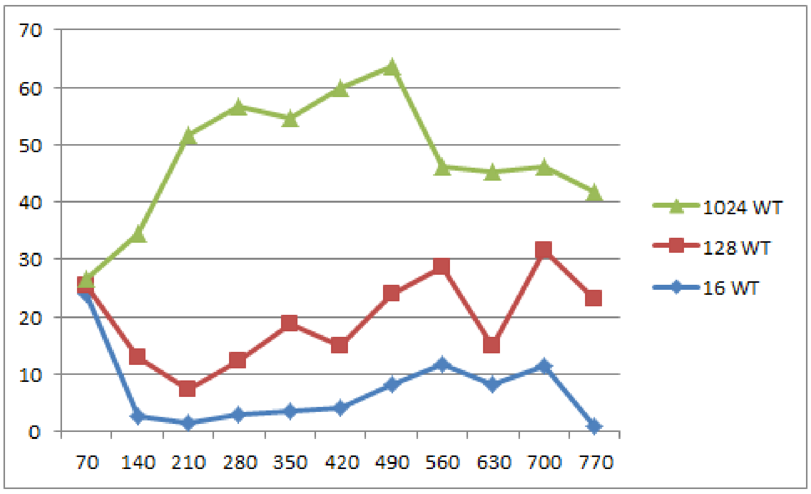

Figure 12: Total memory used for execution depending on the number of working threads.

Processes	70	140	210	280	350	420	490	560	630	700	770
16 WT	5	8	10	13	16	17	20	23	25	28	41
128 WT	13	24	26	29	30	32	33	37	38	40	31
1024 WT	14	26	38	50	63	76	88	101	113	126	139

Table 5: Time of execution depending on the number of working threads.

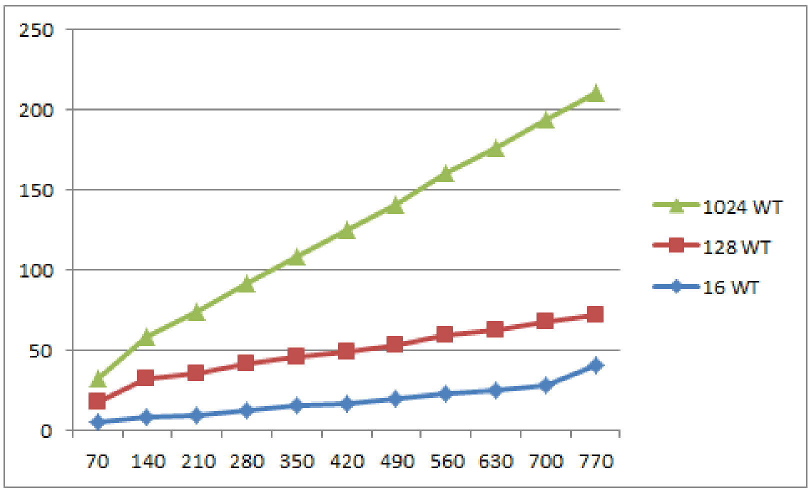

Figure 13: Execution time depending on the number of concurrent threads.

8 CONCLUSIONS AND DISCUSSIONS

Service-oriented computing is one of the emerging paradigms in computer science and the world of software applications nowadays. It is happened because modern software systems are often developed in distributed manner. Every software program or application has its own format and policy of use. Therefore, the problem of interaction between the systems and heterogeneity of the field in general is arising with the increasing of the number of different applications. Service-oriented computing deals with the heterogeneity of devices, software applications and system by introducing corresponding concepts and technologies that cope with such class of problems. Service-oriented computing concepts have been successfully applied in different areas, including pervasive computing, mobile software, embedded devices. Domotics is one of the complicated domains which is highly heterogeneous environment with a huge amount of the distributed physical devices in use. Service-oriented programming has proved to be a perfect solution to deal with the issues of Smart Homes. However, it does not respond to the question how the services can be executed at the same time in reliable manner

The presented work proposes a solution to the problem of concurrent execution of the complex scenarios within distributed higly heterogeneous environments. The development of the software prototype demonstrated that finding a solution to this problem is definitely feasible and can be done using modern technologies.

Ohe of the main advantages of the developed solution is always to keep high levels of performance while executing the concurrent plans. Even when the plans are overlapping in their needs for the same resource and a critical section concept is used, it never takes considerable amounts of time to continue the execution.

The other advantage of the chosen scheme is that it based on an event-based Actor system which is light-weight distributed, highly performing and very scalable solution. Every plan could be executed within the heterogeneous distributed system by different Actor which communicate with each other in order to coordinate their activities.

Testing results prove the system to be correct and reliable. Based on the evaluation results and the consumed resources, it is obvious that the solution can be used even in Embedded systems environment, such as mobile applications. Moreover, the running prototype is a general solution that can be taken in consideration not only for the domotics domain, but it could be applied for the different distributed systems.

The proposed solution is a core of the orchestration engine for the concurrent execution of complex scenarios. However, some issues are still remaining out of scope of the presented work. For example, the orchestration engine prototype can be easily expanded by developing new features that facilitate the achievement of the higher level of performance and correspond to the other system quality requirements.

One of the other future development is elaborating an intermediary layer between the Planner and Orchestrator modules able to format plans incoming from Planner the way they are needed by the orhcestration engine.

Moreover, the system needs not only concurrent execution of the complex processes, but also a synergy solution for the situation when different users use the resource together. An example of such situation is a door opened for two users, and only when they both enter the next room, the door closes. Another example is a table that could be shared by a specific number of users simultaneously. These issues is beyond of the scope of the presented work, however, it is very important to have it in Smart Homes.

APPENDIX A

```
TicketManager - 2: Ticket issued: Ticket(4) for List(x)
TicketManager - 2: TICKETS: Map(Ticket(4) -> 2)
TicketManager - 2: RES_TIC: Map(x -> Ticket(4))
TicketManager - 3: Ticket issued: Ticket(5) for List(p)
TicketManager - 3: TICKETS: Map(Ticket(4) -> 2, Ticket(5) -> 3)
TicketManager - 3: RES_TIC: Map(x -> Ticket(4), p -> Ticket(5))
TicketManager - 2: Ticket issued: Ticket(6) for List(y)
TicketManager - 2: TICKETS: Map(Ticket(4) -> 2, Ticket(5) -> 3,
Ticket(6) -> 2)
TicketManager - 2: RES_TIC: Map(x -> Ticket(4), p -> Ticket(5),
y -> Ticket(6))
ProcessManager - 3: Conflict: My future resources are locked:
List(y) while asking for List(z)
TicketManager - 2: Ticket issued: Ticket(7) for List(z)
TicketManager - 2: TICKETS: Map(Ticket(4) -> 2, Ticket(5) -> 3,
Ticket(6) -> 2, Ticket(7) -> 2)
TicketManager - 2: RES_TIC: Map(x -> Ticket(4), p -> Ticket(5),
y -> Ticket(6), z -> Ticket(7))
ProcessManager - 2: Invoke started: Invoke 1
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(z) while asking for List(z)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(z) while asking for List(z)
ProcessManager - 2: Invoke finished: Invoke 1
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(z) while asking for List(z)
TicketManager - 2: Ticket released: 7
TicketManager - 2: TICKETS: Map(Ticket(4) -> 2, Ticket(5) -> 3,
Ticket(6) -> 2)
TicketManager - 2: RES_TIC: Map(x -> Ticket(4), p -> Ticket(5),
y -> Ticket(6))
TicketManager - 2: Ticket released: 6
TicketManager - 2: TICKETS: Map(Ticket(4) -> 2, Ticket(5) -> 3)
TicketManager - 2: RES_TIC: Map(x -> Ticket(4), p -> Ticket(5))
TicketManager - 2: Ticket released: 4
TicketManager - 2: TICKETS: Map(Ticket(5) -> 3)
TicketManager - 2: RES_TIC: Map(p -> Ticket(5))
TicketManager - 2: Ticket issued: Ticket(8) for List(x)
TicketManager - 2: TICKETS: Map(Ticket(5) -> 3, Ticket(8) -> 2)
TicketManager - 2: RES_TIC: Map(p -> Ticket(5), x -> Ticket(8))
TicketManager - 2: Ticket issued: Ticket(9) for List(y)
TicketManager - 2: TICKETS: Map(Ticket(5) -> 3, Ticket(8) -> 2,
Ticket(9) -> 2)
```

```
TicketManager - 2: RES_TIC: Map(p -> Ticket(5), x -> Ticket(8),
y -> Ticket(9))
TicketManager - 2: Ticket issued: Ticket(10) for List(z)
TicketManager - 2: TICKETS: Map(Ticket(5) -> 3, Ticket(8) -> 2,
Ticket(9) -> 2, Ticket(10) -> 2)
TicketManager - 2: RES_TIC: Map(p -> Ticket(5), x -> Ticket(8),
y -> Ticket(9), z -> Ticket(10))
ProcessManager - 2: Invoke started: Invoke 1
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(z) while asking for List(z)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(z) while asking for List(z)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(z) while asking for List(z)
ProcessManager - 2: Invoke finished: Invoke 1
TicketManager - 2: Ticket released: 10
TicketManager - 2: TICKETS: Map(Ticket(5) -> 3, Ticket(8) -> 2,
Ticket(9) -> 2)
TicketManager - 2: RES_TIC: Map(p -> Ticket(5), x -> Ticket(8),
y -> Ticket(9))
TicketManager - 2: Ticket released: 9
TicketManager - 2: TICKETS: Map(Ticket(5) -> 3, Ticket(8) -> 2)
TicketManager - 2: RES_TIC: Map(p -> Ticket(5), x -> Ticket(8))
TicketManager - 2: Ticket released: 8
TicketManager - 2: TICKETS: Map(Ticket(5) -> 3)
TicketManager - 2: RES_TIC: Map(p -> Ticket(5))
TicketManager - 3: Ticket issued: Ticket(11) for List(z)
TicketManager - 3: TICKETS: Map(Ticket(5) -> 3, Ticket(11) -> 3)
TicketManager - 3: RES_TIC: Map(p -> Ticket(5), z -> Ticket(11))
TicketManager - 3: Ticket issued: Ticket(12) for List(y)
TicketManager - 3: TICKETS: Map(Ticket(5) -> 3, Ticket(11) -> 3,
Ticket(12) -> 3)
TicketManager - 3: RES_TIC: Map(p -> Ticket(5), z -> Ticket(11),
y -> Ticket(12))
ProcessManager - 3: Invoke started: Invoke 2
ProcessManager - 3: Invoke finished: Invoke 2
TicketManager - 3: Ticket released: 12
TicketManager - 3: TICKETS: Map(Ticket(5) -> 3, Ticket(11) -> 3)
TicketManager - 3: RES_TIC: Map(p -> Ticket(5), z -> Ticket(11))
TicketManager - 3: Ticket released: 11
TicketManager - 3: TICKETS: Map(Ticket(5) -> 3)
TicketManager - 3: RES_TIC: Map(p -> Ticket(5))
TicketManager - 3: Ticket released: 5
TicketManager - 3: TICKETS: Map()
TicketManager - 3: RES_TIC: Map()
TicketManager - 3: Ticket issued: Ticket(13) for List(p)
TicketManager - 3: TICKETS: Map(Ticket(13) -> 3)
TicketManager - 3: RES_TIC: Map(p -> Ticket(13))
TicketManager - 3: Ticket issued: Ticket(14) for List(z)
TicketManager - 3: TICKETS: Map(Ticket(13) -> 3, Ticket(14) -> 3)
TicketManager - 3: RES_TIC: Map(p -> Ticket(13), z -> Ticket(14))
TicketManager - 3: Ticket issued: Ticket(15) for List(y)
TicketManager - 3: TICKETS: Map(Ticket(13) -> 3, Ticket(14) -> 3,
Ticket(15) -> 3)
TicketManager - 3: RES_TIC: Map(p -> Ticket(13), z -> Ticket(14),
y -> Ticket(15))
ProcessManager - 3: Invoke started: Invoke 2
ProcessManager - 3: Invoke finished: Invoke 2
TicketManager - 3: Ticket released: 15
TicketManager - 3: TICKETS: Map(Ticket(13) -> 3, Ticket(14) -> 3)
TicketManager - 3: RES_TIC: Map(p -> Ticket(13), z -> Ticket(14))
TicketManager - 3: Ticket released: 14
TicketManager - 3: TICKETS: Map(Ticket(13) -> 3)
TicketManager - 3: RES_TIC: Map(p -> Ticket(13))
TicketManager - 3: Ticket released: 13
TicketManager - 3: TICKETS: Map()
TicketManager - 3: RES_TIC: Map()

Process finished with exit code 0
```

APPENDIX B

```
TicketManager - 2: Ticket issued: Ticket(7) for List(f1, f2)
TicketManager - 2: TICKETS: Map(Ticket(7) -> 2)
TicketManager - 2: RES_TIC: Map(f1 -> Ticket(7), f2 -> Ticket(7))
ProcessManager - 2: Invoke started: P1 eats
TicketManager - 5: Ticket issued: Ticket(8) for List(f4, f5)
TicketManager - 5: TICKETS: Map(Ticket(7) -> 2, Ticket(8) -> 5)
TicketManager - 5: RES_TIC: Map(f1 -> Ticket(7), f2 -> Ticket(7),
f4 -> Ticket(8), f5 -> Ticket(8))
ProcessManager - 5: Invoke started: P4 eats
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5, f1) while asking for List(f5, f1)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3, f4) while asking for List(f3, f4)
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5, f1) while asking for List(f5, f1)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3, f4) while asking for List(f3, f4)
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5, f1) while asking for List(f5, f1)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3, f4) while asking for List(f3, f4)
ProcessManager - 2: Invoke finished: P1 eats
TicketManager - 2: Ticket released: 7
TicketManager - 2: TICKETS: Map(Ticket(8) -> 5)
TicketManager - 2: RES_TIC: Map(f4 -> Ticket(8), f5 -> Ticket(8))
ProcessManager - 5: Invoke finished: P4 eats
TicketManager - 2: Ticket issued: Ticket(9) for List(f1, f2)
TicketManager - 2: TICKETS: Map(Ticket(8) -> 5, Ticket(9) -> 2)
TicketManager - 2: RES_TIC: Map(f4 -> Ticket(8), f5 -> Ticket(8),
f1 -> Ticket(9), f2 -> Ticket(9))
ProcessManager - 2: Invoke started: P1 eats
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5, f1) while asking for List(f5, f1)
TicketManager - 5: Ticket released: 8
TicketManager - 5: TICKETS: Map(Ticket(9) -> 2)
TicketManager - 5: RES_TIC: Map(f1 -> Ticket(9), f2 -> Ticket(9))
TicketManager - 5: Ticket issued: Ticket(10) for List(f4, f5)
TicketManager - 5: TICKETS: Map(Ticket(9) -> 2, Ticket(10) -> 5)
TicketManager - 5: RES_TIC: Map(f1 -> Ticket(9), f2 -> Ticket(9),
f4 -> Ticket(10), f5 -> Ticket(10))
ProcessManager - 5: Invoke started: P4 eats
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3, f4) while asking for List(f3, f4)
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5, f1) while asking for List(f5, f1)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3, f4) while asking for List(f3, f4)
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5, f1) while asking for List(f5, f1)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3, f4) while asking for List(f3, f4)
ProcessManager - 2: Invoke finished: P1 eats
TicketManager - 2: Ticket released: 9
TicketManager - 2: TICKETS: Map(Ticket(10) -> 5)
TicketManager - 2: RES_TIC: Map(f4 -> Ticket(10),
f5 -> Ticket(10))
TicketManager - 2: Ticket issued: Ticket(11) for List(f1, f2)
TicketManager - 2: TICKETS: Map(Ticket(10) -> 5,
Ticket(11) -> 2)
ProcessManager - 5: Invoke finished: P4 eats
TicketManager - 2: RES_TIC: Map(f4 -> Ticket(10),
f5 -> Ticket(10),
f1 -> Ticket(11),
f2 -> Ticket(11))
ProcessManager - 2: Invoke started: P1 eats
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5, f1) while asking for List(f5, f1)
```

```
TicketManager - 5: Ticket released: 10
TicketManager - 5: TICKETS: Map(Ticket(11) -> 2)
TicketManager - 5: RES_TIC: Map(f1 -> Ticket(11),
f2 -> Ticket(11))
TicketManager - 5: Ticket issued: Ticket(12) for List(f4, f5)
TicketManager - 5: TICKETS: Map(Ticket(11) -> 2,
Ticket(12) -> 5)
TicketManager - 5: RES_TIC: Map(f1 -> Ticket(11),
f2 -> Ticket(11),
f4 -> Ticket(12),
f5 -> Ticket(12))
ProcessManager - 5: Invoke started: P4 eats
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3, f4) while asking for List(f3, f4)
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5, f1) while asking for List(f5, f1)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3, f4) while asking for List(f3, f4)
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5, f1) while asking for List(f5, f1)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3, f4) while asking for List(f3, f4)
ProcessManager - 2: Invoke finished: P1 eats
TicketManager - 2: Ticket released: 11
TicketManager - 2: TICKETS: Map(Ticket(12) -> 5)
TicketManager - 2: RES_TIC: Map(f4 -> Ticket(12),
f5 -> Ticket(12))
ProcessManager - 5: Invoke finished: P4 eats
TicketManager - 5: Ticket released: 12
TicketManager - 5: TICKETS: Map()
TicketManager - 5: RES_TIC: Map()
TicketManager - 4: Ticket issued: Ticket(13) for List(f3, f4)
TicketManager - 4: TICKETS: Map(Ticket(13) -> 4)
TicketManager - 4: RES_TIC: Map(f3 -> Ticket(13),
f4 -> Ticket(13))
ProcessManager - 4: Invoke started: P3 eats
ProcessManager - 6: Conflict: One of the requested resources
is locked:
```

```
List(f5, f1) while asking for List(f5, f1)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
TicketManager - 6: Ticket issued: Ticket(14) for List(f5, f1)
TicketManager - 6: TICKETS: Map(Ticket(13) -> 4,
Ticket(14) -> 6)
TicketManager - 6: RES_TIC: Map(f3 -> Ticket(13),
f4 -> Ticket(13),
f5 -> Ticket(14),
f1 -> Ticket(14))
ProcessManager - 6: Invoke started: P5 eats
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Invoke finished: P3 eats
TicketManager - 4: Ticket released: 13
TicketManager - 4: TICKETS: Map(Ticket(14) -> 6)
TicketManager - 4: RES_TIC: Map(f5 -> Ticket(14),
f1 -> Ticket(14))
TicketManager - 4: Ticket issued: Ticket(15) for List(f3, f4)
TicketManager - 4: TICKETS: Map(Ticket(14) -> 6,
Ticket(15) -> 4)
TicketManager - 4: RES_TIC: Map(f5 -> Ticket(14),
f1 -> Ticket(14),
f3 -> Ticket(15),
f4 -> Ticket(15))
ProcessManager - 4: Invoke started: P3 eats
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 6: Invoke finished: P5 eats
TicketManager - 6: Ticket released: 14
TicketManager - 6: TICKETS: Map(Ticket(15) -> 4)
TicketManager - 6: RES_TIC: Map(f3 -> Ticket(15),
f4 -> Ticket(15))
TicketManager - 6: Ticket issued: Ticket(16) for List(f5, f1)
TicketManager - 6: TICKETS: Map(Ticket(15) -> 4,
Ticket(16) -> 6)
TicketManager - 6: RES_TIC: Map(f3 -> Ticket(15),
f4 -> Ticket(15),
f5 -> Ticket(16),
f1 -> Ticket(16))
ProcessManager - 6: Invoke started: P5 eats
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
```

```
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Invoke finished: P3 eats
TicketManager - 4: Ticket released: 15
TicketManager - 4: TICKETS: Map(Ticket(16) -> 6)
TicketManager - 4: RES_TIC: Map(f5 -> Ticket(16),
f1 -> Ticket(16))
TicketManager - 4: Ticket issued: Ticket(17) for List(f3, f4)
TicketManager - 4: TICKETS: Map(Ticket(16) -> 6,
Ticket(17) -> 4)
TicketManager - 4: RES_TIC: Map(f5 -> Ticket(16),
f1 -> Ticket(16),
f3 -> Ticket(17),
f4 -> Ticket(17))
ProcessManager - 4: Invoke started: P3 eats
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 6: Invoke finished: P5 eats
TicketManager - 6: Ticket released: 16
TicketManager - 6: TICKETS: Map(Ticket(17) -> 4)
TicketManager - 6: RES_TIC: Map(f3 -> Ticket(17),
f4 -> Ticket(17))
TicketManager - 6: Ticket issued: Ticket(18) for List(f5, f1)
TicketManager - 6: TICKETS: Map(Ticket(17) -> 4,
Ticket(18) -> 6)
TicketManager - 6: RES_TIC: Map(f3 -> Ticket(17),
f4 -> Ticket(17),
f5 -> Ticket(18),
f1 -> Ticket(18))
ProcessManager - 6: Invoke started: P5 eats
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 3: Conflict: One of the requested resources
is locked:
List(f2, f3) while asking for List(f2, f3)
ProcessManager - 4: Invoke finished: P3 eats
TicketManager - 4: Ticket released: 17
TicketManager - 4: TICKETS: Map(Ticket(18) -> 6)
TicketManager - 4: RES_TIC: Map(f5 -> Ticket(18),
f1 -> Ticket(18))
TicketManager - 3: Ticket issued: Ticket(19) for List(f2, f3)
TicketManager - 3: TICKETS: Map(Ticket(18) -> 6,
Ticket(19) -> 3)
TicketManager - 3: RES_TIC: Map(f5 -> Ticket(18),
f1 -> Ticket(18),
f2 -> Ticket(19),
f3 -> Ticket(19))
ProcessManager - 3: Invoke started: P2 eats
ProcessManager - 6: Invoke finished: P5 eats
TicketManager - 6: Ticket released: 18
TicketManager - 6: TICKETS: Map(Ticket(19) -> 3)
TicketManager - 6: RES_TIC: Map(f2 -> Ticket(19),
f3 -> Ticket(19))
ProcessManager - 3: Invoke finished: P2 eats
TicketManager - 3: Ticket released: 19
TicketManager - 3: TICKETS: Map()
TicketManager - 3: RES_TIC: Map()
TicketManager - 3: Ticket issued: Ticket(20) for List(f2, f3)
TicketManager - 3: TICKETS: Map(Ticket(20) -> 3)
TicketManager - 3: RES_TIC: Map(f2 -> Ticket(20),
f3 -> Ticket(20))
ProcessManager - 3: Invoke started: P2 eats
ProcessManager - 3: Invoke finished: P2 eats
TicketManager - 3: Ticket released: 20
TicketManager - 3: TICKETS: Map()
TicketManager - 3: RES_TIC: Map()
TicketManager - 3: Ticket issued: Ticket(21) for List(f2, f3)
TicketManager - 3: TICKETS: Map(Ticket(21) -> 3)
TicketManager - 3: RES_TIC: Map(f2 -> Ticket(21),
f3 -> Ticket(21))
ProcessManager - 3: Invoke started: P2 eats
ProcessManager - 3: Invoke finished: P2 eats
TicketManager - 3: Ticket released: 21
TicketManager - 3: TICKETS: Map()
TicketManager - 3: RES_TIC: Map()

Process finished with exit code 0
```

APPENDIX C

```
TicketManager - 2: Ticket issued: Ticket(7) for List(f1)
TicketManager - 2: TICKETS: Map(Ticket(7) -> 2)
TicketManager - 2: RES_TIC: Map(f1 -> Ticket(7))
TicketManager - 5: Ticket issued: Ticket(8) for List(f4)
TicketManager - 5: TICKETS: Map(Ticket(7) -> 2,
Ticket(8) -> 5)
TicketManager - 5: RES_TIC: Map(f1 -> Ticket(7),
f4 -> Ticket(8))
ProcessManager - 6: Conflict: My future resources
are locked:
List(f1) while asking for List(f5)
ProcessManager - 4: Conflict: My future resources
are locked:
List(f4) while asking for List(f3)
TicketManager - 3: Ticket issued: Ticket(9) for List(f2)
TicketManager - 3: TICKETS: Map(Ticket(7) -> 2,
Ticket(8) -> 5,
Ticket(9) -> 3)
TicketManager - 3: RES_TIC: Map(f1 -> Ticket(7),
f4 -> Ticket(8),
f2 -> Ticket(9))
TicketManager - 5: Ticket issued: Ticket(10) for List(f5)
ProcessManager - 2: Conflict: One of the requested resources
is locked:
List(f2) while asking for List(f2)
TicketManager - 5: TICKETS: Map(Ticket(7) -> 2,
Ticket(8) -> 5,
Ticket(9) -> 3,
Ticket(10) -> 5)
TicketManager - 5: RES_TIC: Map(f1 -> Ticket(7),
f4 -> Ticket(8),
f2 -> Ticket(9),
f5 -> Ticket(10))
ProcessManager - 5: Invoke started: Phil_4 eats
TicketManager - 3: Ticket issued: Ticket(11) for List(f3)
TicketManager - 3: TICKETS: Map(Ticket(8) -> 5,
Ticket(7) -> 2,
Ticket(10) -> 5,
Ticket(11) -> 3,
Ticket(9) -> 3)
TicketManager - 3: RES_TIC: Map(f2 -> Ticket(9),
f4 -> Ticket(8),
f3 -> Ticket(11),
f5 -> Ticket(10),
f1 -> Ticket(7))
ProcessManager - 3: Invoke started: Phil_2 eats
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5) while asking for List(f5)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3) while asking for List(f3)
ProcessManager - 2: Conflict: One of the requested resources
is locked: List(f2) while asking for List(f2)
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5) while asking for List(f5)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3) while asking for List(f3)
ProcessManager - 2: Conflict: One of the requested resources
is locked:
List(f2) while asking for List(f2)
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5) while asking for List(f5)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3) while asking for List(f3)
ProcessManager - 5: Invoke finished: Phil_4 eats
ProcessManager - 2: Conflict: One of the requested resources
is locked:
List(f2) while asking for List(f2)
ProcessManager - 3: Invoke finished: Phil_2 eats
TicketManager - 5: Ticket released: 10
TicketManager - 5: TICKETS: Map(Ticket(8) -> 5,
Ticket(7) -> 2,
Ticket(11) -> 3,
Ticket(9) -> 3)
TicketManager - 5: RES_TIC: Map(f2 -> Ticket(9),
f4 -> Ticket(8),
f3 -> Ticket(11),
f1 -> Ticket(7))
TicketManager - 5: Ticket released: 8
TicketManager - 5: TICKETS: Map(Ticket(7) -> 2,
Ticket(11) -> 3,
Ticket(9) -> 3)
TicketManager - 5: RES_TIC: Map(f2 -> Ticket(9),
f3 -> Ticket(11),
f1 -> Ticket(7))
TicketManager - 5: Ticket issued: Ticket(12) for List(f4)
TicketManager - 5: TICKETS: Map(Ticket(7) -> 2,
Ticket(11) -> 3,
Ticket(9) -> 3,
Ticket(12) -> 5)
TicketManager - 5: RES_TIC: Map(f2 -> Ticket(9),
f4 -> Ticket(12),
f3 -> Ticket(11),
f1 -> Ticket(7))
TicketManager - 3: Ticket released: 11
TicketManager - 3: TICKETS: Map(Ticket(7) -> 2,
Ticket(9) -> 3,
Ticket(12) -> 5)
TicketManager - 3: RES_TIC: Map(f2 -> Ticket(9),
f4 -> Ticket(12),
f1 -> Ticket(7))
TicketManager - 3: Ticket released: 9
TicketManager - 3: TICKETS: Map(Ticket(7) -> 2,
Ticket(12) -> 5)
TicketManager - 3: RES_TIC: Map(f4 -> Ticket(12),
f1 -> Ticket(7))
TicketManager - 3: Ticket issued: Ticket(13) for List(f2)
TicketManager - 3: TICKETS: Map(Ticket(13) -> 3,
Ticket(7) -> 2,
Ticket(12) -> 5)
TicketManager - 3: RES_TIC: Map(f2 -> Ticket(13),
f4 -> Ticket(12),
f1 -> Ticket(7))
TicketManager - 5: Ticket issued: Ticket(14) for List(f5)
TicketManager - 5: TICKETS: Map(Ticket(14) -> 5,
Ticket(13) -> 3,
Ticket(7) -> 2,
Ticket(12) -> 5)
TicketManager - 5: RES_TIC: Map(f2 -> Ticket(13),
f4 -> Ticket(12),
f5 -> Ticket(14),
f1 -> Ticket(7))
ProcessManager - 5: Invoke started: Phil_4 eats
TicketManager - 3: Ticket issued: Ticket(15) for List(f3)
TicketManager - 3: TICKETS: Map(Ticket(14) -> 5,
Ticket(15) -> 3,
Ticket(13) -> 3,
Ticket(7) -> 2,
Ticket(12) -> 5)
TicketManager - 3: RES_TIC: Map(f2 -> Ticket(13),
f4 -> Ticket(12),
f3 -> Ticket(15),
f5 -> Ticket(14),
f1 -> Ticket(7))
ProcessManager - 3: Invoke started: Phil_2 eats
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5) while asking for List(f5)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3) while asking for List(f3)
ProcessManager - 2: Conflict: One of the requested resources
is locked:
List(f2) while asking for List(f2)
ProcessManager - 6: Conflict: One of the requested resources
```

```
is locked:
List(f5) while asking for List(f5)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3) while asking for List(f3)
ProcessManager - 2: Conflict: One of the requested resources
is locked:
List(f2) while asking for List(f2)
ProcessManager - 6: Conflict: One of the requested resources
is locked:
List(f5) while asking for List(f5)
ProcessManager - 4: Conflict: One of the requested resources
is locked:
List(f3) while asking for List(f3)
ProcessManager - 2: Conflict: One of the requested resources
is locked:
List(f2) while asking for List(f2)
ProcessManager - 5: Invoke finished: Phil_4 eats
TicketManager - 5: Ticket released: 14
TicketManager - 5: TICKETS: Map(Ticket(15) -> 3,
Ticket(13) -> 3,
Ticket(7) -> 2,
Ticket(12) -> 5)
TicketManager - 5: RES_TIC: Map(f2 -> Ticket(13),
f4 -> Ticket(12),
f3 -> Ticket(15),
f1 -> Ticket(7))
TicketManager - 5: Ticket released: 12
TicketManager - 5: TICKETS: Map(Ticket(15) -> 3,
Ticket(13) -> 3,
Ticket(7) -> 2)
TicketManager - 5: RES_TIC: Map(f2 -> Ticket(13),
f3 -> Ticket(15),
f1 -> Ticket(7))
ProcessManager - 3: Invoke finished: Phil_2 eats
TicketManager - 3: Ticket released: 15
TicketManager - 3: TICKETS: Map(Ticket(13) -> 3,
Ticket(7) -> 2)
TicketManager - 3: RES_TIC: Map(f2 -> Ticket(13),
f1 -> Ticket(7))
TicketManager - 3: Ticket released: 13
TicketManager - 3: TICKETS: Map(Ticket(7) -> 2)
TicketManager - 3: RES_TIC: Map(f1 -> Ticket(7))
ProcessManager - 6: Conflict: My future resources
are locked:
List(f1) while asking for List(f5)
TicketManager - 4: Ticket issued: Ticket(16) for List(f3)
TicketManager - 4: TICKETS: Map(Ticket(16) -> 4,
Ticket(7) -> 2)
TicketManager - 4: RES_TIC: Map(f3 -> Ticket(16),
f1 -> Ticket(7))
TicketManager - 4: Ticket issued: Ticket(17) for List(f4)
TicketManager - 4: TICKETS: Map(Ticket(16) -> 4,
Ticket(17) -> 4,
Ticket(7) -> 2)
TicketManager - 4: RES_TIC: Map(f4 -> Ticket(17),
f3 -> Ticket(16),
f1 -> Ticket(7))
ProcessManager - 4: Invoke started: Phil_3 eats
TicketManager - 2: Ticket issued: Ticket(18) for List(f2)
TicketManager - 2: TICKETS: Map(Ticket(18) -> 2,
Ticket(16) -> 4,
Ticket(17) -> 4,
Ticket(7) -> 2)
TicketManager - 2: RES_TIC: Map(f2 -> Ticket(18),
f4 -> Ticket(17),
f3 -> Ticket(16),
f1 -> Ticket(7))
ProcessManager - 2: Invoke started: Phil_1 eats
ProcessManager - 6: Conflict: My future resources
are locked:
List(f1) while asking for List(f5)
ProcessManager - 6: Conflict: My future resources
are locked:
List(f1) while asking for List(f5)
ProcessManager - 6: Conflict: My future resources
are locked:
List(f1) while asking for List(f5)
ProcessManager - 4: Invoke finished: Phil_3 eats
TicketManager - 4: Ticket released: 17
ProcessManager - 2: Invoke finished: Phil_1 eats
TicketManager - 4: TICKETS: Map(Ticket(18) -> 2,
Ticket(16) -> 4,
Ticket(7) -> 2)
TicketManager - 4: RES_TIC: Map(f2 -> Ticket(18),
f3 -> Ticket(16),
f1 -> Ticket(7))
TicketManager - 4: Ticket released: 16
TicketManager - 4: TICKETS: Map(Ticket(18) -> 2,
Ticket(7) -> 2)
TicketManager - 4: RES_TIC: Map(f2 -> Ticket(18),
f1 -> Ticket(7))
TicketManager - 4: Ticket issued: Ticket(19) for List(f3)
TicketManager - 4: TICKETS: Map(Ticket(18) -> 2,
Ticket(19) -> 4,
Ticket(7) -> 2)
TicketManager - 4: RES_TIC: Map(f2 -> Ticket(18),
f3 -> Ticket(19),
f1 -> Ticket(7))
TicketManager - 2: Ticket released: 18
TicketManager - 2: TICKETS: Map(Ticket(19) -> 4,
Ticket(7) -> 2)
TicketManager - 2: RES_TIC: Map(f3 -> Ticket(19),
f1 -> Ticket(7))
```

```
TicketManager - 2: Ticket released: 7
TicketManager - 2: TICKETS: Map(Ticket(19) -> 4)
TicketManager - 2: RES_TIC: Map(f3 -> Ticket(19))
TicketManager - 2: Ticket issued: Ticket(20) for List(f1)
TicketManager - 2: TICKETS: Map(Ticket(19) -> 4,
Ticket(20) -> 2)
TicketManager - 2: RES_TIC: Map(f3 -> Ticket(19),
f1 -> Ticket(20))
TicketManager - 4: Ticket issued: Ticket(21) for List(f4)
TicketManager - 4: TICKETS: Map(Ticket(19) -> 4,
Ticket(20) -> 2,
Ticket(21) -> 4)
TicketManager - 4: RES_TIC: Map(f4 -> Ticket(21),
f3 -> Ticket(19),
f1 -> Ticket(20))
ProcessManager - 4: Invoke started: Phil_3 eats
TicketManager - 2: Ticket issued: Ticket(22) for List(f2)
TicketManager - 2: TICKETS: Map(Ticket(19) -> 4,
Ticket(22) -> 2,
Ticket(20) -> 2,
Ticket(21) -> 4)
TicketManager - 2: RES_TIC: Map(f2 -> Ticket(22),
f4 -> Ticket(21),
f3 -> Ticket(19),
f1 -> Ticket(20))
ProcessManager - 2: Invoke started: Phil_1 eats
ProcessManager - 6: Conflict: My future resources
are locked:
List(f1) while asking for List(f5)
ProcessManager - 6: Conflict: My future resources
are locked:
List(f1) while asking for List(f5)
ProcessManager - 6: Conflict: My future resources
are locked:
List(f1) while asking for List(f5)
ProcessManager - 4: Invoke finished: Phil_3 eats
TicketManager - 4: Ticket released: 21
TicketManager - 4: TICKETS: Map(Ticket(19) -> 4,
Ticket(22) -> 2,
Ticket(20) -> 2)
TicketManager - 4: RES_TIC: Map(f2 -> Ticket(22),
f3 -> Ticket(19),
f1 -> Ticket(20))
TicketManager - 4: Ticket released: 19
TicketManager - 4: TICKETS: Map(Ticket(22) -> 2,
Ticket(20) -> 2)
TicketManager - 4: RES_TIC: Map(f2 -> Ticket(22),
f1 -> Ticket(20))
ProcessManager - 2: Invoke finished: Phil_1 eats
TicketManager - 2: Ticket released: 22
TicketManager - 2: TICKETS: Map(Ticket(20) -> 2)
TicketManager - 2: RES_TIC: Map(f1 -> Ticket(20))
TicketManager - 2: Ticket released: 20
TicketManager - 2: TICKETS: Map()
TicketManager - 2: RES_TIC: Map()
TicketManager - 6: Ticket issued: Ticket(23) for List(f5)
TicketManager - 6: TICKETS: Map(Ticket(23) -> 6)
TicketManager - 6: RES_TIC: Map(f5 -> Ticket(23))
TicketManager - 6: Ticket issued: Ticket(24) for List(f1)
TicketManager - 6: TICKETS: Map(Ticket(23) -> 6,
Ticket(24) -> 6)
TicketManager - 6: RES_TIC: Map(f5 -> Ticket(23),
f1 -> Ticket(24))
ProcessManager - 6: Invoke started: Phil_5 eats
ProcessManager - 6: Invoke finished: Phil_5 eats
TicketManager - 6: Ticket released: 24
TicketManager - 6: TICKETS: Map(Ticket(23) -> 6)
TicketManager - 6: RES_TIC: Map(f5 -> Ticket(23))
TicketManager - 6: Ticket released: 23
TicketManager - 6: TICKETS: Map()
TicketManager - 6: RES_TIC: Map()
TicketManager - 6: Ticket issued: Ticket(25) for List(f5)
TicketManager - 6: TICKETS: Map(Ticket(25) -> 6)
TicketManager - 6: RES_TIC: Map(f5 -> Ticket(25))
TicketManager - 6: Ticket issued: Ticket(26) for List(f1)
TicketManager - 6: TICKETS: Map(Ticket(25) -> 6,
Ticket(26) -> 6)
TicketManager - 6: RES_TIC: Map(f5 -> Ticket(25),
f1 -> Ticket(26))
ProcessManager - 6: Invoke started: Phil_5 eats
ProcessManager - 6: Invoke finished: Phil_5 eats
TicketManager - 6: Ticket released: 26
TicketManager - 6: TICKETS: Map(Ticket(25) -> 6)
TicketManager - 6: RES_TIC: Map(f5 -> Ticket(25))
TicketManager - 6: Ticket released: 25
TicketManager - 6: TICKETS: Map()
TicketManager - 6: RES_TIC: Map()

Process finished with exit code 0
```

BIBLIOGRAPHY

[1] Tiresias organization. http://www.tiresias.org/about/history.htm, 1980.

[2] Duke project. http://smarthome.duke.edu/, 1995.

[3] Tiresias organization. http://www.tiresias.org/research/researchers/projects/ami/askit.htm, 2004.

[4] DAFFIE project. scv.bu.edu/visualization/daffie/index.html, 2008.

[5] SketchUp project. sketchup.google.com, 2008.

[6] Netcarity organization. http://www.netcarity.org/About.11.0.html, 2009.

[7] ViSi demo. www.sm4all-project.eu/index.php/activities/videos.html, 2009.

[8] M. Aiello. The Role of Web Services at Home. In *IEEE Web Service-based Systems and Applications (WEBSA)*, 2006.

[9] M. Aiello and S. Dustdar. Are our homes ready for services? A Domotic Infrastructure based on the Web Service Stack. *Pervasive and Mobile Computing*, 4(4):506–525, 2008.

[10] Akka. http://akka.io/docs/akka/1.2-RC3/.

[11] Akka. http://akka.io/docs/akka/1.2-RC3/java/untyped-actors.html.

[12] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services. Concepts, Architectures and Applications*. Springer, 2004.

[13] BPEL. Business process execution language for web services. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[14] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

[15] W. K. Chan Heng Lu and T. H. Tse. Static slicing for pervasive programs. In *6th International Conference on Quality Software (QSIC 2006)*, pages 185–192, 2006.

[16] Yu Huang, Xiaoxing Ma, Jiannong Cao, Xianping Tao, and Jian Lu. Concurrent event detection for asynchronous consistency checking of pervasive context. In *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications*, pages 1–9, Washington, DC, USA, 2009. IEEE Computer Society.

[17] JSON. http://http://www.json.org/.

[18] A. Lazovik, E. Kaldeli, E. Lazovik, and M. Aiello. Planning in a smart home: Visualization and simulation. In *19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 2009.

[19] E. Lazovik, P. den Dulk, M. de Groote, A. Lazovik, and M. Aiello. Services inside the smart home. a simulation and visualization tool. In *International Conference on Service Oriented Computing (ICSOC 2009)*, 2009.

[20] M. Papazoglou. *Web Services: Principles and Technology*. Prentice Hall, 2008.

[21] SM4ALL project. EU STREP Project FP7-224332 Smart Homes for All. www.sm4all-project.eu, 2008.

[22] SoftCraft. Btrieve. http://en.wikipedia.org/wiki/Btrieve, 2008.

[23] RDP support. Pervasive sql client for citrix server. http://support.resortdata.com/PervasiveSales/Pervasive.htm, 2010.

[24] SOAP v1.2. Simple object access protocol. http://www.w3.org/TR/soap12-part1.

[25] Wikipedia. http://en.wikipedia.org/wiki/Event-driven_programming.

[26] Wikipedia. http://en.wikipedia.org/wiki/Dining_philosophers_problem.

[27] WS-Transaction. http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html.