THESIS ON INFORMATICS AND SYSTEM ENGINEERING C80

Whiteboard Architecture for the Multi-agent Sensor Systems

ENAR REILENT

TUT PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science

Dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer Science on 19 November, 2012

Defence of the thesis: 18 December 2012

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any academic degree.

/Enar Reilent/

European Union
European Social Fund

Investing in your future

ISSN 1406-4731
ISBN 978-9949-23-408-0 (publication)
ISBN 978-9949-23-409-7 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C80

Tahvelarhitektuur multi-agent sensorsüsteemide jaoks

ENAR REILENT

TTÜ
KIRJASTUS

ABSTRACT

This thesis investigates software architectures for multi-agent sensor systems. Multi-agent systems are considered both in the context of robotics and the context of personalized telecare systems. The goal of the work is to develop a software architecture which combines the desired properties of flexibility and developer-friendliness with efficiency and scalability.

The basic approach taken for building such architecture is to use the classical blackboard principles with the new flavor we call *whiteboard* and combine these with the RDF-based approach for knowledge representation. The desired combination of flexibility and efficiency is achieved by taking specific architectural choices suitable for the domain and introducing numerous detailed improvements to the basic approach.

In particular, we take a content-centric approach in the sense that the architecture is designed to support universal semantically described context information and formal reasoning for automated profile generation and data aggregation. The approach is implemented both in a multi-robot system and several home telecare systems.

We compare the efficiency of selected components to a variety of alternatives and consider several options for knowledge representation. We argue that the choices and optimizations presented are suitable for a wide range of application domains for multi-agent sensor systems.

KOKKUVÕTE

Doktoritöö teemaks on multi-agent sensorsüsteemid, mida käsitletakse nii robootika kui personaliseeritud telemeditsiini kontekstis. Töö eesmärgiks on luua selline tarkvara arhitektuur, kus süsteemi paindlikkus ja arendajasõbralikkus oleks kombineeritud efektiivsuse ja skaleeruvusega.

Soovitud arhitektuuri loomise aluspõhimõtteks on kasutada klassikalist nn *blackboard*-arhitektuuri uues vaates, mida kutsume *whiteboard* arhitektuuriks ja kombineerida seda teadmiste esitamise RDF-põhise lähenemisega. Paindlikkuse ja efektiivsuse kombinatsioon saavutatakse vaadeldavate rakendusvaldkondadega sobivate spetsiifiliste arhitektuursete valikutega ja aluspõhimõtetele mitmesuguste täienduste sisseviimisega.

Töö lähenemisviis arhitektuurile on sisu-keskne selles mõttes, et arhitektuur sobib universaalse, semantiliselt kirjeldatud konteksti-informatsiooni ja formaalsete järeldusmeetoditega automaatseks profiili-genereerimiseks ja andmete agregeerimiseks. Lähenemisviis on realiseeritud nii multi-robot süsteemis kui mitmes kodukasutuseks mõeldud telemeditsiini-süsteemis.

Me võrdleme valitud komponentide efektiivsust mitme eri tehnilise realisatsioonivariandi ja mitme eri teadmiste esitamise meetodi vahel. Kokkuvõttes näitame, et töös esitatud valikud ja optimeeringud on sobivad multi-agent süsteemide laia rakendusvaldkondade spektri jaoks.

ACKNOWLEDGEMENTS

First of all, I wish to express my highest gratitude to my supervisor Prof. Tanel Tammet for the motivation, encouragement and all-around guidance through all these years. This thesis would not have come to existence without him.

The thesis is mostly based on the work done in two projects. All what concerns the robot's middleware has its foundations in the Roboswarm project of the EU's 6th Framework Program (FP6). The study in the field of telemonitoring and telemedicine has been supported by the research project of the Eliko Competence Centre.

I would like to thank all these people who have contributed to the process leading to the completion of the given work. Special appreciation to Alar Kuusik for all kinds of support with hardware issues and philosophical discussions on data encoding; Madis Puju for implementing, providing technical support and tuning the Roboswarm's database – this is where it all began; Andres Puusepp for providing the custom made simulator for the robot; Carlos Marques for mechanical and structural help on the robot; Janne Haverinen for the initial concept of the dispatcher module and low-level software for sensor devices; Tanel Tammet also for the assistance with the reasoning engine on the robot and originating the Wgandalf database, which plays important role in this work.

Also, I acknowledge Marko Parve for providing all sorts of medical measurement devices; Prof. Jüri Vain for counseling on Prolog issues; Priit Järv for providing technical support and fixing bugs upon my complaints in Wgandalf issues; Ivor Lõõbas for the heated arguments in the early days of the telemonitoring project; Margo Kopli for explaining the numerous versions of the interfaces of the server side software which my telemonitoring home gateway device had to interact with; and other colleagues from Eliko.

I must apologize to all my family members for not being able to spare my time with them during the past months when composing the thesis. I sincerely thank everybody for their tolerance and patience.

CONTENTS

ABSTRACT 5
KOKKUVÕTE 6
ACKNOWLEDGEMENTS 7
ARTICLES PUBLISHED BY THE THESIS AUTHOR 10
1. INTRODUCTION 12
1.1. Motivation and the problem statement 12
1.2. Contribution of the thesis 14
1.3. Thesis organization 15
2. INTRODUCTION TO WHITEBOARD 16
2.1. Example systems – a robot and a telemonitoring gateway 17
2.2. Generalization, presumptions, refining requirements 20
2.3. Related work on blackboards and alternatives 21
2.4. Related work on agents 24
3. WHITEBOARD DESIGN DETAILS 25
3.1. Data model 26
3.2. The underlying medium and tools 30
3.3. API details for agents 32
3.4. Processing with the reasoner 35
4. THE IMPLEMENTATIONS AND EXPERIMENTS 37
4.1. Roboswarm 37
4.2. Telemonitoring 42
5. CONCLUSIONS 47
5.1. Advantages of the whiteboard solution 48
5.2. Summary of the contributions of thesis 49
5.3. Authors' contribution in the published articles 49
5.4. Future work 50

REFERENCES 51
Appendix 1 - Interprocess communication 56
Appendix 2 – Scalability of tuples and triples 63
Elulugu 66
Curriculum Vitae 67
PAPER 1 69
PAPER 2 81
PAPER 3 89
PAPER 4 95
PAPER 5 101
PAPER 6 113
PAPER 7 119
PAPER 8 127

ARTICLES PUBLISHED BY THE THESIS AUTHOR

1. J. Vain, T. Tammet, A. Kuusik, E. Reilent. Software Architecture for Swarm Mobile Robots. BEC2008
2. T. Tammet, J. Vain, A. Puusepp, E. Reilent, A. Kuusik. RFID-based communications for a self-organizing robot swarm. In: Proceedings Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2008: 20-24 October 2008, Venice, Italy: (Toim.) Brueckner, Sven; Robertson, Paul; Bellur, Umesh. Los Alamitos, Calif.: IEEE Computer Society, 2008, 45 - 54.
3. T. Tammet, E. Reilent, M.Puju, A. Puusepp, A. Kuusik, A. Knowledge centric architecture for a robot swarm. In: 7th IFAC Symposium on Intelligent Autonomous Vehicles (2010). IFAC-PapersOnLine, 2010, (Intelligent Autonomous Vehicles; 7/1). 2010.
4. E. Reilent, I. Lõõbas, R. Pahtma, A.Kuusik. BEC2010, Medical and Context Data Acquisition System for Patient Home Monitoring. 2010.
5. A. Kuusik, E. Reilent, I. Lõõbas, M. Parve. Semantic Formal Reasoning Solution for Personalized Home Telecare. In: Proceedings of 2010 International Conference on Mechanical and Electrical Technology (ICMET2010): 2010 International Conference on Mechanical and Electrical Technology, Singapore, September 10-12, 2010. (Toim.) Dr. Parvinder et.al. Chengdu, China: IEEE Operations Center, 2010, 72 - 76.
6. A. Kuusik, E. Reilent, I. Lõõbas, A. Luberg, A. Data Acquisition Software Architecture for Patient Monitoring Devices. Journal of Electronics and Electrical Engineering, Kaunas University, 105(9), 97 - 100. 2010.
7. A. Kuusik, E. Reilent, I. Lõõbas, M. Parve. Software architecture for modern telehome care systems. In: Proceedings of the 6th International Conference on Networked Computing (INC 2010): 6th International Conference on Networked Computing, Gyeongju, Korea, May 11-13, 2010. (Toim.) Dr. Chun Yuan, Dr. Li-Shiang Tsay, Dr. Fei-Yue Wang and others. IEEE Computer Society Press, 2010, (IEEE Conference Record number 16757, CFP1084J-ART), 326 - 331.
8. I. Lõõbas, E. Reilent, A. Anier, A. Luberg, A. Kuusik. Towards semantic contextual content-centric assisted living solution. In: Proceedings of 12th IEEE International Conference on e-Health Networking Applications and Services (Healthcom 2010): 12th IEEE International Conference on e-Health Networking Applications and Services, Lyon 1-3 July 2010. IEEE Operations Center, 2010, (1; 1), 56 - 60.
9. E. Reilent, A. Kuusik, I. Lõõbas, P. Ross, P. Improving the data compatibility of PHR and telecare solutions. In: 5th European Conference of the International Federation for Medical and Biological Engineering 14 - 18 September 2011, Budapest, Hungary: (Toim.) Jobbágy, Á.. Springer, 2011, (IFMBE Proceedings; 37), 925 - 928.

10. E. Reilent, I. Lõõbas, A. Kuusik, M. Parve, P. Ross. Extendable Data Model for Universal Health Records. AMA-IEEE Medical Technology Conference, Boston, 16-18 October 2011. IEEE, 2011, 1 - 2.
11. A. Kuusik, E. Reilent, I. Lõõbas, M. Parve. Software architecture for modern telehealth care systems. Journal of Advances on Information Sciences and Service Sciences, 2011, 3(2), 141 - 151.
12. E. Reilent, A. Kuusik, M. Puju. Real-time data streaming for functionally improved eHealth solutions. 2012,International Conference on Biomedical and Health Informatics (BHI2012), Hong Kong and Shenzhen, China, 2-7
13. A. Kuusik, E. Reilent, K. Sarna, M. Parve. Home telecare and rehabilitation system with aspect oriented functional integration. The 46th annual conference of the German Society for Biomedical Engineering, Jena, Germany, September 17-19, 2012.
14. A. Puusepp, T. Tammet, M. Puju, E. Reilent. Robot movement strategies in the environment enriched with RFID tags. 16th International Conference on System Theory, Control and Computing, Sinaia, Romania, 12-14 October 2012.

1. INTRODUCTION

This thesis concerns the software architecture of systems incorporating several different sensor and monitoring systems and capable of taking autonomous action when need arises.

The goal of our work is to develop a software architecture which combines the desired properties of robustness, modularity and flexibility with efficiency and scalability on the other hand. An important aspect of the systems is built-in intelligence in the embedded devices themselves (robots, medical sensor systems, etc.): the basic decisions about the necessary actions and the information to be transferred are taken already at the device level, not just on the level of the central server of the whole multi-agent system. We ground our approach on the use of *whiteboard* and employ both the multi-robot and the personalized telecare systems as the chief domains of our experimental implementations.

The thesis contributes to the area of whiteboard-focused architectures for multi-agent sensor systems based on the articles added to the Appendix.

1.1. Motivation and the problem statement

When creating the systems that involve some independently running components where several activities have to be performed in parallel, it is very natural to use a multi-process architecture. Although it might be feasible and even beneficiary - easier to optimize for performance - to use the monolithic approach, the decomposition of the problem into separate sub-tasks and encapsulating these to dedicated modules gives better maintainability. No less important is the comprehensibility of the model to the developers in the first place.

However, based on the details of the planned system there are several possibilities to create a multi-process architecture. Besides dividing the system functionality among the processes one must decide how to make the processes interact with each other. It is not a trivial issue, especially on the level of implementation, as the communication mechanism adds new complexities and an extra load. Depending on the posed requirements (e.g. security, speed, flexibility, traceability) to the system and its parts there are numerous possible solutions for the organization of synchronization and data passing between the processes, some more and some less efficient.

An important issue to consider here is the kinds of data flows between the processes. In case the multipoint to multipoint communications are common (the data provided by one process is needed by several consumers and one consumer is interested in the output of several data providers) as it happens, for example in the sensor data acquisition and actuator controlling systems, then the *blackboard* architecture would be a natural design choice. The classical *blackboard* systems [1], [2], [3] originated as an architecture for problem solving: the problem is written on the blackboard, the system runs and adds new data on the blackboard until it produces the solution.

The blackboard allows to avoid the message passing frameworks: the system can be set up with a shared medium – a global database – where all the system components upload and access all the data they need.

The basic idea of the *blackboard* [1] system is very simple, but the implementation details are rather not. There are three components to consider: the knowledge sources, the blackboard and the control mechanism: the scheduler. The knowledge sources (KSs) solve the given problem collaboratively by applying their expertise on the present state of the problem. KS do not interact directly with one another or know what other specific KSs are present in the system [2]. They add their contribution to the shared blackboard as well as get their input by reading it. The blackboard itself is just a storage. The scheduler is where the real complexity stems from, as it has to drive the whole system towards the solution. It has to manage issues like which KSs can be executed at the given state; if many competing executions possible, which is the most helpful; preventing deadlocks. This poses challenges to the implementation even if the tasks are understood in detail.

In the blackboard systems discussed later in this work we do not strictly follow the classical design ideas. To avoid confusion we will make a difference between the official *blackboard* and the similar concept we will call hereafter *whiteboard*.

In the scope of this work we will define the *whiteboard* to be a shared database for agents to be used as a communication channel. The whiteboard thus has dual functionality, both of which are important and require corresponding architectural choices.

As seen later, the blackboards and the whiteboards have slightly different purposes. Whiteboard is used as a name because it carries linguistically the same meaning and in a sense the whiteboard system under discussion is a lightweight version of the blackboard system. Another fact in favor of the term *whiteboard* is the circumstance that Wgandalf[1] was mostly used as a base for building the whiteboards in the implemented systems. If generalized enough they can be taken as synonyms. It is also more appropriate to call processes *agents* instead of *knowledge sources.*

The whiteboard model poses several important questions which are tackled in the scope of this thesis.

First, there is the issue of a data model. The relational data model with a fixed schema is clearly a very straightforward way to build the whiteboard, but the poor extendability is the main shortcoming of the relational model [4], [5]. Hence we focus on RDF or RDF-like schemas and consider the aspects and suitable answers for data modelling in the distributed sensor/actuator systems. The efficiency of different data models is presented and compared in the experiments in the appendix of the thesis.

[1] Wgandalf is the reincarnation of the Gandalf system where the W refers to *white*

One of the attractive benefits of using the whiteboard is the possibility to use reasoning engines using the whole set of data as input and writing output to the same whiteboard, similarly to the classical blackboard systems. We will consider and experiment with several options, bring out the benefits, problems and questions needing further work. The option of using reasoning engines is closely tied to the question of finding suitable data modelling principles.

The issue of underlying tools comes next. A wide range of available options (sockets, files, databases, shared memory, etc.) is described, measured and compared in the appendix.

Yet another issue is the design of the frontend, including data presentation and query languages, the prototypes of the API functions and access policies. The whiteboard should not be very restrictive, but should maintain order and avoid mistakes.

1.2. Contribution of the thesis

The thesis studies questions and options arising while designing architectures for multi-agent systems of intelligent sensors. The focus of research is on the details of using a *whiteboard* for intelligent communication between software agents on a device.

The design decisions of a whiteboard affect the way the agents are created and the final application is put together. The control functionalities still exist, even if not explicitly, and should be divided reasonably between the whiteboard and the agents.

Our main goal is to design architecture for the multi-agent sensor systems which would be:

- *robust*: (temporal) failures of some agents should not bring the system down,
- *modular and flexible*: it should be easy to add new types of sensor and reasoning/data processing agents with new types of data,
- *efficient*: the architecture should strive for enabling maximal possible efficiency, i.e. being more efficient than other robust and modular architectures.

The general idea of building multi-process systems with ambitious data sharing around the central whiteboard is shown to be promising. We allow the system to be built agent by agent, keeping the agents' responsibilities clear and routing all workflows through the whiteboard.

The cornerstone of our solution is to employ enhanced RDF-style knowledge representation on the whiteboard, allowing good data interoperability, ontology-based formal reasoning and sufficient independence and flexibility for separate agents and software developers while achieving high efficiency at the same time.

The designed architecture is implemented and optimized on large combined hardware/software projects in two significantly different domains, demonstrating the feasibility of the approach in practice.

1.3. Thesis organization

The thesis starts with the general introduction, motivation and the contribution overviews above.

The next part of the thesis presents an overview of the whiteboard: the principles and the main differences between the classical blackboard and the whiteboard as described in the current work. The two large projects where our whiteboard architecture was designed and tested are presented next. The final chapters of this part give an overview of the related work in blackboard and similar systems, as well as multi-agent systems.

The third part of the thesis focuses on the details of whiteboard: architectural options, motivations and reasons of using one or another option. We will consider and explain our choices regarding the data model, the underlying communication model, the API details and processing the data with the reasoning engine.

The fourth part describes the requirements, details and differences of our solutions for the two different large research projects: the Roboswarm and the Telemonitoring project.

The conclusions chapter summarizes the advantages of the whiteboard architecture, our contributions in papers and gives suggestions for future work in the topic.

The two appendixes present our experimental results of measuring and comparing the performance of different alternative mechanisms for interprocess communications and data represention.

The rest of thesis consists of the eight selected publications from the full list of fourteen.

2. INTRODUCTION TO WHITEBOARD

The classical blackboard systems [1], [3] were proposed as architecture for problem solving. In the initial state the problem is written to the blackboard. After that the system will run until it produces the solution. The control mechanism has the key role, with the blackboard component itself having no significant importance. Opportunistic KSs are rather like procedures that can be called by the control mechanism to perform some certain action. There is no parallelism to avoid data access conflicts. To choose between KSs they are tested first. In the test a KS reports whether it has enough input data to run successfully or what are the missing items. It also predicts its outcome it should produce in the case of given a chance to execute. Sometimes the test management is implemented as a part of the KS and sometimes on the side of the control mechanism. However, we could still say that these control issues are basically the matters of blackboard design.

Keeping in mind the sensor data acquisition systems which run for a longer period of time and are not directly meant for problem solving, the blackboard system in its classical sense is not exactly what is needed. Rather we could think of the whiteboard as an environment that provides means for sustaining constantly running agents around a common medium. Agents are meant to run in parallel and without any special scheduler or testing mechanism. The whiteboard should be simply the framework for all agents for communicating with each other which enables the workflows. If someone writes a piece of knowledge to the whiteboard, it can be read by either one interested agent or by everybody or by no one – the data provider does not have to be concerned with the existence of possible consumers. By slightly elaborating this idea we can say that the agent itself can be the addressee or consumer of its own data. The whiteboard can be used just as a data storage. Therefore the whiteboard in the scope of this work has two primary roles:

- a communication channel between agents,
- a database

By saying *a database* here we do not mean classical database management systems with a database server, complex query handling and long term storage of large amounts of data. Our *database* is just a temporary storage for variables, parameters, configurations and results without guaranteeing ACID[2]. Parallels can be drawn to the so called NoSQL databases as discussed in [4], [5], and [6] but we do not share exactly the same goals. Here the idea of a database is also important because it stresses the effect of history. Communication and message passing by default do not include history. As an agent regularly writes some values (e.g. sensor readings) to the whiteboard and these are not immediately deleted, there will be a buildup of historical values available for using over and over again by other agents, until the values are deleted. All in all, we may say that an agent does

[2] ACID - atomicity, consistency, isolation, durability

not have to get all its input by messages from the other agents, but it can search the database instead.

As stated before, the whiteboard agents should be rather seen as constantly running processes and not temporarily invoked KSs, even if their tasks can be the same. However, this is an opportunity, not the obligation. If the system's design requires starting and stopping new agents during runtime, it should definitely be possible in the whiteboard system. The data on the whiteboard remains intact and available for further uses. Hence the whiteboard offers the possibility for the agents to keep their states in the public storage and in the case of being stopped (or crash) to continue from the last state when restarted.

By allowing only sequential execution of KSs the blackboard systems did not have to deal with conflict situations where several agents update a single value concurrently. Whiteboard's agents run in parallel which is sensible from the application's point of view. It must be noted that also in the case of whiteboard the previously mentioned complexities of synchronization cannot be overlooked. We do not have the control (scheduler) as such, which means that the conflicts have to be solved somewhere else – in the whiteboard's implementation and in the whiteboard's API design.

2.1. Example systems – a robot and a telemonitoring gateway

For getting a better understanding of the requirements for the whiteboard we can examine two real life case studies. Both of these examples are sensor data acquisition systems with actuators and built-in decision making. First there is the Roboswarm project where the whiteboard is used as the middleware for controlling a simple mobile robot with a small number of sensors. The second system is the middleware for the home telemonitoring gateway device. The general idea of the systems is exactly the same: get data from the sensor devices, deliver it to the decision making modules, analyze the data and probably drive some actuators. When we take a close look at the utilization of the whiteboards in these systems we can spot several differences.

In the Roboswarm project had a set of identical mobile robots (based on the iRobot Roomba) and a server. The robots were fairly simple in their design. The equipment included some sensor devices (e.g. bumpers, cliff sensors, sonars, wheel odometers, and magnetometer), some actuator devices (e.g. wheel motors, antenna multiplexer, LED indicators) and devices which are both sensors and actuators at the same time (RFID reader). On the logical level the wheels and bumpers are totally independent entities. However, on the physical level they must be handled together as they are connected through the same cable and protocol. There was also a network device (WiFi). The server hosted the user interface system and the task allocation mechanism.

Although a typical approach at that time would have been to use Player/Stage it was decided to go with the whiteboard model (implemented as a fast shared memory data store). The agents were created based on the physical hardware devices because it was not realistic to share open connections among the processes. As noted, there could be many logical entities behind one connection, but as the devices were using serial or USB ports and had proprietary access libraries and initialization procedures, only one process interacted with one device. The sensor readings were sent to the whiteboard. All the commands were also sent to the whiteboard. The corresponding actuator agent picked them up from the whiteboard and delivered to the real device. If the devices used a bitwise protocol then a higher level representation was used in the whiteboard and the communicating agent performed the conversions.

The whiteboard itself could have two types of entries: regular and persistent. Regular entries disappeared by themselves after a while because the whiteboard was a circular buffer and old data got overwritten. That means no special garbage collection was needed. For optimization reasons some exceptions were needed and thus the persistent data entries did not follow the same pattern. The data entries were rather simple and short, following a set of common principles. The possibility to easily add new sensor devices and data types was highly important. There was not much need for complex data structures, but good performance (15 ms sensor polling time) and small footprint of code were crucial (Gumstix Verdex was the controller). The data model used throughout the system was RDF triples with additional meta-fields. Besides the robots and the server the RFID tags could also store up to six RDF-like entries 32 bytes each (see Paper 1). The whiteboard's API was rather low-level due to the tradeoff between convenience and high performance, yet it had to be easy to comprehend.

The telemonitoring gateway project introduces a set of medical measurement devices to the home of a patient so that she can regularly check her health parameters and report these to hospital. According to [7], recent surveys reveal that elderly individuals prefer to remain in their accustomed living environment for as long as possible, even in the eventuality of increasing reliance upon assistance services and caregivers. Another target group is the patients with chronic diseases are subjects to monitoring activities of daily living (ADLs), vital signs and self-reporting [8]. For these kinds of use cases a gateway device is inevitable due to the fact that measurement devices have local USB or Bluetooth (Zigbee, 6Lowpan) connectivity and the data has to get to the hospital automatically. Manual transfer with paper and pen has certain drawbacks ([9]). In addition to the sensors (blood pressure monitor, ECG device, glucometer, scale, motion tracking accelerometers/gyroscopes etc.) there exist actuators for patient notification and guidance (display screen, loudspeakers). The gateway talks to the devices with their low-level proprietary protocols (still preferred by many manufacturers [7]) and forwards the measurements to the hospital system, but it does some local processing as well. Not all the data should be sent to the hospital (which can become a bottleneck if too many patients) and therefore the gateway does some computing locally, like aggregation, filtering, threshold checking and quick patient feedback. Some measurements are not taken from the devices but are input through the UI by the patient as a self-reporting evaluation of pain level, stress, mood, breathlessness and tiredness [8], calories intake (for diabetes patient, extremely hard to automate, e.g. [10] uses crowdsourcing for photo analyses) etc. All sensors do not have to measure the patient: the environment can be monitored as well, like ambient temperature, noise level, and humidity. Sensors can be *virtual* [11] meaning that they combine outputs of other data sources.

The hardware used for running the gateway application is not a powerful PC but an embedded computer: set-top box (Papers 3, 4) or media display (8" Chumby), equipped with a network connector and all the necessary interfaces for sensor connectivity. There are projects that use a PC at the home setup [12] or use many computing devices [7] (a gateway plus a set-top box) which is a serious problem for cost efficiency. For the majority of cases the fast performance is not really important, because measurements are taken only a couple of times per day and having a reaction time in seconds is acceptable enough. For the home monitoring system the main focus is put to data structures and API. Compared to the robot's case one measurement event of telemonitoring may contain more details (fields) and have a deeper structure. For example, a record of a blood glucose sample has to include the indication of when the measurement was taken: before or after the meal, and what meal it was (e.g. breakfast, snack). Parallel use of terminologies is possible – "blood pressure" entity should contain the name used in the hospital systems as well as the name from some global medical nomenclature. Ontologies have a more important role here and there should be readiness for supporting several of them simultaneously (e.g. as in HL7 v3 [13], [14]). The system has to cope with various kinds of data. Some measurements have one clear output (for example, weight of a person), some have one sample per second for a short period of time (as pulse and SpO_2). The ECG measurement (Paper 6) event is a signal with one thousand samples per second (possibly more than one signal[3]). Another contrast with the robot's case is the longevity of the data items. In Roboswarm the whiteboard records had typically meaning only for tens of milliseconds and at most for the duration of the task (tens of minutes). In medical monitoring some data must be available and interpretable tens of years later to check for long term trends which again advocates for good annotations.

As the system is neither well defined nor complete, the ideas of flexibility and extendibility are quite important. It must be as easy as possible to introduce new sensor devices for known parameters and add new types of sensors to measure new parameters. Patients with different diseases need different types of equipment and it should be possible to switch off all the unneeded functionalities without disturbing anything else. The ease of adding or removing devices is thus even more important than in the robot case. In general, the system can contain a lot more agents, both for device drivers and decision making. Thus the fine grained and easy to use API would prove to be useful.

[3] If it is 12-lead or there can be extra signals, e.g. accelerometers

2.2. Generalization, presumptions, refining requirements

While trying to settle down the essence of the whiteboard in the scope of this work it becomes clear that it is a shared database for agents to be used as a communication channel. In addition to this functionality there are several key factors which need to be constrained. How exactly do we want the whiteboard to be designed – which responsibilities should be left to the agents and which to the whiteboard's API? As seen from the example cases it depends on the problem domain. The expectations for the whiteboard's functionality can be mismatching, for example fast performance and comfortable high-level API calls for data handling.

It is good to keep in mind that all of the target hardware platforms have had limited processing capabilities. Therefore the whiteboard's design and implementation has to be lightweight enough for fitting onto embedded computers. We exclude microcontrollers without a proper OS. Linux (e.g. Busybox) environment is deemed necessary. But still, as it is an embedded system, there are typically no high level tools, programming languages or libraries available. Issues that are not noticed on the regular multicore PCs can become critical here, especially when time constraints are important for the application. We have to maintain balance between the footprint of the whiteboard's code and the benefits granted for the agents.

We consider the whiteboard spanning one physical machine only, to the contrary of the principles of NoSql databases [4]. The mentioned sensor data collecting systems were not heavily distributed – lots of the processes run on the small number of machines. The processes running on one machine do much more interaction among themselves than with the processes residing on different machines, as noticed by [15]. The complete system spans over several machines (robots, servers) but the processes of the different machines do not share a whiteboard. Either they have their private instances or do not have a whiteboard at all. In the example applications we have had communication agents who picked up the send-worthy information from their local whiteboards and transmitted it to some other machine. Another possible choice would have been to build a whiteboard with inter-machine connectivity, as a usual database engine, but the performance considerations and lack of need decided the matter.

It would be nice to have the same data format in every level of the system, but it is not always achievable. As said, since most of the inter-agent traffic goes on inside one machine, we will concentrate on that case. For example, if the server is built by different parties and features its own data format, it does not automatically mean that the whiteboard on a sensor gateway has to use that format which typically is not compatible with the other requirements of the whiteboard. For example, large XML documents with redundant fields on the server are not the best option for storing high frequency sensor data on the gateway. Hence we use converters. The set of local agents should just prepare all the data needed to make coherent conversions possible in the future. Thus the whiteboard is free to use its own data format. However, due to simplicity, clarity, and limited resources we allow only one format for all the agents, fixing it in the whiteboard's API.

The specification of the system is not known beforehand: the system is in a constant state of evolution. The discussed measurement collecting systems had to be open to new sensors and new business logic. We have in mind the extensibility described by [16] – new knowledge sources can be developed and applied to the system not changing the existing system and without having to specify its existence in any other knowledge source. That means we want to restrict an agent as little as possible – hence we assume it to be a regular Linux process. The whiteboard should not be a complex framework that encapsulates the agents or uses special language to define the agent (as [17], [18]). The agents' behaviors (reactive and goal-driven/deliberative, stateless and stateful) are up to themselves. The whiteboard just provides an API for easy and flexible data management. Whatever is that data format on the whiteboard, it should accept the data of the agents of new sensor devices with little effort, regardless of whether the new data is laconic or heavily annotated.

In short, we must keep in mind the list of requirements for the whiteboard in the context of sensor data acquisition applications:

- Suitable for embedded computers
- Located in one machine
- Single data format
- Flexible for new agents

2.3. Related work on blackboards and alternatives

As the background scenario, the idea of blackboard/whiteboard architecture and the basic set of requirements have been introduced, we will now take a broader viewpoint on the topic. Obviously there exist systems for controlling sensors and actuators. Several solutions have been created over time to manage data sharing in the multi-agent systems with similar purposes. The prevalent approach appears to be the utilization of message passing with the publisher-subscriber model between the agents while the shared blackboard can also be found in some systems. The very case of the robot's middleware suits for illustrating the field.

The Player project is one substantial example. It offered a viable and strong option to consider at the time when the Roboswarm project searched for its foundation. The Player [19] functions as a device abstraction server which allows remote client programs to access sensors and actuators over the TCP sockets. The idea is to keep the server side simple and fast, and let the clients solve potential complexities. Distributed architecture is certainly a goal. This allows putting the control program off-board the robot, one robot can access other's sensors, monitoring and logging application can gather the data from different machines over the Internet, etc. Thus sockets are inevitable.

There should be one instance of a Player server per single robot that manages all the available sensor and actuator devices. Any number of client programs can be connected to the server and can send commands to the devices or subscribe for sensor data streams. The command protocol, as described in [20], is rather simple and limited[4]. Implementation of the Player makes heavy use of threads: every device is handled by a thread; there is one reader and one writer thread per every client connection, plus the main thread. If several clients command the same actuator simultaneously, the racing condition occurs [21] (all but the last one will be lost) and a client can receive data packages with a constant frequency while the package may contain output from sensors with different sampling rates (thus missing some values or receiving old values again). Device specific threads are called drivers and must be written and compiled into the Player when a new sensor is added to the system. There was no intended communication between drivers and the client side is also out of scope (a robot control program, which is a client for sensors and actuators can also be a driver in a Player server). In later versions, each driver has a single incoming message queue and can publish messages to the incoming queue of other drivers [22].

The Orca [23] middleware for robots focuses on component based architecture, thus turning sensor drivers into independent components (stand-alone processes) with well-defined interfaces. This supports better reusability of device drivers and control algorithm. The framework implements a proprietary transport mechanism, which eventually relies on TCP sockets.

The UPnP approach for robot middleware [24] increases the freedom even more and uses peer to peer communication between modules which can dynamically leave the system or come online.

In ROS [25] processes (called nodes) communicate messages peer to peer using both publish-subscribe model and services. ROS also provides a large infrastructure with numerous utilities, including a data store (Parameter Server).

The Carnegie Mellon Navigation (CARMEN) Toolkit [26] organizes major capabilities as separate modules. Modules are grouped into hierarchical layers and communicate with each other over a communication protocol called IPC5, developed at Carnegie Mellon University (using TCP sockets again).

Miro [27] is explicitly object-oriented robot middleware that uses the distributed object paradigm. It relies on the common object request broker architecture (CORBA) standard and its real-time C++ implementation TAO (The Ace Orb). Sensors and actuators will be naturally modeled as objects and clients use standard CORBA object protocols to interface to any object, either on a local or a remote machine. Event-based communication services based on the CORBA notification services are also available.

The intelligent robotic wheelchair project described in [28] solely relies on the blackboard communication model and is very similar to the Roboswarm's approach in this respect. The agents communicate by manipulating information on

[4] Some examples:
c<device_name><size><command> - sending a command to the device
dr<size><device_name><access> - requesting read or write access to a device

[5] http://www.cs.cmu.edu/~IPC/

the blackboard and there is no global controller for the agents. Their system has, however, only four agents and all the sensors and devices are managed by one agent (also only one type of actuators exist, wheels' motors).

The RoboFrame [29] framework uses blackboard besides message passing for large structures (e.g. map data which is occasionally updated by different agents). The regular messages are objects capable of serializing to and (deserializing from) a byte stream. More information about these middlewares and many more can be found in [30] and [31].

In the realm of telemedicine solutions there has been less focus on the internal architecture of the home gateway device which collects the sensor readings, preprocesses and eventually submits them to hospital. Still, the principal possibilities are the same as with the robot middlewares. However, it should be noted that some projects abandon the multi-agent approach completely, e.g. Home Client in [32] is built as a Windows application or [33] uses gaming platforms (Wii, Playstation 3, and Xbox 360) for interacting with the patient and gathering data where the applications were built as platform specific applications (also no USB or Bluetooth sensor connectivity). One explanation to the lack of motivation to split the applications to independently running modules is that the home gateway systems cannot be distributed similarly to the robot's case. While it was usually possible and sometimes practiced to put some parts of the robot's system (e.g. control process) off-board, i.e. on a server, then it is typically impossible to access the sensors at home directly from the hospital's servers. Those telemonitoring solutions that comprise multiple modules use some sort of message passing. The project described in [34] uses D-Bus ([35], messaging bus system originating from Linux desktop environment KDE, later also in GNOME, relies on UNIX domain sockets) which gave them several advantages over the previously used Java-based solutions. Another health monitoring solution [36] claims that data acquisition modules receive data from providers via SOAP messages (e.g. web service-enabled sensor networks).

The basic idea of the blackboard communication model can be spotted in a large number of arbitrary applications, but is not always recognized as such. Indeed, we could think of a usual information system with a database and web interfaces (e.g. e-store, e-banking) as a blackboard system where endpoints do not exchange data directly but by editing and querying the database. However, using ordinary SQL databases for interprocess communication when other, often faster and lighter, possibilities exist is considered an anti-pattern in software engineering (see also Appendix 1). The original (or official) blackboard does not seem to be a popular research topic or a popular foundation for implementing systems. The [3] brings out several problem domains for which the blackboard solution is especially well suited: sensory interpretation, design and layout, process control, planning and scheduling, computer vision. The early famous blackboard exploitations were the speech understanding systems HEARSAY-II [37] and HASP project [38] for interpreting continuous sonar signals for detecting submarines. The BB1 [1] added the second level of blackboard to the architecture, thus providing better control (next KS to be executed) in the latter two systems. More examples of blackboards can be found in [39] – a movie theatre administration system and [40] – system for forecasting the atmospheric transport of the radioactive noble gas radon based on measure wind and emission fields.

2.4. Related work on agents

The next issue that has to be elaborated is the concept of *agents*. The term *agent* is used extensively in literature and in this work. In the loose sense the *agent* is a synonym for a program or process that runs independently and fulfilling some task and that is how it should be taken generally. There are also the "official" agents known from the agent-oriented software design. Based on this paradigm ([15], [41], and [42]) the agents are fairly complex programs or even systems which are reactive and proactive at the same time, also possessing the ability to learn and improve with experience [43]. The biggest difference with the blackboard's knowledge sources is that an agent is always autonomous whereas a blackboard has a scheduler. The proactiveness implies creating and adjusting plans to achieve the agent's goals, reactiveness is the ability to respond to the stimuli and events from the environment. Finding the balance between proactiveness and reactiveness is a key issue [41]. The agents are also assumed to be social and negotiate with each other. However, no assumptions are made about the platform and agents of the same system can be written in different programming languages [44].

We use the term agents in the loose sense while speaking about the whiteboard design and the case studies (e.g. Roboswarm). Our lightweight agents, following the statement by [16] that simple parts will make up robust system, are not proactive. However, nothing actually prevented them to be. Although not being consistent with the official concept of agent they are more similar to the agents than to KS'. The closest classification by the [45] should be the response function agent, because our agents do not build or possess internal representation of the world/environment. Again, nothing deliberately prevents it.

Agents can be organized into various social structures [46] – flat, hierarchical, subagents, modular (each module is a multi-agent system). For facilitating message exchange between agents several middleware architectures have been proposed with slight differences, e.g. [46], [47], [48]. Communication plays major role in the world of agents mimicking the circumstance that most work in human organizations is done based on intelligence, communication, cooperation, and massive parallel processing [45]. However any kind of synchronization between agents inhibits autonomy [18] and there is a conflict between the goals of autonomous agents and the best interests of the group as a whole [49]. There are special languages for creating messages like KQML [17] and FIPA ACL [50] and even higher level data exchange languages which function in terms of commitments as presented by [18]. On the other hand, these languages do not specify how the real communication should be implemented, as pointed out by [51].

In parallel to the genuine notion of autonomous agents some research in the field is actually drifting towards the ideas of the blackboard approach. The [52] advocates for separation of the code that implements some behavior from the code that tells the agent when to apply each behavior. There exists also the computational market topology (social structure) where all the agents have access to a common marketplace where information can be exchanged and negotiations take place [45] – resembling directly the blackboard architecture. Instead of being maximally autonomous and communicate directly, in many cases agents rely on supportive entities (middle agents) as brokers, auctioneers, facilitators, mediators, matchmakers, agent name servers, information extraction agents, web proxies, and agent management agents [43], [47]. The [46] adds a shared database, although claiming it to be not as common or necessary as the other ones. The [53] uses intermediate (interpreter) agents to solve the cases where two agents need to change data but work with different ontologies. The [48] uses blackboard inside an agent for belief structures.

It is also possible to observe other similarities between KSs and agents or encounter properties of multi-agent systems among the blackboards. For example, there is a natural tendency that KSs group into hierarchical (social) structures because KSs respond only to a particular class or classes of hypotheses reflected in the blackboard and information can be transferred from one level in the hierarchy to another only through processing by the knowledge sources, as discussed by [54] and [55]. The authors of [56] use a system (real-time strategy game AI) which includes both agents (units in the field) and KSs (control) where the agents are kept very simple but can appear in large numbers (400). The [57] calls their blackboard whiteboard and builds an extra layer of managers between the whiteboard and real KSs (called components), thus hiding the contents of the shared medium from the KS and making them feel more like agents. In [58] each *KS agent* registers interest in particular events that may be announced by other agents. When an event is announced, the broadcasting system (connector) invokes all of the procedures that have been registered for the event.

3. WHITEBOARD DESIGN DETAILS

When trying to implement the whiteboard according to the previously mentioned principles there are several choices to be made. There hardly exists a universal solution: if the optimizations are also considered, which the case is, then the details of the domain have an important role here. We must find out how the actual data items and their variability looks like and then choose the data model. This will have a great effect on the whiteboard's implementation, on the query language and on the overall way how the agents access the data. Of course, the inner structure can be hidden from the agents by the API, but only with extra stress on resources.

The next question is the choice of underlying tools. One possibility is to use third party libraries (D-bus, Sqlite, etc.) for realizing whiteboard's functionality which reduces development work and can be very convenient. At the same time it brings along some useless stacks of calls and undesired effects on performance. Another solution would be using the mechanisms for inter-process communication that the OS provides (pipes, sockets, shared memory, etc.). Discarding libraries means that several low-level functionalities have to be implemented from scratch (locking, garbage collection). Theoretically it is possible to go down one more level and change OS but this would need even more effort.

Yet another issue is the design of the frontend. This includes data presentation and query languages, the prototypes of the API functions and access policies - what should be allowed for the agents and what should be forbidden. The whiteboard should not be very restrictive, but should maintain order and avoid mistakes; hence there must be some constraints. For example, do all the entries need to be unique or not (primary key) or can the agent update the output the second agent. As said before, there is no direct obligation to have one to one mapping between the internal storage of the data and the format presented to the agents – e.g. the data is kept in tables but the query output yields objects or structures (with copies or direct links to data).

Depending on the data flows between the agents and the desired workflows of the application a need for special purpose control agents may rise. For example, when the system wants to start and stop agents frequently and use whiteboard entries for triggers, it could host a special dispatcher process (see Paper 2, similar to the agent management agent of [47]). It is not directly a part of the whiteboard but is also not a typical agent of business logic. Another example is the rule engine. The behavior of the overall system is hardcoded into the agents but some of it (the part which has no hardware access involved) might be expressed by rules as well. The rule engine would be a general purpose agent that applies the rules automatically and thus makes the behaviors to come alive. For doing it effectively it has to adapt to the data without a major effort (searching, converting, and coping) and that in turn affects the design choices of data model and API.

3.1. Data model

When talking about the data model we are referring to the method of how the data is organized into records (entries). The term is usually associated with databases but it is not wrong to apply it to the whiteboard. For simplicity we could say that there is no difference whether the whiteboard is used as a message channel or a database while the frontend of the whiteboard only accepts and returns entries with certain syntax and semantics. As a part of the data model issue we also discuss the schema because they are very closely related.

Let us imagine that a sensor agent running in the home telemonitoring gateway system (adapter to blood pressure monitor) has acquired a piece of information it wants to put into the whiteboard, e.g.: *"Got a blood pressure measurement 135/98 with pulse 80 at timestamp 2012.01.01T10:20:30 by device Foo A10."* The pulse value is included into the statement because the measurement device outputs this parameter. There is no hint to the patient's ID as the adapter agent has nor should have any knowledge about that.

The relational data model with a fixed schema is clearly a very straightforward way to build the whiteboard. Every measurement or message type should be given its dedicated relation. This approach would allow good possibilities for optimization, for example de-normalization for fast access and normalization for memory conservation. From the perspective of the agent developer it is a well understood data model. The example data item could look like this:

Blood pressure measurement				
Systolic	**Diastolic**	**Pulse**	**Timestamp**	**Source**
135	98	80	2012.01.01T10:20:30	Foo A10

(note: id field is not necessarily needed as timestamp functions as the primary key)

The poor extendability is the main shortcoming of the relational model [5]. The set of relations form the explicit schema. The schema must be maintained (e.g. system tables about user tables in Postgresql) and is expected to be rather static. Adding a new measurement type means automatically updating the schema. Since the whiteboard could experience lots of types of data items, like personal memos of agents, the schema will grow in size, data validation takes time, conflicts can happen. If the situation happens where some relation needs updating then serious problems will arise. For example, let there be a new device *Foo B20* which buffers measurements internally and outputs timestamps of the measurement – thus adding new attribute *source timestamp* to the entity. Or the *source* field is needed to split into *manufacturer* and *model*. The whiteboard may manage with the schema changes but all the agents, by default, might not.

Therefore, in the case where the schema is vaguely defined and can evolve, it would be more appropriate to use a schema-less data model (i.e. a universal schema). This means encoding attribute names in fields and not in column names as usual. Roboswarm's extended RDF is one example. An extreme case would be to express everything in a triples model with the smallest possible number of columns (assessed in Appendix 2). This data model belongs to the NoSQL theme and is known as RDF or entity-attribute-value (EAV) model. The triples form a hierarchical structure where single entries are linked by contents as in the relational model. Though one triple may have only one value and one subject, many-to-many connections are still possible. The example data would yield the following set of triples:

Subject / entity	**Property / predicate / attribute**	**Value / object**
Blood pressure	Measurement	#key1
#key1	Systolic	135
#key1	Diastolic	98
#key1	Pulse	80
#key1	Timestamp	2012.01.01T10:20:30
#key1	Source	Foo A10
Or		
#key1	Source	#key2
#key2	Manufacturer	Foo
#key2	Model	A10

RDF is not absolutely schema-less. The lower level schema of three columns must exist anyway and higher level schema (of user data) exists implicitly because the agents have to know what the attributes mean and which attribute to expect at which situation. Totally unknown values could not be interpreted by the agents.

From the whiteboard's point of view, however, the lower level schema is not subject to changes and the upper level schema is left for the agents to manage. In that sense we can call the RDF triples schema-less. By this choice the design and implementation of the whiteboard becomes simpler. At the same time the risks of misunderstandings and flaws in the communication between the agents grow. By giving this task to the agents and taking it away from the central whiteboard we split the schema to smaller sub-schemas shared by the small sets of corresponding agents only (providers and consumers of some type of entities) who should agree on the schema and schema changes.

The other set of data models organize information directly into graph structures: object-oriented, hierarchical, and network models. With chains of parent and child nodes they yield similar effect as RDF but without the triple encoding. Records are not linked by content but by direct pointer references. The hierarchical model only features the tree structure and disallows many-to-many relations. In the context of the implementation of an agent it is very natural to think of the data items (measurements, commands) in terms of objects or structures, thus making this data model a credible choice. On the other hand there might be the need for more effort to be put into the data search mechanisms, cleanup, defragmentation, etc. If data types (object classes) are considered static (defined before use) then the model is also sensitive to schema changes and type inconsistencies may result. With some generalization we can also look at the graph based models as high-level wrappers for the relational (and RDF) model. The research in graph databases is, however, claimed to be died out since the early 1990s for a series of reasons [6].

In some situations - , especially in the case of simple data types - the key-value pairs model can be very efficient (see QDBM[6] in Appendix 1). For the whiteboard application with unlimited data types it is not a good candidate. The classical implementation for a key-value store is the hash table where all searching is done by keys. Though possible, the solution with a key-value model would be messy. A lot of redundancy is needed and data fields will have to contain lists. For example, to present a record there should be at least n+1 key-value pairs if the record has n attributes: one pair has a list of names and values of all attributes and unique ID for the key; other pairs construct keys from the name and the value of each attribute in the original record and have a list of all matching record IDs in the value field.

The choice of whether to bother with extendibility or not depends on the needs of application. It is worth of some attention. Let us come back to the example piece of data encoded into the triples format. This is a raw output of a sensor agent. Other agents might want to process and refine the given measurement, e.g. evaluate the result. They could create a new entry with a different structure and leave the original intact or save space and expand the original with additional attributes and values. In some cases it is rather safe to add new sub-records (if security policy permits it), for example consider the triple *#key1 – evaluation – above normal*. Those agents who also access the same type of measurements can just ignore the unknown attribute (or tree of attributes) and everything should remain working regardless of the schema changes.

A more problematic case is when the new piece of information tries to extend the triple with a terminal value. Say, we have a triple *#key1 – pulse – 80* and there is an extra piece of knowledge about the value, e.g. explicitly stated low confidence (the patient moved during the measurement). The original triple has to be split into pieces and only then it is possible to link the new information to the triple. The same thing happens when a triple wants to refer to another triple that has a terminal subject (not a key). The side effect of this splitting, referencing through artificially added keys (blank nodes) and reassembling (reification) would be problems for possible consumer agents of the data items. Searching becomes more complex and reification (either on the side of the whiteboard's API or in agents' implementation) consumes resources. The following examples illustrate the row splitting:

Subject	**Property**	**Value**
#key1	Pulse	#key3
#key3	Value	80
#key3	Confidence	30%
#key4	Subject	Blood pressure
#key4	Property	Measurement
#key4	Value	#key1
Archive 1	Contains	#key4

[6] QDBM - Quick DataBase Manager, http://sourceforge.net/projects/qdbm/

Again there is a tradeoff between performance and space. One might create enough blank nodes in advance. Then it is easy to link with whatever new data and the changes cannot come unexpectedly to any agent. All the data will be spread as sparsely as possible, no splitting and reassembling are allowed. Managing the data items becomes incredibly difficult and the result is not RDF anymore, but rather a graph model encoded into triples. All the data will be in the edges (RDF property column) and vertexes are just arbitrary points where the edges can start and end. The higher level scheme is hard to observe. Whether there exists any potential use case for such extreme extendibility or not, the data of the demo measurement can be encoded as:

#key0	Blood pressure	#key1
#key1	Measurement	#key2
#key2	Systolic	#key3
#key3	135	#key4
#key2	Diastolic	#key5
#key5	98	#key6
etc.		

To conclude the issue of data model we can say that several possibilities exist when choosing the core structures for holding the data in the whiteboard. The models differ from each other mostly by the level of scheme explicitness. This in turn affects the extendibility of data items which is not a common practice in the realm of databases (fixed scheme) but suits together with the concept of whiteboard.

3.2. The underlying medium and tools

There are several choices for the underlying medium of the whiteboard. As said before, one could implement everything from scratch and even delve into the kernel development of the OS used, but this approach does not belong into the context of the current work. The focus is on the native mechanisms that the OS provides and the functionalities of third party libraries built upon the same OS. The key point of the whiteboard is inter-process communication combined with the memory aspect. This opens up two perspectives: to use a communication tool and to try to add memory or to use a memory tool (database) and to add communication.

The Linux operating system kernel has several mechanisms for communicating information from one process to another. Most of them are dedicated for message passing with small differences and others are just shared mediums. They are all accessible (create, open, send, receive) via small APIs of system calls. Let us consider the list:

- TCP, UDP sockets
- UNIX domain sockets

- Pipes
- Files
- Message queues
- Shared memory
- Signals
- Semaphores

Sockets and pipes are meant for sending and receiving byte streams. TCP/IP sockets (stream type sockets with internet addresses) are a typical solution for implementing inter-process communication. Internet sockets are the only option to be used if some processes (agents) should run in a separate machine. If there is no such need, the UNIX domain sockets provide a similar interface with a slightly faster performance (comparison tests by [59], see Appendix 1). The pipes (the named pipes are also known as FIFOs) have basically the same purpose but a different interface. However, they yield lower performance than the UNIX domain sockets [59]. As the whiteboard targets the exchanging of complex structures then one must spend some processing power for serializing high-level messages into the byte stream.

Message queues operate in terms of structures, not byte streams. They also do not require the establishing of the connection in the first place as the sockets and pips do. A process can just open a queue and insert the data without concerning the receivers. Any number of agents can join a queue and insert or claim data items. There can be more than one message type per queue and the reader can ask for the next message with a particular type. Still, every message can be read only once (removed from the queue) and the types of the messages are rather fixed, so that the reader must know the length and the structure of the received message.

Files and shared memory are general purpose mediums not specially meant for message passing. Agents can read and write whatever they like, in byte sequences. Concurrent access is possible, but no locking or synchronization exists for ensuring data consistency and integrity by default, hence this has to be added by the user application. The schema has to be built by the user. Files and shared memory match well with the idea of the whiteboard. On the other hand, semaphores and signals are methods for synchronization and are not suitable for moving large messages between processes. Their benefit can be seen in the supportive role of helping to manage the shared access to the common medium inside the whiteboard.

It would seem that only files or shared memory are exploitable tools for building the whiteboard, since sockets, pipes and queues have no memory effect. However, one could create a dedicated server-like process that keeps the needed data for a longer period of time and other agents connect to that process via the channel tools like pipes. For example, [60] proposes the blackboard where contents are stored in a set of distributed blackboard-data processes accessed through a set of distributed blackboard-interface processes. What truly matters here is the performance or the balance between performance and convenience/functionality of the tool. For example, the solution with shared memory also does not come without cost, since building proper locking is very important there in the perspective of fast performance, while creating messaging functionality needs additional effort. Intuition would say that channel tools are definitely better for channel tasks, but experiments show (see Appendix 1) that memory (and file) based tools can compete with the fastest messaging tools. The right choice depends on the precise requirements of the whiteboard for the particular application. Of course, there is actually nothing that prevents using a combination of these methods.

As can be concluded based on [61] there exist many dimensions in this problem of choosing the right tools: the data, the queries, the indexes, column or row orientedness, etc., so it is basically impossible to settle for any definite optimal solution. By [6] the maturity, the level of support, ease of programming, flexibility, and security are also significant criteria in deciding which type of database implementation to adopt. One could also find a set of existing systems built for different purposes that more or less overlap with the needs of the whiteboard. They wrap the given methods and give the user a somewhat higher starting point by solving synchronization and data management issues. Systems for fast inter-process communications use different types of sockets, e.g. Player/Stage, D-Bus, and ROS. Small scale database systems that function as libraries, not daemons, use files. Good examples for this are Sqlite and QDBM. Using files automatically means that they are slower than the other methods, even in the case of keeping the database file on a ramdisk. From the tools that exert shared memory two proprietary implementation are used in this work: Roboswarm's database and its successor Wgandalf (benchmarks in Appendixes 1 and 2).

3.3. API details for agents

The next set of decision points concerns the API design for the whiteboard. The API not only shapes the character of the whiteboard but forces the developer of the agents to think in a given direction. Experience has demonstrated that choices made in the blackboard representation can have a major effect on system performance and complexity [2]. One clear purpose of the API is to hide the technical peculiarities of the whiteboard's internal implementation from the agent level and offer a well-defined easy to comprehend interface. As the performance of the whiteboard and the overall system (agents included) matters, we have to be careful to avoid unnecessary operations. In short, fixing the API means finalizing the data model, applying a security model, and giving a set of commands what the agents can call, including the query language.

The API layer is the place where the data models of the underlying tools and whiteboard's internal logic, as well as the data model of the agents come together. For example, the whiteboard operates in terms of triples which it keeps in a relational database (e.g. Sqlite) and present to the agents as objects or XML documents (strings). Basically it means that the agents do not have to be aware of the physical data modal or the tools used. It also leaves open the possibility to find better opportunities to replace the components in the future. The model that appears to the agents has the greatest impact. Customizations are possible here and an intermediate approach between relational presentation and RDF can be used (Paper 1). The interface might be made very application specific and feature the schema of used data types or be rather universal, i.e. schema-less. A useful strategy would be to create different access levels: low level function calls return original records for fastest access and high level wrapper functions convert the data items to structures or objects that can be handled more easily in large quantities.

Security is a separate topic and worthy of thorough investigation and discussion. However, as the example systems presented in this work discard the security concerns we decide not to tackle the data protection details in this work. We could consider two types of protection, one against the accidental damage and the other against the deliberate attacks. The first case is handled in the API functions to some extent (e.g. check the uniqueness of RDF keys). The complex system of user roles, access rights and authentication is not considered in the current work, since the philosophy of the whiteboard is sharing and contributing, not protecting the data which is often the case with regular database systems. Therefore, all the agents which are allowed to run in the system are trusted not to commit anything mischievous like stealing data, insert counterfeit measurements or forge existing records. By the authors of [46] the similar situation holds for multi-agent systems as only a few of them provide security services as part of their infrastructure.

Using a universal schema does not exclude the possibility of having some functions for handling specific data types. Should absolutely everything be forced under the common universal schema or can there be exceptions is a matter of optimizations and a clarity of the interface. Even as the system is simple, we could encounter many entities, like measurements, other events, comments, goals, tasks, commands, responses, configuration parameters, temporary variables, etc. The API can provide special mechanisms for handling data types of different purposes (e.g. measurements and commands) based on a completely different mechanism or based on the ordinary universal scheme (e.g. triples). However, the first option is discouraged as it adds lots of complexity to the whiteboard or isolates some data from the global access space. The second option is what the agents should do anyway in a usual situation. Doing it on the whiteboard side can have some advantage in performance but risks with the fixed schema problem – some types of data do not fit.

A similar dilemma occurs with the data flow: should there be only one read function and one write function, or several. The basic idea of the whiteboard lays in a very simple access policy: everybody just inserts their records and can read whatever they like. However, this brings along lots of polling. Those agents who expect some commands must poll the whiteboard regularly for the given data. The solution would be to notify the receiver agent when a command arrives or directly deliver the data to the receiver's buffer. To name some drawbacks of this approach, the receiver has to register itself first for some types of messages and the whiteboard must examine all the incoming data to catch the matching records or the sender agent should somehow specify the target process and the whiteboard notifies the addressee. For example the [62] requires registering of every plugin (agent) at the core. Nevertheless, this is not coherent with the initial goals of decoupling data providers and consumers.

One major task of the API is query handling. It determines what kind of queries can be made and how these queries would behave. Sometimes it is beneficiary to limit the query mechanism to avoid misuse of resources. The query language itself can have several forms. For example, if the whiteboard contains triples in a table, then the query interface could be inspired by SQL, Sparql or RDQL. There are usually many triples representing one agent level record and usually they are needed at the same time, hence the query engine should automatically return the referred triples. But hierarchies of triples can be deep – how many levels down or up is reasonable? A similar situation happens when updating or deleting: there is the risk of dangling references and unreferred triples. The common behavior would be to return the copy of the demanded record. A more resource friendly solution returns pointers to the original data record. Depending on the workflows of the system this can also result in the case where the data changes in the middle of the processing of the query result by the agent.

Suppose that the query interface returns a set of triples. While processing the records of a SQL query (*n* fields) is generally acceptable for the developer of the agents then handling a set of triples can be very inconvenient and barely human-readable. The agent might loop over the result set over and over again or parse the result into some structure. That brings us again to the idea of converting the triples to some higher level structures already on the whiteboard's side, i.e. moving towards a hierarchical data model again. For example, there could be an XML interface to the whiteboard or - even better - something like OEM [63] and JSON. The basic idea of the latter two is to express the data with key-value pairs where values are lists of other objects, like atomic values, arrays, key-value pairs. This schema is perfectly extendible as new values can be added to the existing lists, thus refining the original data items. OEM and JSON are textual data representations and can be treated by the API as such while nothing really prohibits the whiteboard to use structures/objects when serving the agents. As the RDF has the Sparql query language the OEM, on the other hand, has Lorel [64].

The model of triples allows all kind of referencing which the hierarchical structures usually do not allow. What is most useful for the whiteboard implementation depends on the application and its requirements. Allowing a graph like structure with references between the branches of the node tree helps to avoid redundant copies of some data items, saving space, but adds more complexity. For example, an agent saves a measurement which includes several values with the same unit and the unit is described by a sub-tree of nodes (not a single string) – if the referencing inside a document (measurement) is allowed as in OEM, only one description is needed. We can also look at the issue from the perspective of referencing in the global scope. If an agent saves one more measurement of that type it can be required to give again the unit's sub-tree regardless of any previous actions or it can be required to look up the reference first (as somebody might have already inserted that data before) and use the reference. In both cases there are different policy choices for the whiteboard – save the data items exactly as given by the agent or take steps to reduce redundancy.

The extendable hierarchical/network data models like the OEM can be easily expressed in tabular formats. For example, [5] proposes to encode the OEM into two tables: the binary relation VALUE (object id, atomic value) for specifying the terminal atomic values and the ternary relation MEMBER (object id, label, object id) to specify the values of complex objects. This gives us inspiration to organize data into triples with possible OEM-like frontend (Paper 8) thus providing a relatively strong extendibility. The following example illustrates the idea, (the textual encoding is neither pure JSON nor OEM but very similar):

Subject	**Property**	**Value**
Entry	contains	#key1
Entry	contains	#key2
#key1	is	activity
#key1	contains	cycling
#key1	contains	#key3
#key3	is	duration
#key3	contains	30 min
#key2	is	timestamp
#key2	contains	2011.08.20T 17:15:00

<–>

```
  entry = {
    activity = {
      cycling,
      duration = {30 min}
    },
    timestamp                    =
{2011.08.20T17:15:00}
  }
```

Similar practices exist in the RDF research where triple-stores are implemented upon the relational databases and some authors specially advocate them for the utilization of functionalities (ACID, indexes, query plan optimizer, intermediate results table, etc.) offered by SQL databases [65]. The [66] builds the database of RDF triples (3store) using MySQL with the schema of four tables. Obviously there can be many different schemas depending on the design and implementation of the query layer and string handling (e.g. should the subjects and values be put into the same or separate tables, store strings in triples or string IDs). To name some, the [6] uses two tables; [67] uses 5 tables.

3.4. Processing with the reasoner

When building a multi-process system in the described manner with lots of small device-oriented agents and the central whiteboard we have gained an interesting opportunity to use a rule-based processing mechanism. In a typical setup all the business logic is hardcoded into the agents. There are three types of agents: sensor device adapters, actuator device adapters, and decision making agents. The latter ones perform the control operations. To do this they only need to connect to the whiteboard and not anywhere else: by reading the data on the whiteboard they can get all the needed input (including feedback) and by writing the right records to the whiteboard they can cause actions. All the data passes through or stays in the whiteboard and there is only one language to express different types of data. Under these circumstances it can be very encouraging to build a general purpose processing agent.

The main motivation for a general purpose processing agent is the reduction of the ratio of the hard coded business logic. Of course, the processing agents can have configuration parameters for adjustability but the core functionality which resides in loops and conditional clauses remains typically static. When adding some new behavior to the system or changing an existing one the only option would be to edit the source code, recompile, and deploy. However, if the business logic would be expressed by some kind of textual (human-readable) rules which are repeatedly interpreted by a dedicated reasoner (rule-engine) agent, one has to edit the given set of rules to change the behavior of the system. This is definitely a cheaper way for refactoring and possibly allows doing it in a live system. The idea of having agents' behaviors written down in rules resembles very much the concept of directly executing the formal specification of the agent as mentioned in [41]. A rule engine can be also used for the distribution of messages, feeding the output of some agent to another agent [62].

The reasoner agent behaves as a dummy control process managing the entire system and the other agents behave as procedural attachments which gather input data and execute commands. At any moment of time the whiteboard is a snapshot of the entire knowledge of the system. The rules express the relations between the different states (snapshots) of the whiteboard, e.g. if there are some particular entries present then perform the given modifications, i.e. add/change some entries. The reasoner takes the contents of the whiteboard and applies a set of rules: if any of them fires and produces an output it will be put to the whiteboard. This procedure is repeated infinitely.

Having the reasoner agent in the system does not exclude the possibility of using anything else beside it. Regular hard-coded agents can exist in parallel to the reasoner agent and realize some other tasks. Technically there could be even more than one reasoner (as discussed by [54], e.g. rule based, case-based, model based). In fact, the reasoner creates the need for another set of agents of procedural attachments – the utility functions. Rules themselves are very inconvenient for doing all kind of calculations, aggregations, and data conversions (e.g. dates). One solution would be to add special functions to the rule language and corresponding procedures into the reasoner. Another option is to build regular agents for performing these tasks.

Nevertheless, rules must be used with caution, especially when they are complex or there are a large number of rules. For example in e-health applications rules are considered for contributing to the wide range of contextual, socio-cultural, dynamically situated factors that influence practice guidelines and patient-centered care, but may end up in a mess. [68]. One problem lays in poor traceability. There are also several other technical details to pay attention to. For example, when the reasoner is in a work phase the whiteboard should be isolated from the modifications done by other agents or some rules from the set can have a different input from the others. This can be allowed if the rules are independent, but it should not happen during the processing of a single rule, e.g. the rule is: *"if entry A and entry B and entry C then derive entry D"* and the rule engine has checked for A and B already and at the time when it checks for C somebody deletes the A. There should be a policy for contradicting rules and the rule engine must avoid reproducing the same output over and over again (flooding the whiteboard) if the presumptions happen to be continuously satisfied for a longer period of time. The reasoner loops with one certain frequency but the data provider agents have their own running frequencies, therefore the reasoner can either meet the frequency of the slowest sensor agent and process faster data streams with delay or execute with the frequency of the fastest sensor agent and thus create extra load.

Including a reasoner agent to the system certainly affects the design choices discussed previously. The key factor is the efficient access to the data, which concerns the API and tools. All the database type tools are clearly more promising. There can even be a separate access mechanism for the reasoner. Relational style entries are easiest to handle with rules. Triples, on the other hand, need much more attention by the rule writer because the data is scattered among several atomic facts. If there is no tabular representation at all but objects in the memory then these have to be converted to facts before usable by the reasoner.

4. THE IMPLEMENTATIONS AND EXPERIMENTS

4.1. Roboswarm

The Roboswarm system was the earlier version of the two practical cases of using the whiteboard for inter-process communication as described in the current work. The goal was to spend minimal resources but still have fast responsiveness of the overall system while using the multi-agent architecture with reasoning capabilities. Everything else, like security, API clarity and data integrity had only minor significance. The whiteboard became the central component of the robot's middleware serving all the possible other agents which could also be introduced in runtime (e.g. dynamic openness of [46]). There was no distinction made between different types of data items as sensor readings, messages, commands, and derived facts. Differently from [52], all data items were treated just as whiteboard entries. While unconventional, it made all the possible send and receive operations really straightforward.

The data model used was a hybrid of the relational and RDF models. The basic idea was to use RDF in principle to guarantee preparedness to all kind of data that can be encountered. In other words, it means schema-less design – no schema defined in advance; changes to the schema can be done in a live system. Data expandability was not initially considered. However, it was shortly noticed that the majority of data items shared some common attributes anyway and it would have taken many extra triples to encode them in the proper RDF. These attributes were so called *meta-data* giving some information about the original data entry, namely the *ID*, the *timestamps* (with microsecond precision, marking creation and last modified), the *source*, and the *context*. (see Paper 1) As a result, the whiteboard's entries were extended triples: RDF fields plus meta-fields, resembling more the relational model with one table than the RDF in the end.

From the implementation's point of view the whole whiteboard became really lightweight and minimalistic. We have earlier defined the whiteboard as the intermediate layer between the agents and underlying tool. In the Roboswarm system the whole whiteboard was comprised of the underlying tool and nothing more, since the fast performance was more important than extra functionalities. One option would have been to use socket-based Player/Stage for managing sensor data streams, but then there would have been also the need for additional components for the persistent storage and history effect, causing overhead and/or messy API. Sqlite was initially considered suitable, but in reality it suffered from severe performance problems when used concurrently by rapidly looping processes. Eventually the choice was to use shared memory and to build a custom database with the extended RDF schema.

Thus, the Roboswarm's database was a set of functions (library) for saving the records (extended triples) into the shared memory and also retrieving them. The database's functions were directly compiled into the agents, no separate database process (e.g. server) existed, and there were no wrappers. This means the database was the whiteboard. The first library call allocated a segment of shared memory and this was preserved until reboot or intentional freeing, no matter if any agent process existed or not. The segment of shared memory was split into subareas. The key component was the fixed-length circular buffer of records. Each record held the fields of one extended triple, some of them in place, and pointers to others. The remaining part of the memory segment was filled with structures for keeping strings (hash table) and control information like the pointers to the areas, locking information, pointer to the latest record, etc.

The library functions either just located and returned the information from the memory or did the proper modifications to right areas in case of writing, deleting or updating. All of the functions also encapsulated locking routines so that the agents had not to be aware of the locking matters. Locks were implemented by exerting semaphores. One main idea was to spend as little time as possible in the synchronized sections. The function call gets the lock, quickly does the operation and releases the lock. The assumption is that an agent spends most of its time on other activities than inside the whiteboard calls.

The API of the shared memory database was simple at first sight but tricky in details. The agents could insert data by providing a single record at a time, but not all the fields were specified by the agent: some meta-fields were filled automatically by the whiteboard. Usually the fields had a predetermined type, e.g. integer for *ID* and string for *RDF subject* but the *RDF value* field might contain values of different types, not compatible with the principles of the relational model. There was no query mechanism at all and the only way to find the data was to scan the records. The records in the circular buffer were in the order of creation and an API function returned the pointer to the latest record. By knowing a record, an agent could get the pointer to the previous record, so it was possible to scan through the entire whiteboard, starting from the latest data until the oldest.

The agent had to get the record pointer first and then ask for specific fields one by one without the need to fetch the entire record. It was possible to check the type of the *value* field before making the fetch call. Due to the scarce resources strings were not copied to the agents but instead of that the pointers to the original strings were returned, making the solution extremely vulnerable to accidental overwriting. As the agents do not get copies of the records in general, this means that the contents of the record can change during the processing by the agent. It can happen between the fetching calls of the individual fields, but this causes no problem if used carefully and is needed for some use cases.

Deleting and updating calls are present in the API. An agent can delete records from the whiteboard but this functionality is not needed very frequently because oldest records disappear by themselves when the new records are written to the same slots (circular buffer). However, there is an exception – records could be made persistent by writing a special value to the *context* field and these records were not subjects for overwriting. In practice there were only a few persistent records which were needed infinitely, thus the delete command had no real significance. Updating operation was possible only on the *RDF value* field, while the timestamp (last modified) changed automatically.

No transactions were allowed as the agents were given no control over locking to avoid long periods of locked whiteboard, risk of deadlocks, and starvation. This means no possibility for combining function calls to form larger atomic operations (e.g. transactions in regular database systems). This makes the usage of triples' data model inconvenient: for example, reading the data items that span over several triples must be acquired record by record where the whiteboard might not yet contain all the needed records at the moment when the reading begins. The same happens when saving a larger data item where the agent must guarantee the uniqueness of used reference keys by itself or different data items could get mixed. In fact there were no special means for dealing with triples, which is the price of having a very lightweight API. On the other hand, it is a still relatively easy way to offer scheme-less solution for agents' design. Besides, the whiteboard manages all issues what are absolutely necessary and which are not offered by the OS's API of the shared memory, like bookkeeping of the records' buffer, organizing the string table and controlling the locking.

Let us take a look at some use cases of the whiteboard. For implementing the robot control the essential input is the access to sensor readings. In the given system the sensor devices were handled by adapter agents who uploaded the data to the whiteboard. They had three possible options to do that. A new whiteboard entry could have been created for every sample and the reader agent had to scan the whiteboard to find the latest value. This approach clearly consumes resources, especially when samples are taken with high frequency. Another option was to create a persistent record (or many) and update its value as new samples arrive. The reader needed to scan the whiteboard only once to locate the record and later just fetch the *value* field, while the *timestamp* field could have been used to determine if the given value is new or old (constant values vs. sensor stopped). No trace of the historical values was left. The third solution was to use both methods simultaneously – keep the latest data in one fixed place and save extra records every now and then.

While the output data of one sensor device is produced in one place and can be consumed in many places, it works the other way around with actuator commands. There is an actuator agent managing the given actuator device and executing commands found on the whiteboard. The control agent that gives a command must encode the data (i.e. command parameters) into the records and write it to the whiteboard. The receiver agent has to constantly monitor the whiteboard to discover the added commands and to execute them. This causes a lot of scanning, but optimizations are possible. In each cycle only the very latest part of the whiteboard (circular buffer) had to be scanned – the scanning agent got the latest record and memorized it as a bookmark (using *ID* and *timestamp*), then fetched earlier records until the bookmark from the previous scan. In this way the agent checks only the records added since its last cycle. To identify the commands that the particular agent can handle the *context* field was used. Namely, the one who gave the command put the name of the addressee (or command type in general) into the *context* field and the reading agent makes the first filtering based on this field, never examining the records with mismatching *context* further. In that sense it is a form of message passing. Note that the records of commands could not have been made to persistent records (thus no deleting required). The fact that the majority of commands override the previous ones allowed the actuator agent to scan to the first matching record (which is the latest by timestamp). A good example is the velocity of wheels: there is no point to set it to the value A and then to the value B, instead it should be set to the value B immediately. However, if all the commands were to be executed in sequence, then the scan had to go up to the bookmark.

The rest of the robot's system was control-oriented. The concept of control stands for guiding the robot's actions either reactively or deliberatively (or both, on different levels). To accomplish that, the controlling agent must use the sensor input, the task information (given by a human user), and other feedback to give to the actuator commands. There was the main control agent whose role was to pick up the task from the whiteboard and to invoke suitable behaviors. The actual control was divided among several agents called *sub-behaviors* because they held the code for solving certain subtasks (e.g. navigate, travel, solve obstacle, etc.) and they were able to launch each other (Paper 2). However, the launching operation was indirect. The dispatcher agent was created which got the launching (and stopping) commands from the whiteboard and physically started (or stopped) the agents. Parallels can be drawn here to the agent-oriented design's idea that agents encapsulate certain functionalities and may be involved in a series of employer-subcontractor relationships [15]. In every aspect the sub-behaviors were like the normal agents but they did not run infinitely: after performing the task they also produced the return value in the form of whiteboard's record.

In addition to these agents the system included the reasoner and the communication agent. The communication agent was managing the WiFi connection to the server. Its role was to (a) receive task specifications (or whatever data) from the server and write it into the whiteboard and (b) upload the feedback data. In the case of the robot's system the server's data model was an extended version of the robot's data model and thus no conversions were needed to be done on the robot's side and only minimal conversions on the server's side.

The reasoner agent served as the general purpose processing agent. It was especially useful for doing smaller independent operations with triples where building an agent process would have been overkill. It enabled also the option to replace higher level sub-behaviors with rules: when the preconditions are satisfied then the rules launch lower level sub-behavior agents (with dispatcher command records), these provide output which matches the assumptions of another rules and so on. As all the knowledge passes the whiteboard it would have been theoretically possible to substitute all control agents with rules. However, the reaction time of the reasoner agent did not meet the tight requirements. The rules were expressed as Horn clauses in the text file, basically telling the reasoner to insert the given record to the whiteboard if certain records already exist there. The reasoner should loop infinitely, which causes the problem of flooding: the triggering records remain in the whiteboard for a while and the result can be deduced over and over again. To avoid this kind of behavior the reasoner agent added the result to the whiteboard only if at last one the assumption facts (records) was new, e.g. had a newer timestamp in respect to the newest record of the previous working cycle of the reasoner. The core of the reasoner was the theorem prover Gandalf which was put to run in loop. The real-time performance complications came from the fact that in each of its cycles the records of the whiteboard had to be converted to the reasoner's internal structures.

As the functionality of whiteboard was scarce, the agents had to take care of several issues by themselves. Leaving the implementation of critical functionality into the responsibility of the agents certainly comprises the risk of flaws. On the other hand the argumentation here was that the custom solutions are more optimized than the universal one and in many cases it encourages the agents not to use the sophisticated solutions. One example is the use of (extended) triples. While it was absolutely possible to create and retrieve hierarchical structures of records (linked via the contents), in practice the sensor samples and actuator commands were presented by using only one record. This can be achieved by encoding several values (e.g. angular velocity and translational velocity of a motion command) into one string and place it into the *value* field of the single record. The alternative would be to create several independent records (one for ang. v and one for trans. v) leaving the receiver agent the obligation to search for the set of necessary records.

There was also no explicit mechanism to connect the commands and their responses: the implicit solution was to refer to the command's *ID* in the response record. However, no feedback was needed in the majority of cases. The commands were executed and soon the effect was seen from the sensor readings. The RFID write operation was one of the commands with the feedback (reports of fail, success, used number of retries), but this was inserted into the whiteboard as ordinary records not referring to the command. This was safe because there were no other RFID tags nearby and no possible concurrent ongoing write operations. The agent who gave the command just had to know what kinds of records to search for.

Much of the system's reliability was based on the carefulness of the developer of the agents. For example, the situations where an actuator agent gets conflicting commands (e.g. drive forward vs. stop) from different sources had to be solved or avoided. There were no built-in protections for conflicting commands in actuator agents to save resources. To do this the agent must check the *source* field of all commands and have a policy to choose between different sources. While the list of possible agents is not known in advance, there could also be agent classes or the list of agent priority values in the whiteboard: none of them come cheap. However, the situations where the conflicting commands can occur actually indicate design flaws. In a normal scenario a control agent (sub-behavior) gives lead to another process and gets it back later. There is no reason to make several agents do the same thing at the same time and to issue possibly conflicting commands to the same actuator.

The similar argument holds for the accidental re-launch of agents. The dispatcher had no protection against the situations where the same sub-behavior is launched over and over again while one instance is already running. There was no reason to restrict the dispatcher because there was nothing wrong in calling the same agent again (with different parameters for example). But when the sub-behavior's task is to retreat from an obstacle, for example, and it uses robot's wheels then chaos will result if many instances happen to run simultaneously. It could happen in practice because of the design and implementation flaws of control agents or rules. The agent can always implement self-protection, e.g. put (and update) an aliveness token record to the whiteboard if there is none yet or stop at once if there is.

4.2. Telemonitoring

The telemonitoring gateway system was very similar to the Roboswarm's in most of the aspects. Again, it is all about sensor readings, actuator commands, and control in a multi-process environment, also featuring contact to the server, reasoning, schema-less design, decoupled providers and consumers and scarce resources. The basic concept of the whiteboard was borrowed from the Roboswarm but in the end the whiteboard's design differs well enough to take a closer notice. The focus on the fast (near real-time) performance is no longer required and data items tend to be more complex. This pushes the whiteboard towards a more advanced API functionality with better support for building agents. This also means the reduction of the number of low level details the agent developer has to be aware of.

The problem of the data model of the Roboswarm's whiteboard is that the semantics of the meta-fields is not very clear. If the right set of meta-fields is chosen then the number of triples needed to present data items can be reduced, to one record in the ideal case. On the other hand, if a data item is encoded into many triples, we experience duplicate meta-fields for all the records (except for the *ID* field). The concept of meta-fields is rather vague because there is no explicit border between the proper data and the meta-data. This makes it hard to choose a good set of meta-fields. The meta-fields open up opportunities for better search and optimizations but at the same time make the data model more confusing. The fields can have different meanings in different situations, can be unused, can be used in an ad-hoc manner, etc.

The data model of the telemonitoring gateway's whiteboard discarded the use of meta-fields in the sense they were used in robot's whiteboard. Turning back to the hierarchies of RDF triples is cumbersome, but can be more comprehensible for agent developers. However, a few meta-fields still exist in the internal storage of the whiteboard which are never presented to the agents in the API. On the user's level only triples exist. The additional fields are basically for utility purposes, as the *ID* or the flag that indicates the status of the record (normal, to be deleted, deleted). The triples are still organized into individual records which are linked by matching contents of the *subject* and *value* fields, making it easy to store the data into tables.

The Roboswarm's data store inspired the creation of the next generation shared memory database for general purpose usage. That database (a library to handle data in shared memory) was developed by the makers of Gandalf theorem prover and is deliberately designed with the capability of supporting the low level integration with the reasoner, hence the name Wgandalf. It also features only one table of records, but these are not organized into the circular buffer and no automatic deleting (overwriting) of old data is performed. Every record can have any number of fields and a field may contain any type of data. Not all the values were kept directly in the fields of the record but were stored in other structures and pointed at (as discussed in [65], reducing string comparison). User-defined columns as such do not exist; all fields are addressed by their sequence number.

However, this database features a query interface in addition to scanning. For serving queries there are also indexes. The scanning works in the different direction compared to the robot's database – starting from the oldest entry and moving towards the newest. Locking is realized with custom-made spinlocks, the responsibility of calling the locking functions at the right moments is put to the user.

The agents in the robot connected to the memory database directly and had to be well aware of the nuances of the database. The whiteboard of the telemonitoring gateway followed a more systematic approach. The Wgandalf database is used here as the underlying tool but is wrapped by the whiteboard's API functions. The API hides the interface of the real database from the user (agents) and performs converting operations on the data. It is also important to notice that the API manages locking internally and does not expose it to the agents (similarly to the robot's case). The overall purpose of the API was to simplify the use of triples for the agents but it goes halfway with this goal. The agent stores and requests sets of triples representing data items, the whiteboard helps to keep data integrity, and manages reification, but there is no Sparql or other high-level query interface.

The API provides a small set of essential functionalities which includes reading, writing and deleting. No update was allowed: one must remove the old record and insert the new. All the data items consisted of one or more triples; all the fields (*subject*, *property*, *value*) may contain only string. The write call replaced the reference keys in the given set of triples with unique values before the actual storage, thus the agent did not have to worry about the issue of accidentally intermixing different data items. The API treated the given sets of triples consistently, avoiding the situations where only a half of the set is written to the whiteboard. As the storage space is limited and the whiteboard kept all the data permanently as a regular database, the deleting function was inevitable. Triples could be deleted one by one or in cascade (the function call internally followed the references and deleted the given number of hierarchy levels). For optimization reasons the deleted triples were not removed from the Wgandalf immediately but marked for deletion (never visible to the agents).

The read call had two variations, the regular read and input buffer read. With the regular read any triple could be fetched. The agent specified the query and got the set of matching triples: the scanning option did not exist for the agents. Reading operations behaved in the cascade manner by default, which means the agent got the whole sub-tree of triples (a complete data item) with a single query. Another major difference from the robot's case is that the reading calls return no pointers to the original data, i.e. the output is composed of copies, thus adding more protection against accidental damage. The mechanism of the input buffers (Paper 7) had the following motivation: make the command passing more reliable from the receiver's point of view. In the regular case the receiver has to always keep track of both the executed commands and the commands yet to be executed. The agent must delete the command data, keep bookmarks in its internal memory space or keep bookmarks in the whiteboard – all have some inconveniences. The more systematic approach is to move this functionality into the whiteboard and let the agents just query their input buffers. The input buffers were not built aside the triples' model but upon this – to insert a data item to the buffer means to reify the item and to add the encapsulating triple with the receiver's name. The receiving agent can fetch the contents of its input buffer with a simple command (evading queries) and every item is delivered only once – as the read call returns the contents, the whiteboard is slightly altered and the next call would not yield the same item again. However, the data remains in the whiteboard and is perfectly accessible via the usual query interface.

The reasoning agent of the gateway's whiteboard (Paper 3, Paper 5) made use of the SWI-Prolog similarly to how the [67] NoSQL database made use of the Sicstus Prolog for query solving. Similarly to the robot's case the stumbling block was again the synchronization of data between the whiteboard and the reasoner. To do it through the regular triples' interface is inefficient, hence an alternative interface to the whiteboard's contents was given to the reasoner agent, making it an intermediate entity between the whiteboard and the typical agents. At startup an instance of Prolog is spawned and the reasoner agent maintains the connection to this instance. The agent cyclically feeds (assert) the new data to the Prolog, withdraws (retract) the deleted data, allows the physical deletion from the Wgandalf and calls the rules to produce new knowledge. There is no timestamp field in the data model, so the bookmarking is based on the *ID* field. Avoiding the re-deduction of the same output was not a straightforward solution, but still achievable – the reasoning agent gave the timestamp of the cycle to the rules (when called) as an input parameter and the rules had the opportunity to compare the timestamp to the timestamps found in the data items (which usually had timestamps). The output of the rules was eventually stored in the whiteboard.

The workflow processes are more or less similar to the ones seen in the robot's case, but happening at a much slower pace. The sensors of the robot (e.g. wheel odometers, sonars, cliff detectors) were outputting readings with regular intervals many times per second. The medical measurement devices (e.g. scale, blood pressure monitor, glucometer) on the other hand, produce output irregularly and a couple of times per day at most. Everything depends on the behavior of the patient. Because of these circumstances we could look at the measurements here as being events rather than sensor values. Hence there is no need for a persistent place for holding the latest value and all the events are saved into the whiteboard as individual data items with as much context info as possible. The overall number of fields per one data item representing an event is higher compared to the robot's records. Besides the timestamp, the sensor device name and unit there can be more than one name for the measured phenomenon and there are usually some proprietary outputs of the particular sensor device (e.g. flags *low perfusion*, *marginal perfusion*, *artifact* of a SpO_2 meter; flags *before meal*, *after meal* of a glucometer).

The control part had the role of checking the measurement data, producing warnings and communicating the knowledge to the server. Every control agent was specialized for dealing with certain events only. The input data was simply queried from the whiteboard. Command passing was done via the input buffer mechanism but had a fairly modest role: the commands were involved in the server communication and user interaction. The dispatcher agent was not really needed, since all the agents started at boot time and kept running infinitely. The server had its own data format and communication protocol dictated by the gateway level – this knowledge was encapsulated in the communication agent which had to convert the messages (triples) received from the whiteboard. Data deletion was a relatively important part of the control logic, because the whiteboard was finite and no automatic cleanup was allowed. Basically two options existed (a) to do it centrally by the cleanup process that should keep the number of different types of data items in predefined limits; or (b) let the control agents remove the data items they are managing. In reality both scenarios were used simultaneously as an agent deleted all types of measurements and everything else was left to the agents. The gateway featured also the user interaction apart from the robot's case – a touchscreen was used to show information (e.g. warnings, measurement feedback) to the patient and gain input from the patient (e.g. confirmations, evaluations). Generally the reaction times were accepted to be as high as several seconds – for example, when delivering a blood pressure measurement from the sensor to the server. However, the presence of the user interface directed the system to be near real-time as the ease of use and usefulness are crucial elements to be considered during the design stage [69].

5. CONCLUSIONS

This thesis summarizes the outcomes and carry-over of the two individual projects, both making use of the whiteboard architecture. It explains some of the details concerning the design and implementation of such systems. It is important to understand that the research work is not complete in the given field and must continue. However, the current findings suggest that there is a good potential in the whiteboard systems as understood and defined in the context of the thesis.

There exist a lot of concepts similar to the whiteboard, hence it is important to stress that by the *whiteboard* we do not mean the blackboard systems which typically run on only one processing program (called the knowledge source) at a time, or the official multi-agent systems where complex agents share no common medium. The ideology of the *whiteboard* is to take the complexity out of the agents and put it into the whiteboard, enabling a large number of fairly simple agents to run simultaneously. The complexity should be avoided whenever possible, because the small code base is an important premise for successful performance of the whiteboard.

The very basic question of the whiteboard system is the intended usage – what for is it built, how big is the role of the communication functionality, is the storage aspect (called *history* in the thesis) present or not? Another key issue is the presence of time constraints. Even if there are no explicit time constraints, this type of solution – which is basically the *database-as-IPC* antipattern – works only if it is possible to make the internal mechanisms fast enough for concurrent use.

We have investigated the requirements and design of the whiteboard model in two scenarios based on the projects of a robot middleware and a telemonitoring gateway. In the first scenario the focus is mainly on the real time performance and a minimal API with a simple data model is satisfactory. In the second scenario we have to work with more sophisticated data objects, which would have been tedious without the proper API moving the complexity into the whiteboard and out of the agents.

The issue of the data model is especially interesting, since it combines the questions of the internal data storage principles with the ways of accessing the data by the user (agent). The data model is crucial in case we wish to use reasoner based data processing, which is a feasible approach for the whiteboard system. Though accompanied by several implementation problems, our practice has shown the triples' model to have good potential. Triples are relatively human-readable and comprehensible, yet allow non-fixed schemas, which has been one of the common goals in the applications discussed.

One cornerstone of the work of this thesis has been the exploitation of the Linux shared memory. Linux kernel provides several methods and tools for exchanging data between processes, like the well-known TCP/IP sockets and pipes or less known UNIX domain sockets and Linux message queues. However, the current experience suggests the shared memory to be the best suitable option for building a whiteboard. We note that one must be careful with the implementation aspects, especially locking, since shared memory does not have a built-in locking mechanism.

All in all, there are numerous details which affect the design of the whiteboard for any given application and assumedly there is no unanimous specification. Hardware constraints matter and low-power hardware may exclude some desired functionality, but the system can still benefit from the use of the whiteboard architecture. As always, several choices are mutually exclusive and no silver bullet solution should be expected.

5.1. Advantages of the whiteboard solution

There are several architectures and frameworks used in the field of multiprocess sensor data management as discussed in the thesis. However, the majority of these focuses on technologies different from the whiteboard. The blackboard solutions were popular decades ago.

The whiteboard solution described in our work yields various benefits. The primary application case for the whiteboard is a situation where the sensor agents cannot or do not want to know the recipients of the created data items in advance. This is an alternative to the use of sockets, either in client-server or publisher-subscriber schema.

The whiteboard is especially suitable for the systems where the data items are not perfectly structured. This means that the items of the same type may vary slightly, e.g. if one sensor outputs temperature and the other one outputs temperature with the battery voltage, then the records should be of the same type. In case the system handles numerous different types of items and new types are created during the system operation, then the whiteboard offers a good alternative to SQL.

SQL is not available on all platforms, especially the embedded systems, which cannot sustain large installations of SQL database engines. Although there exists the Sqlite database which can be used on practically every Linux-running system, it performs badly under parallel load. The whiteboard (based on shared memory) is certainly more suitable for these cases.

The overall ideology of the whiteboard brings data out of the agents. By this we mean both the control data (commands) and internal states of the agents (e.g. active, idle, ten commands pending). This allows several interesting possibilities. For example, we can create self-monitoring agents for the whiteboard systems that keep an eye on the activities of the primary agents. If the data is kept in a human-readable format, then system developers get a good insight about the actions of the agents by simply reading the contents of the whiteboard. The entire whiteboard represents the general state of the system and makes saving and reloading the state easier.

The solutions discussed in the thesis assume the presence of the Linux kernel. The kernel is the bottleneck one way or another, whether the sockets are used or anything else. According to the current experience the shared memory based whiteboard allows more efficient use of resources than the sockets. At the same time the present solutions have small footprints – which allows them to be fitted into embedded systems.

5.2. Summary of the contributions of thesis

The thesis studies the options for building multi-agent systems for controlling sensors and actuators in real time or near real time situations on embedded computing platforms. The background survey points out the popular technologies used for similar kinds of situations and explains the differences from the whiteboard.

The questions and options of different aspects of the whiteboard design are discussed, including the overall architecture for the whiteboard, multiple agents and communication of the agents. The thesis explains the motivations of choosing between available options in different situations.

As a result, two different implementations of the whiteboard emerge in two large projects from different domains: Roboswarm and Telemonitoring Gateway. The case studies of the projects are used for demonstrating the issues of implementing and using the whiteboard.

There is no single product that we refer by the term *whiteboard*. However, the generalized whiteboard is a fast shared memory based data store meant for exchanging knowledge between the agents which does not have to be explicitly structured.

5.3. Authors' contribution in the published articles

In the scope of the published papers 1 – 3 (listed at the beginning of the thesis, before the introduction), the author of this thesis was responsible for designing the software architecture and implementing the functional subsystems for RFID sensors, guiding the Roomba robots, using the tiny Linux system added to the Roombas along with several software components built by other authors (in-memory database, the reasoning engine, infrastructure for accessing the server). Overall, the author was responsible for implementing the core functionalities of the Roomba robots as designed in the project.

In the scope of the papers 4 – 13 above, the author was responsible both for the software architecture of the embedded devices, integrating the reasoning subsystem, developing the communication methods between the devices and the central server/data repository, as well as the actual implementation of the software on the devices. The user interface aspects, the central data server/repository and the medical aspects of the systems were in most cases not the focus of the author of this thesis.

In the scope of the paper 14, the author was responsible for the robot simulation subsystem and was actively involved in designing and evaluating the movement strategies.

However, the intellectual input of every author of the above papers should not be underestimated.

5.4. Future work

The research on the whiteboard solutions has many interesting follow-up directions stemming from the details and use cases discussed. To give additional insights we would like to name some of the most important directions.

The general use of reasoning algorithms and tools need further attention. One of the goals here is better integration of the reasoning engine with the whiteboard's data. Another direction in the reasoning subtopic is to search for different languages and corresponding tools for deriving new knowledge. The methods dealt in this work use first order logic, yet nothing prohibits us from considering alternative logical systems.

There is a huge potential of optimization and better performance if we can reduce the number of queries and subqueries. Therefore, further investigation is needed to find clever ways to link the data items and/or make the query mechanisms more efficient.

The whiteboard implementations we have presented rely on a single core tool. As long as it remains hidden from the user, the internal mechanism can basically make use of several tools. Suppose the standard RDF is used to encode the data items. Then it makes sense to add RDF tools and languages to the whiteboard for parsing and querying.

Another possible topic for the future is the distribution of the whiteboard. The architecture presented in this thesis does not consider the possibility of keeping the same whiteboard distributed on different machines. It should be noted that the problems of locking would become more complex and the distributed solution with several services performed over the network will be significantly slower than the performance of the single whiteboard running on shared memory. Nevertheless, for several use cases it will be useful or necessary to have the network services integrated into the whiteboard.

REFERENCES

[1] B. Hayes-Roth. A Blackboard Architecture for Control. Artificial Intelligence 26. Pages 251-321. 1985.

[2] Daniel D. Corkill. Blackboard and MultiAgent Systems & the Future. In Proceedings of the International Lisp Conference. 2003,

[3] D. D. Corkill. Blackboard systems. AI Expert, 6(9). Pages 40–47. 1991.

[4] Rick Cattell. Scalable SQL and NoSQL Data Stores. ACM SIGMOD Record archive, Volume 39, Issue 4. Pages 12-27. 2010.

[5] Serge Abiteboul. Querying Semi-Structured Data. Database Theory — ICDT '97 Lecture Notes in Computer Science, Volume 1186. 1997.

[6] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In ACM SE Proceedings of the 48th Annual Southeast Regional Conference. 2010.

[7] Liliana Ferreira and Pedro Ambrosio. Towards an Interoperable Health-Assistive Environment: the eHealthCom Platform. Proceedings of the In Proceedings of the IEEE-EMBS International Conference on Biomedical and Health Informatics. Pages 930 – 932. 2012.

[8] Yan Huang, Huiru Zheng, Chris Nugent, Paul McCullagh, Norman Black. A Decision Support System for Self-Management of Chronic Conditions. AMA-IEEE Medical Technology Conference. 2011

[9] Kováč Miroslav, Lehocki Fedor, Valky Gabriel. Multi-Platform Telemedicine System for Patient Health Monitoring In Proceedings of the IEEE-EMBS International Conference on Biomedical and Health Informatics. Pages 127 – 130. 2012.

[10] Jon Noronha, Eric Hysen, Haoqi Zhang, Krzysztof Z. Gajos. PlateMate: Crowdsourcing Nutrition Analysis from Food Photographs. In UIST '11 Proceedings of the 24th annual ACM symposium on User interface software and technology. Pages 1-12. 2011.

[11] Michael Compton, Cory Henson, Laurent Lefort, Holger Neuhaus, and Amit Sheth. A Survey of the Semantic Specication of Sensors. 2nd International Workshop on Semantic Sensor Networks. A workshop of the 8th International Semantic Web Conference. 2009.

[12] N.H. Lovell, B.G. Celler, J. Basilakis, F. Magrabi, K. Huynh, M. Mathie. Managing chronic disease with home telecare: a system architecture and case study. In Proceedings of The Second Joint EMBS-BMES Conference (Engineering in Medicine and Biology with Annual Fall Meeting of the Biomedical Engineering Society). Pages 1896 – 1897. 2002.

[13] George W. Beeler. HL7 Version 3—An object-oriented methodology for collaborative standards development. International Journal of Medical Informatics, 48. Pages 151–161. 1998.

[14] R. H. Dolin, L. Alschuler, F. Behlen, P. V. Biron, S. Boyer, D. Essin, L. Harding, T. Lincoln, J. E. Mattison, W. Rishel, R. Sokolowski, J. Spinosa, J. P.

Williams. HL7 document patient record architecture: an XML document architecture based on a shared information model. In Proc AMIA Symp. Pages 52–56. 1999.

[15] Nicholas R. Jennings and Michael Wooldridge. Agent-Oriented Software Engineering. Artificial Intelligence, Volume 117. Pages 277 – 296. 2000.

[16] D. Rudenko and A. Borisov. An Overview Of Blackboard Architecture Application For Real Tasks. Scientific Proceedings Of Riga Technical University. 2007.

[17] Tim Finin, Yannis Labrou, James Mayeld. KQML as an agent communication language. MIT Press. 1995.

[18] Amit K. Chopra and Munindar P. Singh. An Architecture for Multiagent Systems: An Approach Based on Commitments. In Proceedings of the AAMAS Workshop on Programming Multiagent Systems. 2009.

[19] B.P. Gerkey, R.T. Vaughan, K. Stoy, A. Howard, G.S. Sukhatme, M.J. Mataric. Most valuable player: a robot device server for distributed control. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems. Pages 1226-1231. 2001.

[20] B. Gerkey, K. Stoy, R. T. Vaughan. Player robot server. Tech. Rep. IRIS-00-392. Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California. 2000.

[21] B.P. Gerkey, R.T. Vaughan, A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In Proceedings of the International Conference on Advanced Robotics. Pages 317-323. 2003.

[22] Toby H. J. Collett and Bruce A. Macdonald. Player 2.0: Toward a practical robot programming framework. In Proc. of the Australasian Conference on Robotics and Automation. 2005.

[23] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams. Towards Component-Based Robotics. In Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems. Pages 163 – 168. 2005.

[24] Sang Chul Ahn, Jin Hak Kim, Kiwoong Lim, Heedong Ko, Yong-Moo Kwon, Hyoung-Gon Kim. UPnP Approach for Robot Middleware. In Proceedings of the IEEE International Conference on Robotics and Automation. Pages 1959 – 1963. 2005.

[25] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y. Ng. ROS: an open-source Robot Operating System. ICRA Workshop on Open Source Software. 2009.

[26] Michael Montemerlo, Nicholas Roy, Sebastian Thrun. Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit. In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems. Pages 2436-2441. 2003.

[27] Hans Utz, Stefan Sablatnög, Stefan Enderle, Gerhard Kraetzschmar. Miro—Middleware for Mobile Robot Applications. IEEE Transactions on Robotics and Automation, Vol. 18, No. 4. 2002.

[28] Y. Ono, H. Uchiyama, W. Potter. A Mobile Robot For Corridor Navigation: A Multi-Agent Approach. In Proceedings of the 42nd annual Southeast regional conference. Pages 379 – 384. 2004.

[29] Sebastian Petters, Dirk Thomas, Oskar von Stryk. RoboFrame - A Modular Software Framework for Lightweight Autonomous Robots. In Proc. Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems. 2007.

[30] Ayssam Elkady and Tarek Sobh. RoboticsMiddleware: A Comprehensive Literature Survey and Attribute-Based Bibliography. Journal of Robotics. Hindawi Publishing Corporation. 2012.

[31] Nader Mohamed, Jameela Al-Jaroodi, Imad Jawhar. Middleware for Robotics: A Survey. In Proc. of The IEEE Intl. Conf. on Robotics, Automation, and Mechatronics. Pages 736-742. 2008.

[32] F. Magrabi, N.H. Lovell, K. Huynh, B.G. Celler. Home telecare: system architecture to support chronic disease management. In Proceedings of the IEEE 23rd Annual International Conference of Engineering in Medicine and Biology Society. Pages 3559 – 3562. 2001.

[33] Eunme Cha, Jeffrey Wood, Joseph Finkelstein. Using Gaming Platforms for Telemedicine Applications: A Cross-Platform Comparison. In Proceedings of the IEEE-EMBS International Conference on Biomedical and Health Informatics. Pages 918 – 921. 2012.

[34] P. Hanák, N. Kiss, T. Kovácsházy, B. Pataki, M. Salamon, Cs. Seres, Cs. Tóth, J. Varga. System Architecture for Home Health and Patient Activity Monitoring. In IFMBE Proceedings of the 5th European Conference of the International Federation for Medical and Biological Engineering, Volume 37. Pages 945-948. 2012.

[35] Robert Love. Get on the D-BUS. Linux Journal, Issue 130. 2005

[36] F. Paganelli, D. Giuli. An Ontology-based Context Model for Home Health Monitoring and Alerting in Chronic Patient Care Networks. In Proc. of International Conference on Advanced Information Networking and Applications Workshops. Pages 838 – 845. 2007.

[37] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. Computing Surveys, 12(2). Pages 213–253. 1980.

[38] H. P. Nii, E. A. Feigenbaum, J. J. Anton, and A. J. Rockmore. Signal-to-symbol transformation: HASP/SIAP case study. AI Magazine, 3(2). Pages 23–35. 1982.

[39] C. Metzner, L. Cortez, D. Chacin. Using A Blackboard Architecture In A Web Application. The Journal of Issues in Informing Science and Information Technology, Volume 2. Pages 743-756. 2005.

[40] R. van Liere, J. Harkes, W. de Leeuw. A Distributed Blackboard Architecture For Interactive Data Visualization. In Proceedings of the conference on Visualization. Pages 225 – 231. 1998.

[41] Michael Wooldridge and Paolo Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In Agent-Oriented Software Engineering, Volume 1957. Pages 55-82. 2001.

[42] Yoav Shoham. Agent-oriented programming. Artificial Intelligence, Volume 60. Pages 51-92. 1993.

[43] Roberto A. Flores-Mendez. Towards the Standardization of Multi-Agent Systems Architectures: An Overview. ACM CROSSROADS STUDENT MAGAZINE, Volume 5. Pages 18—24. 1999.

[44] Michael Wooldridge, Nicholas R. Jennings, David Kinny. A Methodology for Agent-Oriented Analysis and Design. In Proceedings of the third annual conference on Autonomous Agents. Pages 69 – 76. 1999.

[45] Henk W.M. Gazendam and René J. Jorna. Theories about architecture and performance of multi-agent systems. Tech. rep., SOM research report 98A02, Groningen, NL. 1998.

[46] Onn Shehory. Architectural Properties of MultiAgent Systems. Technical Report CMU-RI-TR-98-28. The Robotics Institute, Carnegie Mellon University. 1998.

[47] John R. Graham, Keith S. Decker, Michael Mersic. DECAF - A Flexible Multi Agent System Architecture. Autonomous Agents and Multi-Agent Systems, 7(1-2):727. 2003.

[48] V. Julian and V. Botti. Developing real-time multi-agent systems. Integrated Computer-Aided Engineering, Vol. 11. IOS Press. Pages 135-149. 2004.

[49] Edmund H. Durfee, Jeffrey S. Rosenschein. Distributed Problem Solving and Multi-Agent Systems: Comparisons and Examples. AAAI Technical Report WS-94-02. 1994.

[50] P D O'Brien and R C Nicol. FIPA — towards a standard for software agents. BT Technology Journal, Volume 16, Number 3. Pages 51-59. 1998.

[51] Yannis Labrou. Standardizing Agent Communication. Mutli-agents systems and applications. Pages 74 – 97. 2001.

[52] Jose M. Vidal and Paul Buhler. A Generic Agent Architecture for Multiagent Systems. USC CSCE. 2002.

[53] Pablo R. Fillottrani. The multi-agent system architecture in SEWASIE. Journal of Computer Science & Technology, Vol. 5, no. 4. Pages 225-231. 2005.

[54] John Hunt. Blackboard Architectures. JayDee Technology Ltd., Corsham, UK. 2002.

[55] Simon Parsons, Tim Brown, Simon King, E. H. Mamdan. A blackboard system for active decision support in configuring telecommunication services. In Proceedings of the 13th International Conference on Artificial Intelligence, Expert Systems and Natural Language. 1993.

[56] Marc Cavazza, Steven J. Mead, Alexander I. Strachan, Alex Whittaker. A Blackboard System for Interpreting Agent Messages. From: AAAI Technical Report SS-01-02. 2001.

[57] Christian Boitet and Mark Scligman. The "Whiteboard" Architecture: A Way to Integrate Heterogeneous Components of Nlp Systems. In Proceedings of the 15th conference on Computational linguistics, Volume 1. Pages 426-430. 1994.

[58] Jing Dong, Shanguo Chen, Jun-Jang Jeng. Event-Based Blackboard Architecture for Multi-Agent Systems. In Proceedings of International Conference on Information Technology: Coding and Computing. Pages 379 – 384. 2005.

[59] Kwame Wright, Kartik Gopalan, Hui Kang. Performance Analysis of Various Mechanisms for Inter-process Communication.

[60] Malcolm D. Brown and Robert B. Fisher. A Distributed Blackboard System for Vision Applications. University of Edinburgh. 1989.

[61] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, Stefan Manegold. Column-store support for RDF data management: not all swans are white. In Proceedings of the VLDB Endowment, Volume 1, Issue 2. Pages 1553-1563. 2008.

[62] Y. Papakonstantinou, H. Garcia-Molina, J. Widom. Object exchange across heterogeneous information sources. In Proceedings of the Eleventh International Conference on Data Engineering. Pages 251 – 260. 1995.

[63] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, Janet L. Wiener. The Lorel Query Language for Semistructured Data. International Journal on Digital Libraries, Volume 1, Number 1. Pages 68-88. 1997.

[64] Steve Harris. SPARQL query processing with conventional relational database systems. Web Information Systems Engineering – WISE Workshops Lecture Notes in Computer Science, Volume 3807. Pages 235-244. 2005.

[65] Stephen Harris, Nicholas Gibbins. 3store: Ecient Bulk RDF Storage. In Proc. of PSSS'03. Pages 1–15. 2003.

[66] Ching-Long Yeh and Ruei-Feng Lin. Design and Implementation of an RDF Triple Store. In Proceedings of the First Workshop of Digital Archive Technology. Taipei, Taiwan. 2002.

[67] Wolfgang Schramm, Harald Köstinger, Klaus Bayrhammer, Michael Fiedler and Thomas Grechenig. Developing a Hospital Information System Ecosystem for Creating new Clinical Collaboration Methodologies. In Proceedings of the IEEE-EMBS International Conference on Biomedical and Health Informatics. Pages 101 – 103. 2012.

[68] Obinna Anya, Hissam Tawfik, Atulya K. Nagar, Khalid M. Lootah. A Framework for Practice-Centred Awareness and Decision Support in Pervasive E-Health. In Proceedings of the IEEE-EMBS International Conference on Biomedical and Health Informatics. Pages 937 – 940. 2012.

[69] Abdul Hakim H. M. Mohamed, Hissam Tawfik, Lin Norton, Dhiya Al-Jumeily. Does e-Health technology design affect m-Health informatics acceptance? A case study. In Proceedings of the IEEE-EMBS International Conference on Biomedical and Health Informatics. Pages 968 – 971. 2012.

Appendix 1 - Interprocess communication

There are several possible solutions available for transferring data from one Linux process to another. It is quite clear that different technologies yield diverse performance characteristics and have unlike interfaces. But it is hard to tell the exact suitability of the technologies for the given application because the performance depends on the actual usage patterns by the application. Therefore some comparative tests could prove useful. The following gives a short overview of the related experiments performed while designing the whiteboard.

The most popular method of choice for transferring data is TCP/IP sockets. However, there are alternative possibilities similar to the sockets, namely pipes, queues and UNIX domain sockets. To give a reference point, we include some database engines into our tests - Sqlite and PostgreSQL – storing data in files and the QDBM key-value base also using files for storage. From the set of tools utilizing the shared memory we examine the Roboswarm database and Wgandalf.

All these tools have different interfaces (APIs) which makes the straightforward comparison impossible. The UNIX domain sockets, TCP/IP sockets and pipes allow reading and writing byte streams. The message queues operate in terms of structures. The SQL databases can be used for transferring data by inserting and deleting records. The custom-made shared memory databases have their own commands for the same purposes.

Hence, the comparison experiment was designed in the simplest possible way to satisfy the most general common functionality of all the tools. The main idea was to test message passing between Linux processes and measure how much time it takes for the message (some piece of knowledge) to travel from the source to the destination. The time mesaured includes also the composition on the sender's side and decoding the raw message on the receiver's side. In other words, it is the time spent to get the piece of knowledge from end point to end point.

Every message contains the same kind of information. The message has four or five fields. There is one field for the sequentially increasing ID number, two fields for the timestamp (seconds and microseconds), and the ballast field. The fifth field is the channel identification which is needed, for example, by shared memory databases while the sockets establish the connection between the sender and the receiver beforehand. This field has nothing to do with the content of the message.

The basic scenario is as follows: the sender process generates a new message and checks the current timestamp with the microseconds precision; then it forms and sends the message according to the protocol of the tool (e.g. SQL statement or byte sequence) and embeds the timestamp; the receiver waits or polls for the incoming messages and parses them on arrival; after that it also checks the current system timestamp. By subtracting the timestamp of the generation of the message (parsed from the contents) from the timestamp of the reception of the message the receiver process determines the "time of flight" for every single message which it then aggregates to find the average and worst case values. Unlike the socket-like tools, the receiver built upon the databases has to delete the processed messages to avoid flooding the database. This is done as a part of the parsing phase.

There are some available parameters for adjusting the test details. By varying the values of the parameters we can run the sender and receiver programs under different conditions to get a better coverage of the domain. To name the basic parameters, the number of *parallel channels* determines how many sender-receiver pairs are executed simultaneously. The *number of tests* tells the programs how many messages they have to transfer. The *sleep* between two messages affects the frequency of message transfer. The *ballast* is the size of the ballast field. Some parameters are tool specific, for example whether to run the database on RAM disk or HDD, to use queries or scanning, to turn on or off Sqlite pragmas, etc.

The majority of the tests were conducted on an average laptop PC running Linux in a virtual machine. Additionally, dome different platforms were used as well. The processor of the main PC is Intel Core 2 Duo at 2,26GHz. Different Linux installations were used: Ubuntu 8 with kernel 2.6.24, Ubuntu 10 with kernel 2.6.32, and Kubuntu 12 with kernel 3.2.0 – but as there were no remarkable differences between these we will not indicate the kernel version on the following charts.

All the time values in the tables and charts are in microseconds (us). For every test there are two types of values: the average and the maximum. These are the final results of two levels of aggregation. At first, a receiver program measures transfer times for individual messages (say 5000, as in the Table 1) and calculates the average time plus the maximum time. Observe that there are many pairs of senders and receivers running simultaneously in the tests (called parallel channels, e.g. 5 in Table 1). After all the parallel copies of the test programs have finished and outputted their results, the overall average is calculated from the individual averages and the same is done for the maximum. These values appear in the tables. The global minimum value was collected in the same manner, but as it does not yield much information, it is is omitted from the tables. The minimum values were very small and static in the majority of the cases.

In the tables *IPC* (Interprocess Communication) stands for UNIX domain sockets. *TCP* stands for the ordinary TCP/IP sockets on the loopback interface ("localhost"). By *Roboswarm* we mean the custom made shared memory database used in the Roboswarm project.

A few notes on the *sleep* parameter: this value is kept fairly small and always the same in all tests presented. The reason is our interest to benchmark the tools under tight conditions. If we were to consider low frequency data streams with no time constraints, all the tools would perform adequately. Omitting the sleep completely and letting the operating system's scheduler to solely manage the racing, on the other hand, gave worse results, especially when the number of parallel channels was high.

In the Table 1 the basic proportions of the tools are clearly draw out. The shared memory tools are significantly faster than the classical sockets. The Linux message queues give the best results, but not so much better when compared to the shared memory databases. The problem with the queues is that the senders and receivers must use predefined structures and the length of the message has to be known by both.

Number of tests: 5000 Sleep: 1000 us Ballast: 30 bytes Parallel channels: 5		
Tool	Average (us)	Maximum (us)
Pipe	292 672	558 068
IPC	108 371	232 232
TCP	112 645	248 741
Roboswarm	13 772	96 745
Queue	1 463	26 592
Wgandalf	12 668	308 231

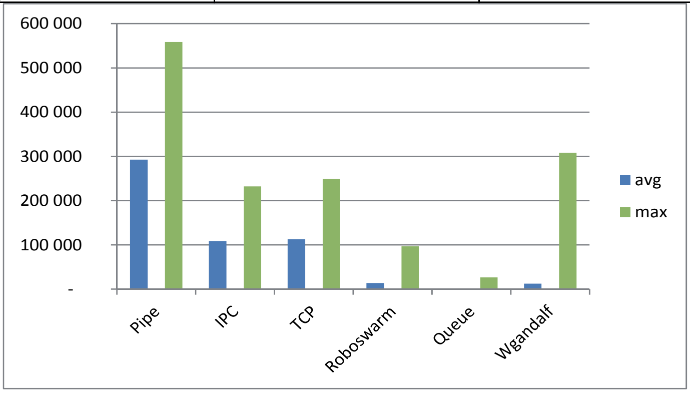

Table 1 – Test results for the shared memory and socket-like tools

Increasing the ballast size has an effect on the tools based on byte streams like the pipes and sockets as shown in the Table 2.

	Tests 1000, sleep 1000, parallel channels 5					
	ballast 100		ballast 200		ballast 500	
Tool	avg	max	avg	max	avg	max
Pipe	127650	278184	392292	607539	395545	487567
IPC	194157	431972	280108	463088	346823	614854
TCP	176049	385762	416376	789639	1080481	2075322
Robosw.	3410	51540	2370	22387	2505	32290
Queue	1153	15470	1123	17164	1333	19222
Wgandalf	1796	21348	2032	20232	2715	25263

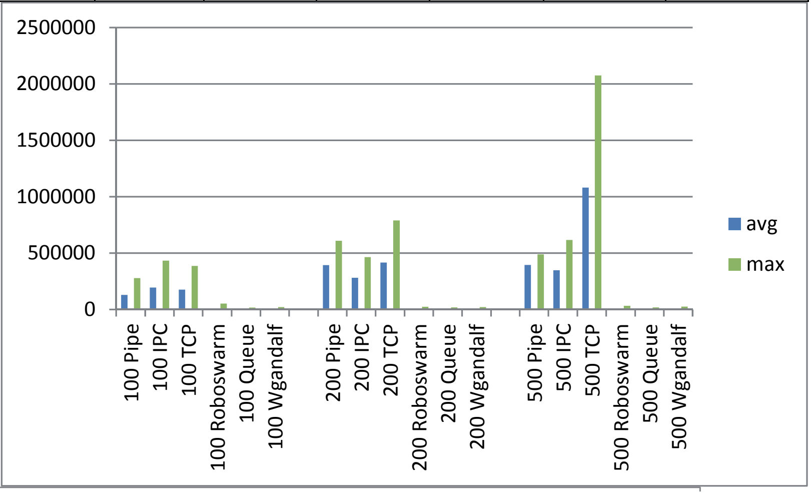

Table 2 – The effect of the ballast size

The large ballast and the high number of parallel channels increase load, thus driving the message transfer times up. The bottleneck appears to be the kernel, especially for the socket-like tools, including IPC which is designed for such types of tasks. The Table 3 shows that the socket-like tools respond to the rise of parallelism more or less linearly. Shared memory tools perform well. Notice that Wgandalf gives better results with smaller loads than the Roboswarm's database but experiences problems under heavy use. This is especially the case with the maximum value – the worst case message delivery time from the source to the destination. 20 channels mean that there are 20 receiver processes and 20 sender processes, running really fast paced. In real life situations there is usually no need for a so high number of high-speed channels.

	Tests 1000, sleep 1000, ballast 30					
	10 channels		20 channels		30 channels	
Tool	avg	max	avg	max	avg	max
Pipe	236104	404068	1027645	1788553	1526335	3001597
IPC	58748	146407	615494	1822617	965838	2191074
TCP	183633	424815	1062106	1966551	1789346	3467861
Roboswarm	4939	40619	48182	571396	174902	965498
Queue	706	17012	2457	32010	6246	67347
Wgandalf	3160	30042	4405	19430	381002	1553501

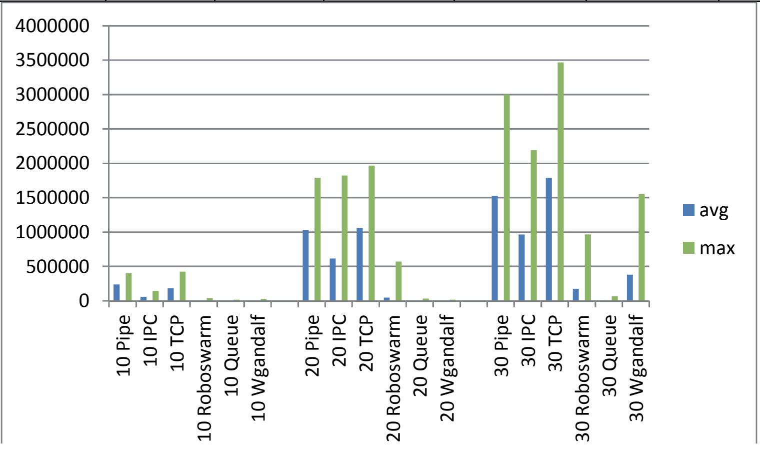

Table 3 – Increasing load by adding more parallel instances

The Table 4 introduces three additional tools, namely the Sqlite database, PostgreSQL database and QDBM. Sqlite was run on ramdisk, PostgreSQL both on ramdisk and regular hard drive. For Postgres the ramdisk yields 12% to 30% faster times when compared to standard disk files. This effect, however, does not matter much since the performance numbers are far from the previously discussed tools. Sqlite is not meant for heavy parallel use and thus gives the worst results in the context of these tests. QDBM comes close to the queues which is a rather encouraging finding. QDBM is a key-value database and uses a hash table. When we would have to use more complex entries instead of the simple test messages, it would need a lot of additional program code (in a whiteboard implemented on top of the QDBM) and is expected to lose its advantages.

	Tests 5000, sleep 1000, ballast 30, channels 5	
Tool	avg	max
Pipe	142 933	672 916
IPC	130 333	612 483
TCP	232 134	1 513 342
Roboswarm	3 010	25 424
Queue	1 534	12 532
Wgandalf	5 268	19 964
Sqlite	9 277 068	20 995 476
Postgres	5 939 256	8 385 902
QDBM	1 727	15 849

Table 4 – Adding Sqlite, Postgres and QDBM

The Table 5 shows the results of the same tests on an embedded hardware – Chumby media display, kernel 2.6.28, Marvell Mohawk 800MHz processor.

Tests 2000, sleep 1000, ballast 30, channels 5		
Tool	avg	max
Pipe	928 713	1 409 113
IPC	490 104	894 036
TCP	1 061 330	1 638 136
Roboswarm	15 275	167 874
Queue	252	7 871
Wgandalf	1 873	28 564

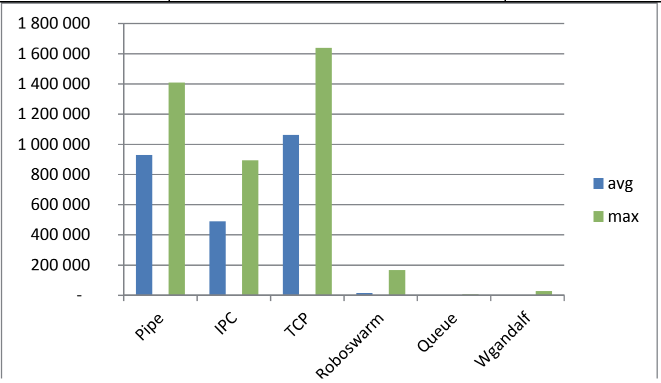

Table 5 – Benchmark results on Chumby

While the intended target architectures are the low-power Chumby-like platforms, it is still interesting to have some insights for more powerful hardware. The Table 6 shows the test results got from the same virtual machine that was used before, but copied now onto the physical machine with a Core I7 3720QM processor. Notice that the parameters are the same as in the Table 5. The familiar patterns are still visible while the differences between tools are much smaller. Surprisingly, Sqlite outperforms Postgres.

Tests 2000, sleep 1000, ballast 30, channels 5		
Tool	avg	max
Pipe	124 326	484 648
IPC	5 144	22 271
TCP	13 639	58 739
Roboswarm	920	20 155
Queue	900	12 291
Wgandalf	2 093	15 612
Sqlite	5 170	23 316
Postgres	166 683	285 173

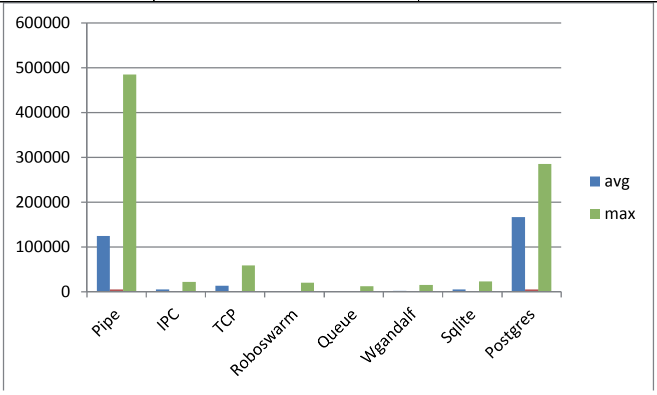

Table 6 – On high-performance hardware, I7 3720QM

Appendix 2 – Scalability of tuples and triples

In this appendix we will present test results of another experiment, this time focusing on the Wgandalf shared memory database as the underlying tool. The basic scenario and measuring policies are exactly the same as described in the previous appendix. The idea of the current test is to benchmark the ability of the tool to withstand subqueries. Hence we measure two different cases (a) encoding all *n* attributes of the message into a single Wgandalf record (tuple), and (b) creating *n* separate records, each with three fields (triples).

It is clear that the schema using tuples is more efficient, both performance- and memory-wise. The schema using triples requires more work on the sender side and requires performing a tedious procedure of subqueries to collect all the pieces of the original message on the receiver's side. On the other hand, the tuples do not possess the flexibility for extending the existing data objects. The tests try to compare the two schemas in the real situation and assess the actual handicap of the triples.

The mechanisms of test and data aggregation are the same as in the previous appendix. The same holds for the test platforms on which the programs were executed. However, there is one thing that needs to be clarified: the meaning of the *ballast* parameter is changed. While previously it meant a string of *n* bytes, now it means *n* attributes. For every attribute there are two fields saved into the record: the name of the attribute and the value. The message has the number of attributes indicated by *ballast* and four extra attributes which have no explicit name field: message ID, channel ID, seconds, and microseconds. For example, the *ballast* 10 means that the tuple type message has 24 fields (4 extra fields and 2 fields per 10 ballast attributes), the triples type message has 14 records (4 triples for extra attributes, 1 triple for every ballast attribute, including name and value).

To get the idea of the worst case times there are no special optimization tricks used in the case of triples schema. All the records have exactly three fields and triples are linked through the contents. Technically it would be possible to create more fields to records (which are hidden from the user) and use direct pointers to link records. However, the purpose of the given test is to let the tool run in the simplest setup and handle an extensive amount of subqueries.

The Table 1 presents the test results with some combinations of parameters in the terms of absolute values. The Table 2 does the same in terms of ratio values. Again, with small loads (not much parallelism) the triples schema does not cause serious problems by falling far behind the tuples. But when the difference occurs, it can be of the magnitude of several hundred times. In the Table 2 the numbers are computed by dividing the average triples value by the average tuples value at the same parameter values. In some rare cases the ratio appears to be below one, which means that the triples schema gave faster result than the tuples schema. This does not, however, change the overall picture.

The Table 3 and the Table 4 give the benchmark results for Chumby and a high-performance platform, respectively. As expected, with lots of processing power there is no penalty when using triples and in the case of low processing power the extensive subquerying severely handicaps the performance.

Tests 1000					
Sleep	Ballast	Channels	Type	Avg (us)	Max (us)
500	2	3	tuples	2 129	27 938
500	2	3	triples	3 178	9 162
500	20	5	tuples	2 015	12 450
500	20	5	triples	10 437	126 508
500	20	10	tuples	4 285	53 554
500	20	10	triples	1 019 992	3 923 560
1000	10	5	tuples	1 481	25 430
1000	10	5	triples	3 654	12 706
1000	20	10	tuples	2 137	7 341
1000	20	10	triples	60 196	517 931

Table 1 –Test results on Core 2 Duo

Ballast	Channels	1000 tests 500 sleep	1000 tests 1000 sleep	1000 tests 1500 sleep
2	3	1,5	5,0	4,0
5	3	1,2	1,2	1,4
10	3	1,7	0,9	0,9
20	3	1,6	1,5	1,4
2	5	3,2	1,6	1,3
5	5	2,2	3,9	2,4
10	5	5,2	2,5	0,5
20	5	5,2	0,9	1,1
2	10	8,2	3,8	3,2
5	10	39,8	5,2	8,6
10	10	425,2	6,5	6,2
20	10	238,0	28,2	12,3

Table 2 – Triples compared to tuples (Core 2 Duo)

Tests 1000, sleep 1000, ballast 10	channels	type	avg	max
	5	tuples	45 515	129 116
	5	triples	2 033 755	12 484 603
	10	tuples	64 409	419 950
	10	triples	24 382 047	75 744 921

Table 3 – On Chumby

Tests 1000, sleep 1000, ballast 10	channels	type	avg	max
	5	tuples	1 616	5 350
	5	triples	2 679	14 997
	10	tuples	2 835	9 974
	10	triples	44 212	139 898

Table 4 – On Core I7

Elulugu

Nimi	Enar Reilent
Sünniaeg ja -koht	22. august 1984, Tallinn, Eesti
Aadress	Metsa 19, 11616 Tallinn
Telefon	+37255938878
e-post	e.reilent@gmail.com

Haridus:

1991 – 2003	Tallinna Nõmme Gümnaasium
2003 – 2008	Tallinna Tehnikaülikool, informaatika magister

Keeleoskus:

Eesti	Emakeel
Inglise	Kõrgtase

Teenistuskäik:

Alates 2006	Eliko Tehnoloogia Arenduskeskus, teadur ja tarkvarainsener

Teadustöö põhisuunad:

Multiprotsess-arhitektuurid kontrollerites;
Ühismälul töötavad andmebaasid;
Loogika ja reeglimootorite kasutamine juhtimisel;
Sensori mõõteandmete semantiline kirjeldamine.

Curriculum Vitae

Name	Enar Reilent
Born	August 22, 1984, Tallinn, Eesti
Aadress	Metsa 19, 11616 Tallinn
Phone	+37255938878
e-mail	e.reilent@gmail.com

Education

1991 – 2003	Tallinn Nõmme Upper Secondary School
2003 – 2008	Tallinn University of Technology, MSc in Computer Science

Language skills

Estonian	native language
English	advanced level

Professional employment

Since 2006	Eliko Competence Centre, researcher and software engineer

Scientific research topics

Multiprocess software architectures for embedded systems;
Shared memory database systems;
Formal logic and reasoning engines for control;
Semantic descriptions of sensor data.

PAPER 1

T. Tammet, J. Vain, A. Puusepp, E. Reilent, A. Kuusik. RFID-based communications for a self-organizing robot swarm. In: Proceedings Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2008: 20-24 October 2008, Venice, Italy: (Toim.) Brueckner, Sven; Robertson, Paul; Bellur, Umesh. Los Alamitos, Calif.: IEEE Computer Society, 2008, 45 - 54.

Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems

RFID-based Communications for a Self-Organising Robot Swarm

Tanel Tammet, Jüri Vain,
Andres Puusepp, Enar Reilent
Department of Computer Science,
Tallinn University of Technology
Ehitajate tee 5, 19086 Tallinn, Estonia
tammet@staff.ttu.ee, vain@ioc.ee,
e.reilent@gmail.com, anduoma@hot.ee

Alar Kuusik
Department of Electronics,
Tallinn University of Technology
Ehitajate tee 5, 19086 Tallinn, Estonia
kalar@va.ttu.ee

Abstract

We investigate the practical questions of building a self-organising robot swarm, using the iRobot Roomba cleaning robot as an experimental platform. Our goal is to employ self-organisation for enhancing the cleaning efficiency of a Roomba swarm. The implementation uses RFID tags both for object and location-based task recognition as well as graffiti- or stigmata-style communication between robots. Easily modifiable rule systems are used for object ontologies and automatic task generation. Long-term planning and central coordination are avoided.

1 Introduction

The concept of a robot swarm denotes a large number of relatively simple physically embodied agents designed in a way that the desired collective behaviour emerges from the local interactions of agents and the interactions between the agents and the environment. The swarms are meant to perform a wide range of tasks which are infeasible to accomplish by a single robot. Their application ranges from simple cleaning tasks to exploration of large unknown areas, surveillance, rescue, coordinated weight lifting, minesweeping etc. where intervention from human operators is minimized.

The goal of a swarm mission can be considered generally as an integrated service provided by the swarm members collectively over a given period of time. Since swarms typically act in a dynamic and partially observable environment, the service requires repetitive and coordinated action by the swarm members throughout the mission.

The overall goal of our project is to develop simple and low-cost technologies for making both single robots and swarms of robots more intelligent. We use the dynamic cleaning problem [1], [7] as a testbed for the developed knowledge architecture, focusing on making swarm cleaning more efficient.

The crucial part of the project is to achieve the efficient cooperative behaviour of robots without any central coordination and planning.

That, again, requires propagation of understandable and reusable information among the robots which may be different in hardware and software. The target goal can be called ”knowledge centric” architecture approach focusing on uniformal (or easily convertible) on-robot and inter-robot data management.

We use ordinary passive RFID chips for marking objects like chairs, walls, doors. This is significantly cheaper and more flexible than using cameras on robots for object recognition. The same RFID chips on objects are also used by the robots to leave messages to other robots. The solution is inspired by ants' communication using pheromone trace known as stigmery. The usage of RFID tags reduces the communication overhead related with coordination significantly [9].

We use a popular iRobot Roomba cleaning robot and attach a tiny ARM-based Gumstix computer (500 MIPS computing power) using a BusyBox 2.6 Linux distribution (without real-time capabilities) and a stock RFID reader/writer on the Roomba. The attached computer takes over control of the Roomba. While a standard Roomba is fairly simple-minded, will clean places recently cleaned and does not understand that some places should be avoided - or vice versa, cleaned often - our system adds necessary intelligence.

First, Roombas understand object descriptions and simple messages written on RFID chips by humans: like ”go away”, ”fragile”, ”clean here”, ”this is a chair” etc. When the robot notices an RFID chip ahead, it will read its content and behave accordingly, following the configurable rules on board. The rule engine uses ontologies and allows the robot to understand, for example, that chair is a furniture and you can probably go around furniture.

Second, Roombas write their own messages on RFID chips. For example, when the robot notices an RFID tag while cleaning, it will write on the tag that it was cleaning there at that particular time. Next time when it comes near the same tag, it will not clean the place, unless enough time has passed. What is more important, when we use a whole swarm of Roombas for cleaning, all the other Roombas will also avoid cleaning on this marked up place for some time, avoiding wasted work. Similar optimizations are achievable for spreading our swarm members to different rooms, mapping the area, etc.

2 Robot control architecture

The architecture for the robot control is based on a layered multi-agent system, with agents implemented as continuosly running processes, and contains three layers (figure 1):

- The sensor-actuator access layer dedicated to communication with robot control hardware. The lowest part of robots sensor-actuator layer is executed by the iCreate onboard microcontroller. The external service time of this microcontroller was set to 20ms as shortest allowed period. Tests showed that the core agent communication solution did not add any additional mentionable response delays.
- The control layer that includes usual short-term planning and behavioural layer tasks. Merging two traditionally separate layers is reasonable due to the fact that the swarm robots (e.g. cleaning devices, room patrols) are relatively simple and the number of different behaviours is rather limited.
- The knowledge layer that targets reasoning (deriving new information from acquired data), communicating with other robots (using RFID tags) and the optional central server (using WIFI, if available).

The layered architecture is built around a fast and transparent RDF database implemented in shared memory. The RDF database realizes a core for interprocess communication of several on-board agents (processes), in particular the Main Control Agent (Central Control Process, control layer), Sensor Agent, Actuator Agent (sensor-actuator access layer), Reasoner Agent (knowledge layer). Moreover, the RDF database can manage the inter-robot communication using different functions/technologies for sending data to other robots (knowledge layer as well).

The internal knowledge architecture follows the classical blackboard model [4] In short, the agents communicate by

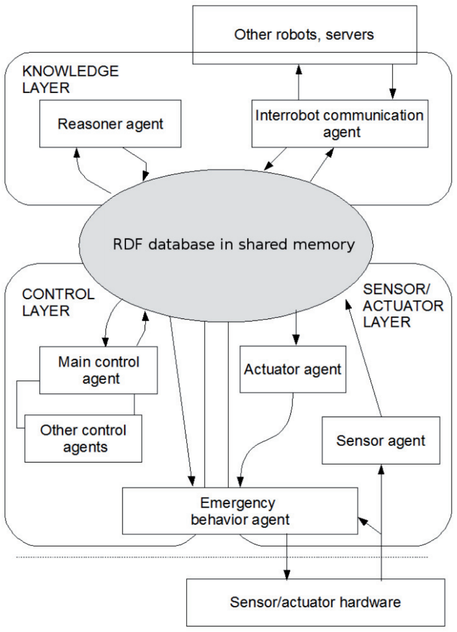

Figure 1. Robot architecture

writing data to the RDF database. The data on the RDF database is available to all agents.

The RDF database serves three roles:

- A postbox between different process agents (including external world communication).
- A fast and simple in-memory data store (circular buffer).
- A deductive database, using a rule language for rule-based generation of new facts.

Technologically the RDF database is built as a simple data store operating in shared memory. Shared memory based database approach is frequently used for low latency robot control architectures performing sufficiently well without real time OS. By our benchmarking tests performed with 500MHz 32 bit embedded ARM processor running non-real-time BusyBox 2.6 Linux distribution, the realized shared memory data store access time was in tens of microseconds.

3 Languages, common data model and the RDF database

The behaviour of the robot is primarily influenced by four players: sensors and control software, internal RDF database contents, RFID tags read, data and rule files read from the swarm server.

The swarm server collects data from the robots and influences them by sending new data and modifications to rule files in the robots.

The different players above use specialised language representations, all based on RDF. Different syntaxes stem from practical needs: for example, since RFID chips contain very little memory, we have to use a space-efficient encoding for information on RFID-s. On the other hand, communication between different servers does not require space efficiency: rather, it is preferable to use common, verbose XML-based standards.

We use the following RDF-based languages in the robot swarm system:

- Our specialised RDF encoding in RFID tags.
- Our specialised rule language for deriving and adding new data from/to the RDF database.
- Standard XML-based RDF syntax for data exchange between robots and the central server (using WIFI if available) and the central server and external systems.

All these languages/use cases share a common data model and the concrete strings for sensor and task representation (robot sensor/task language).

3.1 Common data model

The common data model is based on RDF triplets (proper data fields) to which we add two additional groups of data fields: contextual data fields and automatically generated metadata.

Proper data fields:

- Subject: id of whatever has the property.
- Property: name of the property of the subject.
- Object: value of the property.

The value field has an associated type, indicating the proper way of understanding the value. Observe that the property field typically - but not always - already determines the suitable or expected type.

In addition to basic RDF, we will always add three contextual fields to the beforementioned proper data fields of the triplet.

Contextual data fields:

- Date/time: when this fact held (in most cases same as the time of storing the data).
- Source: identifies the origin of the data (RFID nr, person id, other robot id, etc).
- Context: identifies a data group or addressee or indicates the succession of robot commands, often left empty.

Agents can enter their own contextual values to the RDF database. If no values are given by the agent, the default values (current date/time, robot id, empty context) are entered automatically.

Automatically generated metadata:

- Id: robot-unique id of the data row, auto-increased.
- Timestamp: date/time of storage.

Automatically generated metadata is present only in the RDF database, and not in the other data formats/languages. Agents cannot enter their own values at will. These two fields are important for efficient and convenient management of the data, and are used for example, by the reasoner.

Instead of using additional contextual and metadata fields we could have chosen to use reification of RDF triples to store the same information. However, this would have cumbersome and inefficient both in the internal RDF database used by agents inside the robot, and even more so in the data representation inside RFID chips, as described in the following chapters.

For data exchange between different swarms and external applications we will use the reified form of the contextual data fields, represented in the common XML syntax of RDF.

3.2 RDF database

The RDF database is implemented in the Gumstix computer on the robot as a library for storing and reading information to/from shared memory. Agents in the robot use a simple C API for writing, reading and searching data from the RDF database. Special RDF query languages are not used.

Strings in the RDF database are pointed to from the data fields: they are kept in a separate table, guaranteeing uniqueness: there is always only one copy of each string.

The data rows are organised as a circular list. The last data element will disappear when a new one is added. However, there are exceptions to this order: data items deemed critical are kept longer.

Although the data store should be normally seen as a mid-term memory, containing tens of thousands of rows (old data is thrown away), it is easy to use it as a postbox between different agents onboard: just put the name of the addressee agent in the context field and program the addressee agent to look for the rows with her name, process them and then delete them.

An agent X may also read ”messages” intended to an agent Y, but should under normal circumstances ignore these ”messages”: it should look for data rows with either no addressee at all or an addressee with the name X.

4 Data encoding on RFID tags

The roboswarm architecture requires recognising external objects/locations, reading location-specific messages/instructions from humans and reading/writing location-specific messages from/for other robots.

All these three tasks use RFID tags at different locations. The simplest types of RFID tags contain only the RFID id. However, we have been using RFID tags with a small internal memory: both human operators and robots can write information to the tags. We use the tags as information-carrying graffiti.

A human user is expected to write to a tag information like ”this tag is located on a chair”, ”this tag has coordinates X and Y”, ”there is a tag at direction R at distance 5 meters”, ”keep away from here” etc.

A robot N is expected to write to a tag information like ”N was here at 10.06.2007 at 15.10”, ”N did brush the surroundings at 17.10” etc.

4.1 Kinds of data on RFID chips: conceptual example

The following categorization gives a clearer picture of kinds of data to be written on RFID:

- Present by default on all RFID tags: built-in RFID id number, up to 12 bytes.
- Control information written to the tag by a human user:
  - Stop immediately.
  - Keep away from here.
  - Turn to direction X and move N meters.
  - Do not clean here.
- General information about objects:
  - What kind of object: wall, bed, chair, robot nr X.
  - This place needs cleaning very often.
  - Danger in direction X distance Y.
  - Some object (lift,docking,...) in direction X distance Y.
  - Robot nr X was here at time T and cleaned / could not clean / did not want to clean.
- Localisation of a tag, either:
  - Global, for example gps coordinates.
  - Local, relative to a given base vector: direction and distance.
  - Information about other tags in the neighbourhood: direction and distance.
  - Additional useful information: path to door, path to charger.
- Information about a robot: tag glued on a robot
  - I am a robot.
  - I am a robot nr X.
  - Kind/capabilities of a robot: simple cleaner / complex control robot.

We are using local coordinate vectors and special methods and algorithms for coordinate vector markup and finding. These methods are not covered in this paper.

4.2 Data encoding principles for RFID tags

RFID tags contain relatively little memory: we are currently using tags with 256 bytes. Reading of RFID data over wireless may be prone to error. Hence:

- The data format has to be extremely compact.
- Old data has to be regularly overwritten.
- There must be a way to indicate that some parts of data should not be overwritten by robots.
- We need a control sum for data blocks.

We use 32 bytes for encoding one data block, hence we can put 8 data blocks on our RFID tags.

Data encoded on tags must be easily understood both by robot software and external applications: software used by humans to read/write data to tags, agents different from the roboswarm components.

Hence we provide a simple mapping from RFID data to both robot internal data format and the generic RDF format for data.

All data items written to the RFID tags are essentially data rows with several predefined fields. All fields may contain different data items: strings, integers, floats. Hence the RFID data store is similar to a single database table.

Data is written, read and deleted one full row at a time: while changing stored rows is technically possible, we do not recommend doing that: it is better to add a new full row, and if necessary, delete the old row(s).

Conceptually, each data row corresponds to several RDF triplets. Standard RDF triplets contain the subject, property name and value fields, like this:

```
[12, performingaction, cleaning]
[12, notperformingaction, cleaning]
[15, lookingfortag , 244]
[15, notfoundtag , 244]
```

The subject field is normally filled with an id of an object which has the property with the value indicated. The value field may be filled either with a direct value or an id of some object (for example, a tag).

Using triplets will inevitably mean that recording one data item may require several triplets to be written. Suppose that a robot wants to write the message ”Robot nr 15 has been here at 14.20 on 28. January looking for tag nr 244 and did not find a tag while here.” on the RFID tag.

Using standard triplet format this would translate to the following triplet set:

```
[15, washereattime,
    14.20 on 28 January]
[15, waslookingfortag, 244]
[15, didnotfindtag, 244]
```

For RFID tags we always add timestamp, context and source (agent) id contextual fields. In our sextet data model the information would contain the following fields:

```
[15, 14.20 on 28 Jan, general,
  15,washereattime,14.20 on 28 Jan]
[15, 14.20 on 28 Jan, general,
  15, waslookingfortag, 244]
[15, 14.20 on 28 Jan, general,
  15, didnotfindtag, 244]
```

Data field contents are either direct (integers, RFID chip id-s) or indirect (identifiers of long strings):

- Direct values are put on the tag as-is.
- Long strings are not kept on the RFID, since we do not have enough space: we use a string number in a global string table instead.

The direct values are either 4 or 12 bytes long, depending on the type of a data block (see later sections). A direct value may either indicate one concrete measure (say, distance or time), contain a short string (up to 4 or 12 characters) or encode several short values, for example, a coordinate.

As the standard RDF format requires, the robot RDF database uses strings for identifying subjects and property names. RFID tags do not have enough space for long strings.

Hence we assume that a roboswarm has a common string table of predefined strings, where each string has a concrete number, common for all robots and RFID chips. Robots can certainly use more and dynamically created strings, but these strings will not be encodable on RFID chips.

Property names (but normally not property values) have a namespace prefix. We will commonly use http://www.roboswarm.eu/lang as a namespace for property names in the roboswarm. However, other namespaces may be used as well. For example, a full name string of a ”washereattime” property would be http://www.roboswarm.eu/lang#washereattime and this could be encoded as, say, number 135 in a common string table.

Identifiers for robots and humans are swarm-specific. We will use namespaces for these as well, however. The default swarm namespace for our experiments is http://www.roboswarm.eu/swarm. Concrete swarms may use different namespaces.

The roboswarm environment may potentially contain a huge number of different RFID tags. Old tags may be replaced, new ones may be glued on objects at any time. It would be impractical to assume that the robot software has predefined knowledge of all tags in the environment. Hence the RFID id on the tags is used ”as is”, without encoding it via a separate string table (see the next section). The robot software components will identify RFID tags by strings with the http://www.roboswarm.eu/RFID namespace followed by the hexadecimal encoding of the RFID id number.

As said before, the RFID data blocks contain both direct values (date/time, measures, coordinates, RFID id numbers, short strings) and numbers of strings in a common string table (external to chips)

The numbers in a string table start from number 0 and continue with numbers 1, 2 etc. We use 2 bytes for string table numbers, even if the field containg the string number is longer.

The global string table is loaded into the robot and has to be the same for the whole swarm. It is necessary only for coding and decoding data for the RFID tags.

4.3 RFID tag id numbers and special strings

RFID tags carry an id. The id size may vary. However, there are several widely used standards for product encoding, and most RFID tags are expected to conform to these standards:

- UPC (universal product code): 12 digit numbers identifiying product type, commonly used on bar codes.
- 64-bit EPC (electronic product code): 64 bit code identifying concrete items, forward compatible with a 96-bit version.
- 96-bit EPC (electronic product code): 96 bit code capable of identifying concrete items.

We use direct 96-bit EPC-s to identify RFID tags. Inside the robot software the RFID tags numbers are not used directly (as-is). Instead, they are encoded to identifier strings (uris) with the following algorithm: the initial part of the string is always a namespace prefix http://www.roboswarm.eu/rfid# and the following part of the string is formed from the RFID id number (of whichever length) by converting the number to a lower-case hex string in a conventional manner.

It is very common for a tag to contain information about its own location or the object it is glued to. In order to avoid putting the full 96-bit EPC into the subject id field, our string table contains contain a special string: http://www.roboswarm.eu/lang#me stands for the EPC of the RFID chip containing this data item.

4.4 Encoding details: data fields

We use 32 bytes for one data block (a sextet in our data model). We have two types of blocks. First type contains a short, 4-byte subject field and a long, 12-byte (96 bit) object field. Second type contains a long, 12-byte (96 bit) subject id field containing EPC and a short 4-byte object field.

Otherwise the structure and meaning of the data blocks is identical for both types:

- Blocktype 1 byte: contains block type nr, either 1 or 2.
- Agent 2 bytes: number of the agent string (robot, human, ...) writing data.
- Datetime 4 bytes: datetime of writing, according to the robot clock (up to one second), unix format.
- Context 2 bytes: number of the context string (adressee, data group, etc: often ignored)
- Subject (blocktype 1) or object (blocktype 2) 4 bytes: numeric, datetime, short string or string number in the string table.
- Property 2 bytes: number of the property name string in the name string table.
- Object (blocktype 1) or subject (blocktype 2) 12 bytes: epc, numeric, datetime, short string or string number in the string table.
- Reserved 2 bytes.
- Object type 2 bytes: number in the string table indicating type of value (int, short string, some structure etc).
- Checksum: 1 byte.

We use xml schema datatype names as value type indicator strings, extended by our own specific datatype names.

We use the simplest checksum algorithm: adding bytes 0...31 one after another and keeping the lowest byte of the sum after each addition.

Multibyte integers and floats have to follow the high-endian (intel standard) byte order. Direct short strings start from the leftmost byte and should be terminated with a zero byte. In case there is no zero byte, the data reader has to append the zero byte to the direct string (4 or 12 bytes) read.

In normal cases it is recommended to use the first type of data blocks with a long value field. The second type is suited for cases where we want to write information about a specific RFID chip, different from the current chip (in the latter case we should use the special ’me’ string http://www.roboswarm.eu/lang#me).

4.5 Reading and writing data

In case a robot writes data to an RFID tag, it will normally have to delete some old data to make room for new data to be written. It will also have to take care that important data is not deleted. The robot follows these principles:

It will always delete the oldest data block which is allowed to be deleted. By default all data blocks written by humans have to be preserved. The internal datastore of a robot contains information about kinds of writers (block contains the writer id).

5 Antennas and other practical aspects of RFID reading and writing

Before writing or reading data, the robot will have to understand that an RFID tag is in a reading or writing distance. It will then start reading and - sometimes - also writing the RFID.

We have conducted a number of RFID reading and writing experiments with an iRobot Create equipped with a Gumstix Verdex microcomputer and the Skyetek M9 OEM RFID reader card, operating frequency was 865MHz, output power 27dBm.

Achieved dependable access ranges for ISO 18000-6B and 6C tags have been between 0.7 and 1.2 meters, depending on tag orientation and various other factors.

The practical issue of detecting a tag depends on many factors, quite significantly also on the shape of the tag's antenna and the orientation of tag in the robot's RF field. Therefore one antenna should be omnidirectional (e.g. circular polarization antenna).

However, two switched linearly polarized reader antennas may be used giving additional direction information. That idea will be evaluated further.

On the figure 2 we have an iRobot Create equipped with the 6 dBi Yagi antenna. This antenna appeared to be too sensitive directionally: it was hard to notice tags not directly in front of the robot.

The figure 3 demonstrates a 13dBi spiral antenna designed during the project. While detecting tags from a somewhat longer distance than the Yagi antenna, it had analogous problems with directionality.

The best choice so far has been the patch antenna on the figure 4. The small loss in tag detection distance is compensated by the significantly wider area of coverage, enabling the robot to detect tags not directly in front of it.

In our experiments it has been somewhat easier to detect the RFID and read its id number than to read full RFID memory. Hence, when a tag is detected somehwere in front of the robot, we keep driving for a short while to get to the practical reading/writing distance.

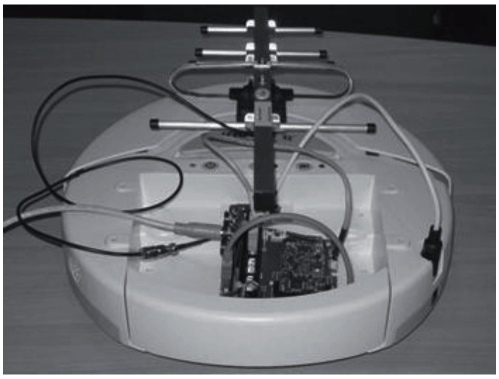

Figure 2. iRobot Create with a 6 dBi Yagi antenna

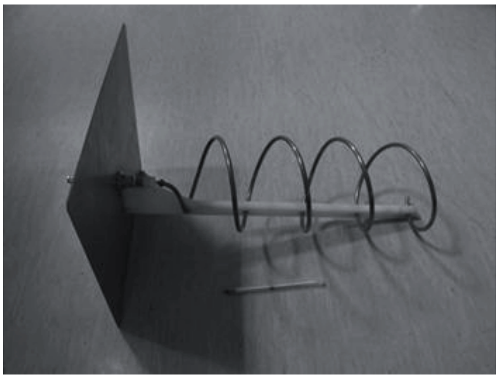

Figure 3. 13dBi spiral antenna

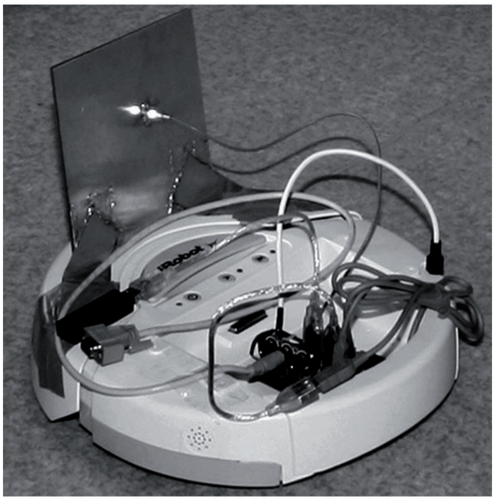

Figure 4. iRobot Create with a 6.5dBi patch antenna

Another important aspect is the frequency of scanning for the RFID tags: since tag reading and scanning draws significant amount of power, high frequency of scanning drains the Roomba internal battery faster than would be practically feasible.

6 Rule engine and the rule language

The central command agent uses the RDF database contents as grounds for deciding whether the robot is doing ok, is in trouble or what to do next.

Programming the robot to act correctly for each case is hard. We are using a rule engine to perform specific checks on data and make decisions based on the given set of rules. Rules are written in a prolog-like syntax and stored initially as a plain-text rule file in the robots file system. The rule engine takes all the input data from RDF database and stores derived facts again into the RDF database. Other agents do not use the rule engine directly, they just read the output from the in-memory database.

In other words, the rule engine is not used for answering queries, but for automatically deriving new facts added to the RDF database. Obviously, the set of rules has to be consistent and should not contain too many or too complex rules. We are using the special modification of the Gandalf first order resolution-based theorem prover [8] as a rule engine.

The rule engine is fired automatically by the rule engine process after each pre-determined interval. Using a relatively simple set of rules we manage to keep the interval under one second: during this time the rule engine performs all possible derivations stemming from the facts added to the RDF database after the last iteration.

The rule system is used for two main kinds of tasks:

- Deriving generalisations (chair is furniture) from ontology rules.
- Deriving commands and subcommands, depending on the situation.

We are not using OWL directly as an ontology language. Instead, the central server contains a component for converting given OWL files to the rule language syntax. These rule files are then preloaded to robots and updated over WIFI, if available.

The # mark in the following examples stands for the full default namespace http://www.roboswarm.eu/lang. Although we use the syntax based on Prolog, the derivation algorithm is a specialized version of the bottom-up resolution algorithm as often used in automated theorem provers, starting from the facts and deriving new facts/lemmas. The derivation process does not attempt to solve a posed ”query”, just to derive new facts. Hence the language does not contain extralogical predicates like cut and closed-world not.

Two simple ontology rules, indicating that anything attached to a glass object is attached to a fragile object, and anything attached to a fragile object is attached to a dangerous object:

```
#attachedTo(X,fragile) :-
    #attachedTo(X,glass).
#attachedTo(X,dangerous) :-
    #attachedTo(X,fragile).
```

Sample rules for firing executable commands with argument 0 and high priority 1 (”me” is a special macro constant indicating robot itself):

```
command(escape,0,1) :-
   #attachedTo(X,dangerous).
command(clean,0,1) :-
   "found-tag"(me,tag2).
```

The next rule derives information about a need to keep away for 10 minutes from the given location. ”now” is a special macro constant indicating current time. There should be further rules given to make the robot actually use this information:

```
#keepaway(Loc,600) :-
    #roombusy(Loc,Time),
    lesstime(now,Time).
```

7 Robot sensor/task language

The sensor/task language does not have a separate syntax. Rather, it is a collection of strings with predetermined meanings, designed for two goals: storing robot sensor data and giving commands to the robot (clean here, drive away, find a certain item).

The sensor/task language uses the RDF database for storing both tasks (commands) and sensor information. In other words, all the commands to the robot and sensor information items are stored in the RDF database as ordinary data objects with a special meaning to the robot control process.

The sensor/task language contains several different categories of object strings:

- Task data items: used for giving general kinds of commands to the robot (clean here, drive away, look for object, exit room etc).
- Sensor/status values: information added by sensor processes or derived using rules.
- Generally useful special values (me, now etc).

A typical task data item contains the following fields:

- Subject: a string indicating actual command, like ”escape” or ”clean”.
- Property: special predicate ”command”.
- Object: used for tasks or commands requiring extra information (like how far to drive). In case the command requires several information fields, these are encoded into a single value.
- Context: used to indicate both the succession of commands and nesting of commands.

Say we have two main commands c_1 and c_2 which should be performed in succession. Command c_1 has two subcommands s_1 and s_2. The command c_1 will be automatically replaced by s_1 and s_2 by the corresponding rule. These three commands will then have the following context sequences:

- s_1: [1,1]
- s_2: [1,2]
- c_2: [2]

Tasks are represented as data items the RDF database. The robot control process starts fullfilling the task as soon as it is seen in the database. The same process should mark this task as being currently fullfilled. In case of conflict or impossibility the robot control process should choose the action itself.

There can be complex tasks that consist of number of smaller subtasks. These kinds of tasks are presented using rules. Suppose somebody adds a task into the in-memory database. After a while the rule engine will take this task, find the matching rules and add derived subtasks into the database. The derived subtasks could again match some rules, in which case they will also be derived and added to database.

The sequence of tasks is encoded in the context field of data item. Task and subtasks should be seen as an ordered forest of trees with branches corresponding to subtasks.

Tasks are loosely grouped into four levels starting from high-level down to low-level tasks. A high-level task is an abstract representation of what should the robot do: for example, clean a room for whole day. A typical low-level task would be turning the robot 50 degrees.

Long-term activity - normally defined by human.

While fullfilling this type of task, the robot can also fill subtasks like recharging, exiting room etc. These tasks do not restrict robot from doing subtasks.

For example: property ”shalldocleaning”, object time in seconds until which activity holds. Robot should be in the general cleaning mode: driving around and cleaning.

Short-term activity - normally given by rules or control process, but can also be defined by human.

Fullfilling this kind of task may consist of several small tasks like turning and moving some distances.

The associated rules are expected to generate atomic commands, which are put into in-memory database waiting to be fullfilled one after another. Only one task is fullfilled at time. Here the time information is not relevant, rather, the succession should be followed.

Atomic activities - basically procedural, normally generated by rules or the control process.

For example:

- `command(turn,Degrees,Context)`: robot should turn the indicated amount of degrees. ”turn” here is a constant string indicating the actual pre-programmed procedure the robot should follow. Context should contain a task order/priority indicator as explained before.
- `command(move,Centimeters,Context)`: robot should move (with ’normal’ speed) the given amount of centimeters.

Direct activities - these kinds of tasks can be directly delegated to the robot API for execution. The Roomba robot has very few such direct commands available: the most imporant command is ”drive with speed X and radius N”.

8 Related work

The SHAGE/AlchemistJ framework [5] can be mentioned as one solution for robot knowledge exchange using data repositories and component brokers.

Using high level data representation is a trend of modern robotics. For example, XML based data coding has been used on robots [5]. However, besides the benefits of universal high level representation the XML encoding requires additional conversions between exchange and machine control domains.

Conventional XML based RDF format is used for time uncritical inter-robot or server communication, the description can be found in [2]. A special, compact RDF format is used for storing real-time algorithms of robot operation.

The RFID technologies with goals similar to our experiments have been investigated in [9]. The authors use RFIDs to allow an autonomous mobile robot to acquire a target and approach it for task execution. The robot is equipped with a dual directional antenna that communicates with controllable RF transponders.

See also [3], [6], [10].

9 Conclusions and future work

We have designed the architecture for the robot swarm and started actual implementation and testing with real tags and robots. So far we have successfully implemented both the robot hardware and software, including the RDF database, RFID and rule engine usage as described in the paper. The experiments have been encouraging when we consider processing power and reaction times: the tiny onboard Gumstix computer manages to run the described agents, use the RFID chips, RDF database and the rule engine without slowing down the robot reactions. On the other hand, detecting, reading and writing RFID tags requires considerable care when selecting tag types, antennas and the scanning frequency. A usable solution has been worked out, but further optimisations and improvements are needed.

We have also started to implement components of the central server and open connections to other robot swarms and external software. However, this work is still ongoing, specific details are being filled in and it is too early to report experiments from the high-level perspective.

Acknowledgements. The work was supported by FP6 ICT ”ROBOSWARM” project, see http://www.roboswarm.eu.

References

[1] Y. Altshuler Y, A.M. Bruckstein, I.A. Wagner: Swarm Robotics for a Dynamic Cleaning Problem. In ”IEEE Swarm Intelligence Symposium”, pages 209–216, 2005.

[2] E. Ardizzone, A. Chella, I. Macaluco, D. Peri: A Lightweight software architecture for robot navigation and visual logging through environmental landmarks recognition, in Proc of International Conference on Parallel Processing Workshops, ICPPW 2006.

[3] A. Elci, B. Rahnama: Human-Robot Interactive Communication Using Semantic Web Tech. in Design and Implementation of Collaboratively Working Robots, RO-MAN 2007. The 16th IEEE International Symposium, pages 273–278 (2007).

[4] B. Hayes-Roth: A blackboard architecture for control. Artificial Intelligence, 26(3): pages 251–321, July 1985.

[5] S. Lee, I.H. Suh and M.S. Kim (Eds): Recent Progress in Robotics, LNCIS 370, Springer, pages 385–397, 2008.

[6] C. Stanton, M.-A. Williams: Grounding Robot Sensory and Symbolic Information Using the Semantic Web in RoboCup 2003: Robot Soccer World Cup VII, Springer LNCS 3020/2004, pages 757-764, 2004.

[7] T. Tammet, J. Vain, A. Kuusik: ”Using RFID tags for robot swarm cooperation”. WSEAS Transactions on Systems, 5(5), pages 1121–1128, 2006.

[8] T. Tammet: Gandalf. Journal of Automated Reasoning vol 18 No 2, pages 199–204, 1997.

[9] V. A. Ziparo, A. Kleiner, B. Nebel, D. Nardi: RFID-Based Exploration for Large Robot Teams. In Proc. IEEE International Conference on Robotics and Automation, pages 4606–4613, 2007.

[10] L. Vasiliu, B. Sakpota, K. Hong-Gee: A semantic Web services driven application on humanoid robots. in the Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop, 2006.

PAPER 2

T. Tammet, E. Reilent, M.Puju, A. Puusepp, A. Kuusik, A. Knowledge centric architecture for a robot swarm. In: 7th IFAC Symposium on Intelligent Autonomous Vehicles (2010). IFAC-PapersOnLine, 2010, (Intelligent Autonomous Vehicles; 7/1). 2010.

Knowledge Centric Architecture for a Robot Swarm

Tanel Tammet, Enar Reilent, Madis Puju, Andres Puusepp [*]
Alar Kuusik [**]

[*] *Department of Computer Science, Tallinn University of Technology Ehitajate tee 5, 19086 Tallinn, Estonia (e-mail: tammet@staff.ttu.ee, e.reilent@gmail.com, pudismaju@gmail.com, anduoma@hot.ee).*
[**] *Department of Electronics, Tallinn University of Technology Ehitajate tee 5, 19086 Tallinn, Estonia (e-mail: kalar@va.ttu.ee).*

Abstract: We have built and tested a knowledge centric system for a robot swarm. Our implementation enhances iRobot Roomba cleaning robots with a tiny linux computer, RFID tag reader/writer and optionally a WIFI card. Robots use the RFID tags for object recognition and message passing. The knowledge architecture of the system is inspired by semantic web principles, spanning over several layers: RFID tags on objects, process interaction in a single robot via a main memory datastore and a rule system, central database for a swarm. The communication components of the system have been already ported to the larger Pioneer and Mugiro robots via the Player middleware. The paper presents our solutions to the knowledge management and communication problems stemming from the robotics issues and demonstrates feasibility of using the semantic web principles in the robotics domain.

1. INTRODUCTION

The overall goal of the project is to develop simple and low-cost technologies for making both single robots and swarms of robots more intelligent. We use the dynamic cleaning problem Altshuler et al. (2005), Tammet et al. (2006) as a testbed for the developed knowledge architecture, focusing on making swarm cleaning more efficient.

Our goal requires propagation of understandable and reusable information among the robots which may be different in hardware and software. The target goal can be called a "knowledge centric" architecture, focusing on uniform (or easily convertible) on-robot and inter-robot data management Tammet et al. (2008), which is achieved by using prolog-like rules and first-order logic.

We use ordinary passive RFID chips for marking objects like chairs, walls, doors. This is significantly cheaper and more flexible than using cameras on robots for object recognition. The same RFID chips on objects are also used by the robots to leave messages to other robots. The solution is inspired by ants' communication using pheromone trace known as stigmery. The usage of RFID tags reduces the communication overhead related with coordination Ziparo et al. (2007).

We use the popular iRobot Roomba cleaning robot and attach a tiny ARM-based Gumstix computer (500 MIPS) using a BusyBox 2.6 Linux distribution (without real-time capabilities) and a stock RFID reader/writer on the Roomba. The attached computer takes over control of the Roomba. While the standard Roomba is fairly simple-minded, will clean places recently cleaned and does not understand that some places should be avoided - or vice versa, cleaned often - our system adds necessary intelligence.

The main communication components of the architecture have been already ported by the industrial project partner Fatronik to two different robots (Pioneer and Mugiro) via the Player middleware.

2. ROBOT KNOWLEDGE ARCHITECTURE

The architecture for the robot control is based on a layered multi-agent system, with agents implemented as continuosly running processes. Three layers can be brought out:

- The sensor-actuator access layer dedicated to communication with the robot control hardware. The lowest part of robots sensor-actuator layer is executed by the Roomba onboard microcontroller.
- The control layer consists of dispatcher process which executes behavioral tasks in our context called binaries.
- The knowledge layer that targets reasoning (deriving new information from acquired data), communicating with other robots (using RFID tags) and the optional central server (using WIFI, if available).

The layered architecture is built around a fast and transparent RDF inspired datastore implemented in shared memory. This kind of approach is frequently used for low latency robot control architectures performing sufficiently well without using a real time OS. The internal knowledge architecture follows the classical blackboard model Hayes-Roth (1985). In short, the agents communicate by writing data to the memory datastore and every agent can access all data inserted to datastore.
The memory datastore serves three roles:

- A postbox between different process agents (including external world communication).

- A fast and simple in-memory data store (circular buffer).
- A deductive database, using a rule language for rule-based generation of new facts.

3. COMMON DATA MODEL AND LANGUAGES

The behavior of the robot is primarily influenced by four players:

- sensors and control software
- internal memory datastore contents
- RFID tags read
- binary executables plus data and rule files read from the swarm server.

The swarm server collects data from the robots and influences them by sending new data back to the robot datastore, updating rule files in the robots and sending new binary executables to the robots.

The different players above use specialised language representations, all based on RDF triples plus metadata: the combination which we will call RDFm. Different syntaxes stem from practical needs. For example, since RFID chips contain very little memory, we have to use a space-efficient encoding for information on RFID-s. On the other hand, communication between different servers does not require space efficiency: rather, it is preferrable to use common, verbose XML-based standards.

We use the following languages in the robot swarm system:

- RDFm encoding in RFID tags.
- Our specialised rule language for deriving new information based on data in memory datastore.
- Both a CSV-based syntax and an XML-based RDF syntax for data exchange between robots and the central server (using WIFI if available) and the central server and external systems.

All these languages share a common data model and the concrete predefined strings for adressing data to agents.

3.1 Common data model

The common data model is inspired by RDF triples to which we add two additional groups of data fields (metadata): contextual data fields and automatically generated metadata.

Data fields taken from RDF triple:

- Subject: id of whatever has the property.
- Property: name of the property of the subject.
- Value: value of the property.

The value field has an associated type, indicating the proper way of understanding the value. Observe that the property field typically - but not always - already determines the suitable or expected type.

In addition to basic RDF, we will always add three contextual metadata fields to the beforementioned proper data fields of the triplet.

Contextual metadata fields:

- Date/time: when this fact held (in most cases same as the time of storing the data).
- Source: identifies the origin of the data (RFID nr, person id, other robot id, agents, etc).
- Context: usually identifies addressee or data group, can also indicate the succession of robot commands.

Agents can enter their own contextual values to the memory datastore. If no values are given by the agent, the default values (current date/time, robot id, empty context) are entered automatically.

Automatically generated metadata:

- Id: unique data row nr for a robot, auto-increased.
- Timestamp: date/time of storage.

Automatically generated metadata is present only in the memory datastore, and not in the other data languages. Agents cannot enter their own values at will. These two fields are important for efficient and convenient management of the data and are used for example, by the reasoner and dispatcher processes.

Instead of using additional contextual and metadata fields we could have chosen to use reification of RDF triples to store the same information. However, this would have been cumbersome and inefficient both for the internal memory datastore used by the agents inside the robot, and even more so for the data representation inside RFID chips, as described in the following chapters.

4. MEMORY DATASTORE

The memory datastore is implemented in the Gumstix computer on the robot as a library for storing and reading information to/from shared memory. Agents in the robot use only a simple C API for writing, reading and searching data from the memory datastore. Agents see datastore as one table based on RDFm format.

Strings in the memory datastore are pointed to from the data fields: they are kept in a separate table, guaranteeing uniqueness: there is always only one copy of each string.

The data rows are organised as a circular list. The last data element will disappear when a new one is added. However, there are exceptions to this order: data items deemed critical are kept longer.

Locking is implemented using semaphores and is row-based. Reading operations do not lock anything. When a row is being written it is invisible for all the concurrent reads.

Writing one row on Gumstix platform takes about 0.14 ms while looping over 2000 rows takes approximately 4.8 ms, which is acceptable for our needs.

Although the data store should be normally seen as a mid-term memory, containing thousands of rows (old data is thrown away), it is easy to use it as a postbox between different agents onboard: just put the name of the addressee agent in the context field and program the addressee agent to look for the rows with its name, process them and then delete them.

5. DATA ENCODING ON THE RFID TAGS

The roboswarm architecture requires recognising external objects/locations, reading location-specific messages/instructions from humans and reading/writing location specific messages from/for other robots. All these three tasks use RFID tags at different locations. The simplest types of RFID tags contain only the RFID id. However, we have been using RFID tags with a small internal memory: both human operators and robots can write information to the tags. We use the tags as information-carrying graffiti, in other words, tiny data stores distributed all over the environment.

A human user is expected to write to a tag information like "this tag is located on a chair", "this tag has coordinates X and Y", "there is a tag at direction R at distance 5 meters", "keep away from here" etc.

A robot N is expected to write to a tag information like "N brushed here for 10 minutes on 10.06.2007 at 15.10", "N left this place for the living room" etc.

Data encoded on tags must be easily understood both by the robot software and the external applications: software used by humans to read and write data to tags as well as agents outside the roboswarm.

All data items written to the RFID tags are essentially data rows with several predefined fields which may contain strings, integers or floats. The RFID data store is very similar to the robot's memory datastore. Data is read, written and deleted one full row at a time, updating is allowed only on the value field, the timestamp field and the source field.

For example, one tag might carry the following data:

subject	**property**	**value**	**source**	**context**
me	inRoom	kitchen	human	static
kitchen	hasPriority	7	human	static
kitchen	dutyStatus	cIP	robot3	work
robot2	wentInDirection	270	robot2	work

* cIP - cleaningInProgress

where the "static" context is used for data describing the surrounding enviroment and the "work" messages are written to the tag by swarm members to improve cooperation while performing tasks.

Data field contents are either direct (integers, RFID chip ids) or indirect (long strings). Direct values are put on the chip as-is. Long strings are not kept on the RFID, since we do not have enough space: we use a string number in a global string table instead. This global string table is loaded into the robot and has to be the same for the whole swarm. It is necessary only for coding and decoding data for the RFID tags. We use 2 bytes for the string table numbers.

5.1 Reading and writing RFID tags

RFID tags carry a built-in id. The id size may vary. However, there are several widely used standards for product encoding, and most RFID tags are expected to conform to these standards.

We use direct 96-bit EPC-s to identify RFID tags. It is very common for a tag to contain information about its own location or the object it is glued to. While referring to itself we use the string "me" in the data row instead of the real EPC value.

In case a robot writes data to an RFID tag, it will normally have to delete some old data to make room for new data to be written. It will also have to take care that important data is not deleted. The robot follows these principles: It will always delete the oldest data block which is allowed to be deleted. By default all data blocks written by humans and the blocks with the context "static" have to be preserved.

6. KNOWLEDGE BASED CONTROL SYSTEM

The robot control and decision making responsibilities in our system are divided between several different agents.

The control system architecture has two layers: the supporting framework and the user applications built upon the framework.

The crucial element in our system is the memory datastore. All the other subsystems are meant to be built around the datastore and interact with each other only via the datastore. As a consequence, all data - sensor readings, decisions, commands, reports, etc - ever created by some agent will be available to all the agents.

Gathering all kinds of knowledge into one place and representing it in the same format encourages us to attach a general data processing mechanism - the prover - to the memory datastore. The prover is used to derive new data items based on the existing data in the memory datastore and predefined logic rules.

The supporting components like the prover, the communication process and the low level hardware access software (sensor process, actuator process) run all the time as separate never-ending processes. However, the control-specific modules are not required to run all the time. Therefore, in addition to the prover the control support framework uses a special dispatcher process with the task to launch other agent processes during runtime.

The implementation of "the real" control software is very flexible. The algorithm can be divided to various modules and rules. For several subtasks we have created dedicated modules (binary executables) which are relatively small and simple. A binary executable can perform an atomic task, for example play a sound or calculate an average, or comprise a set of actions to achieve a complex goal, like performing a localization procedure at the reference point (RFID tag).

Rules have the role of linking binaries together and making decisions during runtime. For example: the agent A stores the fact B into the datastore. The prover derives (according to the given rule files) the new fact C from the fact B, where C is a command to start the agent D. When the dispatcher sees the derived fact C in the datastore, it launches the demanded agent D, which in turn can change the contents of the datastore.

7. RULE ENGINE AND THE RULE LANGUAGE

The control system uses the memory datastore contents as grounds for deciding whether the robot is doing ok, is in trouble or what to do next.

Programming the robot to act correctly for each case is hard. We are using a rule engine to perform specific checks on data and make decisions based on the given set of rules. Rules are written in a prolog-like syntax and stored initially as a plain-text rule file in the file system of the robot. The rule engine takes all the input data from memory datastore and inserts derived facts into the memory datastore. Other agents do not use the rule engine directly, they just read the output from the datastore.

In other words, the rule engine is not used for answering queries, but for automatically deriving new facts added to the memory datastore. Obviously, the set of rules has to be consistent and should not contain too many or too complex rules. We are using the special modification of the Gandalf first order resolution-based theorem prover Tammet (1997) as a rule engine.

The rule engine is fired automatically by the rule engine process after each pre-determined interval. In the following we will call this "firing" process the *derivation session.* Using a relatively simple set of rules we manage to keep the derivation session interval under one second: during this time the rule engine performs all possible derivations stemming from the facts added to the memory datastore after the last iteration.

The rule system is used for two main kinds of tasks:

- Deriving generalisations (chair is furniture) from rules.
- Deriving commands depending on the situation.

While the rule system is working, it uses memory datastore as an additional source of facts.

For example, if we have a rule

```
attachedTo(X, furniture) :-
  attachedTo(X, chair).
```

and the following facts in the memory datastore

subject	property	value	source	context
tag4	attachedTo	chair	RFID	null

then the rule body $attachedTo(X, chair)$ will match the datastore row and the rule will generate the new fact and add it to the memory datastore as follows:

subject	property	value	source	context
tag4	attachedTo	furniture	wGandalf	null

All the words in the rules starting with uppercase are variables. In our example X is a variable.

The following example demonstrates a simple session of robot rule usage.

```
handleTask(me, Task) :-
  state(me, stateIdle),
  receivedTask(N, Task),
  myNameIs(me, N).

state(me, stateWorking) :-
  handleTask(me, T).

startMode(me, cleaningMode) :-
  handleTask(me, clean).

startMode(me, patrollingMode) :-
  handleTask(me, patrol).

state(me, stateIdle) :-
  state(me, stateWorking),
  status(currentTask,finished).
```

We start the rule system and then add the following fact to the datastore:

subject	property	value	source	context
me	state	stateIdle	init	wGandalf
me	myNameIs	robot3	init	wGandalf
robot3	receivedTask	clean	init	wGandalf

The rule system will automatically derive and add these facts to the datastore:

subject	property	value	source	context
me	handleTask	clean	wGandalf	null
me	startMode	cleaningMode	wGandalf	null

When we later add the fact

subject	property	value	source	context
currentTask	status	finished	cleaningAgent	wGandalf

the rule system will automatically derive and add this fact to the datastore:

subject	property	value	source	context
me	state	stateIdle	wGandalf	null

The rule engine uses both the main memory database and a temporary storage area which is cleaned up after each derivation session, typically after every second.

During the derivation process a large set of new facts and clauses (temporary rules) is derived. Most of them are stored in the temporary area and are not accessible to other processes in the robot. Only positive singleton facts without variables (ground unit clauses), not containing nested terms and having a suitable number of arguments are stored in the shared database available to all the processes.

Each rule engine derivation session starts with reading and parsing the rule file and adding all the read rules and facts into the temporary space. Hence the rule file can be changed on the fly.

We employ the widely used discrimination tree index for unit subsumption and unit deletion. Only the temporary area, not the facts in the shared memory database are kept in the index.

The engine uses a version of a set-of-support binary resolution with common optimisations like subsumption and tautology elimination. See Robinson and Voronkov (2001) for the common algorithms employed in first-order automated reasoners.

We have to avoid re-derivation of facts which were already derived during the last session. We cannot rely solely on the subsumption algorithm for this. For example, the robot should not get the derived command facts again each time the derivation session finishes.

Hence we developed a timestamp-oriented special version of the set of support algorithm. The initial facts in the derivation are only those which have been added (or modified) in the database after the previous derivation session. This is possible, since all the facts in the database have the automatically stored timestamp field.

We cannot use, for example, hyperresolution, since this derivation algorithm is not complete in combination with set of support. Hence the use of binary resolution.

The new facts and (partial) rules derived using a binary resolution step can then be used for deriving new facts and rules, guaranteeing that in each derivation chain at least one of the facts has been added/modified after the previous derivation session.

Using the timestamp-oriented set of support algorithm is also crucial for efficiency. The number of new facts added in one second is normally not very big, and most of them typically do not match any or most rules. This keeps the amount of derived facts during one derivation session down even for relatively large rulesets.

7.1 Behaviors

Behaviors are collections of operations that the robot will perform and which are called with one command. Implementation of a behavior is a little binary executable written in the C language. It contains a sequence of commands and conditions to perform a relatively complex operation by the robot.

For example, let us consider the following ruleset:

```
behavior(me, "monitorObstacles"):-
state(me, stateInitial).

behavior(me, "goAhead 200"):-
state(me, stateCanmove),
obstacle(me, nothing).

behavior(me, "handleFailState"):-
result(solveObstacle, fail),
state(me, stateDriveAround).
```

- behavior - a special name, indicates that the fact is the command to launch the given binary.
- monitorObstacles - a binary monitoring whether any obstacles are getting in the robots way. If there is an obstacle in front of the robot, the $obstacle(me, front)$ row will be added to the datastore.
- goAhead - a binary that makes the robot to start moving forward with the given speed (in the current case with the translational velocity 200 mm/s and angular velocity 0)
- handleFailState - a binary that stops the motors and sensor equipement to save power, then tries to communicate the information about the failure situation to other robots or the central server. Used when the robot is stuck and unable to move or trapped in the place where it cannot find the way out.
- solveObstacle - a binary that tries to drive the robot away or around the obstacle which has gotten in the way.

All the behaviours are handled by the process we call dispatcher. The dispatcher executes binaries: small executable programs implementing the behaviours. In order to make the dispatcher to execute one binary, it must be copied to a predefined folder on the robot and at the desired time the proper command must be inserted to the memory datastore.

An example of a command row which forces the dispatcher to execute a binary:

subject	**property**	**value**	**source**	**context**
me	behavior	command	wGandalf	dispatcher

* command - "behaviorName arg1 .. argN"

While implementing the robot control application on top of the prover and a relatively large set of behaviours, timing becomes a critical issue. The elapsed time between a stimulus and its reaction varies greatly depending on the current contents of the memory datastore, the length and the complexity of the rule file, the number of processes running in the system, the length of the reaction chain and other factors. However, in our case study the response times have proved to be acceptable.

For example, let us consider a cooperation between the prover, the dispatcher and two behaviours to avoid the robot colliding with an obstacle. Typically it takes about 400 ms from the moment when one behaviour (monitorObstacles) discovers an obstacle to the moment where the prover inserts a command into the memory datastore to launch another behaviour. After about 20 ms the dispatcher has received the command and is ready to start the given behaviour. After additional 100 ms the second behaviour (solveObstacle) takes over the control of the robots movement.

High-level decision making can safely rely on the given architectural scheme. However, critical emergency responses like avoiding the robot falling down the stairs after the cliff sensor detects descent should be implemented in hardware or low-level software agents.

8. ROBOT DATA STORAGE ON THE SERVER

Robots using WIFI can use the robot-server centralised communication and robot-robot ad hoc communication in case no WIFI access-points are available. A separate process on the robot sends new data items from the memory datastore to the server, currently at an one-second interval. On the server side the data of the whole swarm is stored in a postgresql database for further processing.

The server replies each uploading act with the new data items intended for this particular robot, accumulated since the last communication session. Software agents on the server cannot directly send any data or commands to robots, in lieu of that they will write the data into the same postgresql database. The special communication agent then passes it to the selected robot as soon as the robot contacts the server. The selected robot adds the data items received from the server to its own memory datastore.

Human users can control and monitor swarm or single robots via dedicated user interfaces built on the server database. It is technically possible to assign a direct task to the robot, even though the data flow normally passes several intermediate agents on the server. Data produced by the user interface is sent to the task decomposition module which specifies and assigns proper subtasks for the robots.

The server has an additional swarm coordination role in some applications. For example, if we consider the task where a group of robots must search for an RFID-tagged object, it is reasonable to use the server. After the user has given the task, the server distributes the task information down to the suitable set of robots which then spread out in the environment and start performing the search. As soon as one of the robots has found the demanded object, it will communicate the knowledge to the server which then informs the user and notifies the other robots to stop searching.

8.1 Communication protocols

The CSV protocol version sends data over http POST. The first row in the data block contains a sender id, the following rows contain the memory datastore rows in the standard CSV format. The protocol is used to both send data from the robot datastore to the swarm postgresql database on the server, and vice versa: from the swarm database to the single robot datastore.

The CSV protocol data format:

```
robots
subject,propery,...,usecstamp
...
subject,property,...,usecstamp
```

with the rows containing the same fields as the memory datastore: subject, property, value, valuetype, source, context and two timestamps both comprising seconds and microseconds.

Some parts of the row may be omitted if the original record in the memory datastore lacks these particular elements, eg context. In this case we will simply have commas right after another (,,,) as commonly used in the csv format.

In case the memory datastore contains an integer or float in the value field, it will be presented as a human-readable string according to the obvious xml schema principles. The special subject value ”me” in the memory datastore will be replaced with the id of robot who is sending the data, otherwise value is left as it is.

9. RELATED WORK

The SHAGE/AlchemistJ framework Lee et al. (2008) should be mentioned as a solution for robot knowledge exchange using data repositories and component brokers.

High level data representation is a trend of modern robotics. For example, XML based data coding has been used on robots Lee et al. (2008). However, besides the benefits of universal high level representation the XML encoding requires additional conversions between exchange and machine control domains.

Conventional XML based RDF format is used for time uncritical inter-robot or server communication, the description can be found in Ardizzone et al. (2006). A special, compact RDF format is used for storing real-time algorithms of robot operation.

10. CONCLUSIONS AND FUTURE WORK

We have designed the robot swarm system and are conducting actual testing with real tags and robots. So far we have successfully implemented both the robot hardware and software, including the memory datastore, RFID and rule engine usage, communications with the server and the server database as described in the paper. The experiments have been encouraging, especially when we consider the weak processing power and high reaction times: the tiny onboard Gumstix computer manages to run the described agents, use the RFID chips, memory datastore and the rule engine without slowing down the robot reactions. Porting the system to the larger Pioneer and Mugiro robots via the Player interface (conducted by Fatronik) was relatively easy. As a consequence, we now have three very different robots able to communicate through the same infrastructure.

On the other hand, detecting, reading and writing RFID tags requires considerable care when selecting tag types, antennas and the scanning frequency. A usable solution has been worked out, but further optimisations and improvements are needed.

REFERENCES

Altshuler, Y., Bruckstein, A., and Wagner, I. (2005). Swarm robotics for a dynamic cleaning problem. *IEEE Swarm Intelligence Symposium 2005 (SIS05)*, 209–216.

Ardizzone, E., Chella, A., Macaluco, I., and Peri, D. (2006). A lightweight software architecture for robot navigation and visual logging through environmental landmarks recognition. In *International Conference on Parallel Processing Workshops*.

Hayes-Roth, B. (1985). A blackboard architecture for control. *Artificial Intelligence*, 26(3), 251–321.

Lee, S., Suh, I., and (Eds), M.K. (2008). Recent progress in robotics. In *Lecture Notes in Control and Information Sciences 370*, 385–397. Springer.

Robinson, J.A. and Voronkov, A. (eds.) (2001). *Handbook of Automated Reasoning*. MIT press.

Tammet, T. (1997). Gandalf. *Automated Reasoning*, 18(2), 199–204.

Tammet, T., Vain, J., and Kuusik, A. (2006). Using rfid tags for robot swarm cooperation. *WSEAS Transactions on Systems*, 5(5), 1121–1128.

Tammet, T., Vain, J., Kuusik, A., Puusepp, A., and Reilent, E. (2008). Rfid-based communications for a self-organising robot swarm. In *Self-Adaptive and Self-Organizing Systems*.

Ziparo, V.A., Kleiner, A., Nebel, B., and Nardi, D. (2007). Rfid-based exploration for large robot teams. In *IEEE International Conference on Robotics and Automation*, 4606–4613.

PAPER 3

E. Reilent, I. Lõõbas, R. Pahtma, A.Kuusik. Medical and Context Data Acquisition System for Patient Home Monitoring. In Electronics Conference (BEC), 2010 12th Biennial Baltic (pp. 269-272). IEEE.

Medical and Context Data Acquisition System for Patient Home Monitoring

Enar Reilent[1], Ivor Lõõbas[1], Raido Pahtma[2], Alar Kuusik[3]

[1]*ELIKO Competence Centre, Teaduspargi 6/2, 12618 Tallinn, Estonia, E-mail: firstname.lastname@eliko.ee*
[2]*Laboratory for Proactive Technologies, TUT, Ehitajate tee 5, 19086 Tallinn, Estonia, E-mail: raido.pahtma@dcc.ttu.ee*
[3]*Department of Electronics, TUT, Ehitajate tee 5, 19086 Tallinn, Estonia, E-mail: alar.kuusik@ttu.ee*

ABSTRACT: Patient remote monitoring has continuously rising importance for aging countries. Computer based assisted living systems are too difficult to handle by elderly people, therefore it has been proposed to extend smart home control or entertainment systems with patient monitoring functionalities. However, such implementations are usually platform and data protocol specific making their extension time consuming and data interpretation complicated. We propose a semantics driven messaging system and agent based software architecture for home telecare that is open for adding any kind of medical or context sensors, either wired or wireless ones. System can be reconfigured during the operation and its output data is highly compatible with eHealth databases. The solution is prototyped.

1 Introduction

Telecare (tele-home-care) provides a recognized solution to control the growth of medical expenses caused by an increasing proportion of elderly and chronically ill people [1]. As well described by Doughty et al. the modern telecare solutions should be able to monitor slow deteriorations of well-being for early discovery of health risks [2]. It is believed that discovery of slight deviations in health condition requires lifestyle-monitoring in addition to medical parameter measurements, as outlined by Barnes [3]. In case of the long term monitoring, especially while considering contextual data, a significant increase in the amount of data and its handling involution is usually not addressed on the system architecture level in a comprehensive way. From one side, the existing telecare solutions are designed as classical data acquisition systems not providing the flexibility to add weakly specified context information. From the other side – which is more important – if the context information is provided for a particular monitoring task, it is difficult to analyse and compare the data against other content sources like Electronic Health Records (EHR) and different telecare implementations. The present paper describes content centric software architecture for telecare systems that simplifies machine processing of patient and context information through the semantic representation of data and semantic reasoning.

2 Previous work

The common approach in patient lifestyle monitoring is utilization of Smart Home (SH) control platforms. Monitoring of the duration, frequency, and patterns of patient's daily activities gives important context for medical measurements and even can be used for discovery of emergency conditions [4, 5]. As presented by Chen et al., the SH environments are producing massive amounts of sensor data. However, until the data is imbued with well-defined meaning, the potential use for describing lifestyle context for patients is rather limited [6]. The use of widely accepted ontologies (controlled, relational vocabularies) allow more simple interpretation and reasoning of the information - it is already proposed to use ontologies for describing (smart) environment context [7]. In the MATCH project, Turner et al. proposes to use ontologies for data clustering [8]. However, while the works of both Chen and Turner show that semantic enrichment of context information simplifies its processing, they do not specify any practical ontologies to use. Neither do they propose how the semantic data should be handled within real telemonitoring systems, starting from the information source (semantic sensor) up to the hospital EHRs.

Essential components for modern patient telemonitoring include patient profile and automated sensor handling. From the theoretical side, the policy (i.e. rule) based home care systems, by Turner and others [9, 10], are promising for personalization and simple customization. Again, described solutions represent prototype implementations not compatible with actual EHRs or practical data acquisition systems by means of sensor integration and existing formal reasoning tools.

3 Proposed software architecture

Based on analysis of recent related work we can say that modern telecare solutions should support:

- Context monitoring to enrich the medical information;
- Personalization up to the level of individuals;

- Simple sensor integration, automated service invocation and runtime reconfiguration;

Semantic content driven data processing and rule based reasoning solutions should satisfy described requirements for modern telecare systems. Existing commercial telecare systems, even supporting HL7 v3 XML communication standard [11], have conventional client server data acquisition system architectures and are targeting single application use. The rule based and semantic approaches of telemonitoring described above represent theoretical State-of-the-Art and do not propose any practical implementation frameworks for their realizations.

We propose an open software architecture which supports semantic data processing and (soft) real-time reasoning with patient policies. The architecture is a distributed multi-agent system of independent asynchronous processes that follows the classical blackboard communication model of Hayes-Roth [12]. All agents (executable processes) within the same hardware device (e.g home embedded monitoring controller - we call Smart Home GateWay (SHGW) or a server) communicate by writing data to the central datastore and every agent can access all data inserted to the datastore.

The main advantage of data exchange via a blackboard is that, opposite to the popular socket based communication, there is no need to specify target user processes (of local real-time data) in advance which allows to implement subsumption architecture based solutions. There is always possibility to add or modify content processing agents without the need to modify agents related to sensors and other hardware.

There can be various types of agents executed in a telemonitoring SHGW device. Sensor agents acquire data from sensors devices (and systems), store it on the local blackboard and send configuration info published on the blackboard back to sensors. As a sensor agent receives raw data from a hardware device it has to convert the data into the semantic representation to be described in the following chapter. Data processing agents perform a variety of tasks including analysis of measured samples and discovery of data inconsistency. Essentially, data processing agents are supposed to make use of (semantic) formal reasoning on the data available on the blackboard. Output agents communicate with the host services e.g databases of responsible medical institution to upload aggregated data and download new configuration setting for sensing system.

4 Data presentation

All data in the system is gathered and saved into one physical datastore – the blackboard. Therefore the data format used by the blackboard must satisfy versatile needs of all agents in the SHGW system. We have to stress that existing semantic telemonitoring approaches do not address special requirements of wireless devices regarding the (data encoding) protocol efficiency by means of energy use. We expect the data format to be simple and efficient but the same time suitable for accepting readings from both medical and contextual sensors, patient's profile, messages from medical personnel. Therefore we have to achieve trade-off between efficiency and flexibility of the data format.

Considering these circumstances we propose to express all knowledge in the system by using RDF triples. The RDF-based data representation (and associated OWL-based ontology systems) for defining and describing relations and concepts is emerging in different computerized data processing applications using "the semantic web" standards and technologies. Using RDF data (knowledge) coding makes it possible to integrate existing formal reasoners and other knowledge processing and querying (SPARQL) tools.

Thus, a fact (a data item) on the blackboard in our system has the following three fields:

- Subject: id of whatever has the property.
- Property: name of the property of the subject. In RDF terminology this would be the predicate.
- Value: value of the property. In RDF terminology this would be the object.

To present all the data in the system semantically, facts should be composed by using well-defined terms that are understood by different parties in the same way. Semantical content supports later processing, exchange and unified interpretation of facts.

In our approach we stress to use keywords from ontologies published on the web. That way we can guarantee that correct interpretation of information is always possible for content user (correct use still remains user responsibility). Especially important is the proper use of concepts in the medical domain - for „heart rate" for example, around 20 different terms are in use, some of them have a specific flavour. Therefore inadequate labelling and later interpretation of sensor signals may lead to critical situations for patients. It has to be mentioned that correct interpretation of data encoded into HL7 messages, remains fully user responsibility by existing solutions. From the other side, if used ontologies are published and accessible, automated conversion of information becomes possible. Right now we are using several different ontologies, narrowing of selection will be done in the future. World-widely accepted SNOMED CT domain ontology seems to be one of the most attractive selections [13].

5 Instrumentation and data of demo system

For testing the proposed telecare solution we have implemented the software for an enhanced DVB receiver as a SHGW device. Beside cable and IP TV reception capabilities and HDMI output, the device is equipped with Bluetooth and Zigbee interfaces to support wireless medical and presence sensors. The device has 300MHz 32bit dual core MIPS type CPU and non-RT Linux OS.

The blackboard is implemented as a transparent custom shared memory datastore. There is no separate process for the datastore, agents in the system can use a set of API calls to insert and query data (triples) from shared memory. The datastore serves as mid-term memory for storing data and as a postbox between different agents. It can also be seen as a deductive database for a prover, using a rule language for rule-based generation of new facts.

For demonstrating the implementation of an sensor agent let us consider the adapter for the selected PPG (PhotoPlethysmoGraphy, measures heart rate and oxygen saturation in blood) sensor Nonin Onyx II which can communicate via Bluetooth 1.1 interface. The PPG sensor agent always runs in the background, monitoring whether the sensor is online or not. When the sensor device is turned on the agent establishes a connection and starts receiving all the output generated by the device, which it inserts to the database in the form of triples.

The example output of a single measurement sample looks as following („#3920“ is a unique key for binding the particular set of different triples into one entity):

PPG_URI, sample_URI, #3920
#3920, pulse_URI, 73
#3920, saturationO2_URI, 98
#3920, timestamp_URI, 1264423298

– where real URI strings are abbreviated. For example, „PPG_URI“ identifies one sensor device like „http://www.eliko.ee/ssg/demo/hospital1/patient3/sensors /PpgSensor“, „pulse_URI“ stands for „http://bioinfo.icapture.ubc.ca/subversion/SIRS/clinicalphenotype.owl#HeartRate“ to express sensor readings with terms of particular ontology meaning system.

Environment (part of context) sensing capabilities can be introduced to a patients home as a WSN (Wireless Sensor Network), based on for example IEEE802.15.4 radio technology. The devices are unobtrusive, battery operated and require little to no maintenance. We have interfaced our Crossbow IRIS and TinyOS based WSN to the home gateway. The nodes can be equipped with sensors for measuring temperature, humidity, light intensity, movement (PIR) etc. The nodes are location aware, initial nodes are configured with information about their placement in the environment.

A special agent runs on the SHGW to translate the data received from the WSN (in compact encoding format [14]) to RDF triples and inserts it to the blackboard. As the sensor nodes are determined to conserve their batteries for most of the time, the WSN adapter agent also has the role of managing the activity of the network. Sampling frequencies of sensor nodes in the WSN can be reconfigured by sending subscription commands to the network.

A sensor data subscription usually consists of the data type, permitted maximum age of the last reading (essentially specifying the sensing period), the area from where the reading has to be taken from and the duration of the subscription. Usually a subscription is directed to a spatial area, not a specific node, however it can be. The actual nodes that are going to provide the requested data are determined in the WSN. This means that the agent on the SHGW does not need to know the configuration of the sensor network, just spatial information about the user's environment. Additionally, the sensor subscription can contain various rules for reconfiguration, which for example specify how the sensing period should be adjusted if the reading exceeds some threshold. Subscriptions can also reference other environmental parameters which can be measured by other sensor nodes, for example, a light sensor reading is requested only if a PIR sensor detected movement.

The knowledge base in a Smart Home GateWay device could contain data about the patient's environment as given in the following set of triples where raw sensor readings have been aggregated and presented in a structured way:

Room_3_URI, type_URI, bedroom_URI
Room_3_URI, environment_URI, #1205
#1205, latest_values_URI, #2486
#1205, valid_time_URI, 15 min
#2486, light_intensity_URI, unknown
#2486, temperature_URI, 20 ^{o}C
#2486, humidity_URI, 67%
#2486, movement_URI, 30

This set of facts reads that in the patient's bedroom there are currently 20 ^{o}C of heat, humidity is 67%, movement level is 30 (in agreed units), and there is no fresh data about the light intensity in the room. Movement level is calculated from the frequency of PIR sensor(s) readings (movement detected) in the particular room. The valid time tells that none of the present values are older than 15 minutes in the current case. The system updates these values based on the received sensor data and if no data is to be found to fit into the given time limit then the latest value of the parameter is set to unknown.

6 Semantic reasoner of demo system

By our architecture we can natively apply several formal reasoners as different data processing agents in parallel for the same physical datastore. For data processing and decision making, e.g. strictly personalised emergency condition detection, simple hard-coded agents can be installed, as well as full formal reasoners like Jena, Gandalf and others for handling thousands of patient and context related facts. For example, we make use of Prolog to process semantically represented data from the PPG sensor with a blackboard agent executing a Prolog engine as a child process and feeding it with queries.

For evaluating the patient's condition on behalf of the heart rate measurements taken with a PPG sensor we also have to take into account the available contextual knowledge. In the simplified example of processing of samples additionally PIR sensors and bed sensor are considered. The bed sensor uses accelerometers to detect when the patient goes to the bed, when she leaves the bed, and how fitful or sound is the sleep.

Say, if a heart rate sample is found to be below 50 beats per minute we can predict health (or device malfunctioning) problems. Alternatively, if there is a sample above 90, derivation cannot be made. With sub-queries we try to justify the high pulse first and only if it is not possible then positive result (problem found) is returned. For example, if the patient has had significant physical movement activities 10 minutes prior the pulse measurement, we have no reason to rise alarm. Contrariwise, if the patient has not moved around or has even been in bed then heart rate over 90 indicates health problems. The Prolog rule of the main query to rise an alarm is as follows:

```
problem_suspected :-
        fact('PPG_URI', 'sample_URI', Y),
        fact(Y, 'pulse_URI', X),
        fact(Y, 'timestamp', T),
        (X<50;
        X>90,
        (not(was_active_between(T, T-(60*10)));
        was_in_bed_between(T, T-(60*10)))).
```

In experiments SWI-Prolog 5.6.58 was used to execute Prolog programs at described SHGW platform. The measured worst case reasoning time with several thousand facts and ca 15 rules was 2 seconds and average reasoning time below 200ms. The RT performance is clearly sufficient to discover patient emergency conditions quickly enough.

7 Conclusions and future work

Experiments show that proposed and prototyped semantic reasoning based telecare system satisfies real life needs of modern patient home monitoring with lifestyle context information support.

The further work will focus on integration of feasible set of medical and environmental sensors by means of developing device adapter agents supporting SNOMED CT taxonomy. The target is the real life use of the developed telecare solution including enhanced DVB receiver and wireless sensor subsystem.

8 Acknowledgements

This research has been supported by European Regional Development Fund, Competence Centre program of Enterprise of Estonia and ITEA2 EU program.

References

[1] The e-Health Innovation Professionals Group, http://www.health-informatics.org/tehip/tehipstudy.PDF (2005)

[2] K. Doughty, K. Cameron, P. Garner, Three generations of telecare of the elderly, Journal of Telemedicine and Telecare, vol. 2, no. 2, pp. 71-80, 1996.

[3] N.M. Barnes, N.H. Edwards, D.A.D. Rose, P. Garner, Lifestyle monitoring - technology for supported independence. IEE Computing and Control Engineering Journal, volume 9, number 4, August, 1998, pp. 169-174.

[4] T. Amaral, N. Hine, and J. L. Arnott, Integrating the Single Assessment Process into a lifestyle-monitoring system. In 3rd International Conference On Smart homes and health Telematic (ICOST 2005), pages 42–49

[5] A. Sixsmith, An evaluation of an intelligent home monitoring system, Journal of Telemedicine and Telecare 6 (2) (2000), pp 63-72

[6] L. Chen, C. D. Nugent, MD. Mulvenna, DD. Finlay, X. Hong X, Semantic Smart Homes: Towards Knowledge Rich Assisted Living Environment, in Intelligence on Intelligent Patient Management (Ed. by McClean S., Millard P, El-Darzi, Nugent C), Springer 2009, ISBN 978-3-642-00178-9, Pages 279-296

[7] E. Kim and J. Choi, An Ontology-Based Context Model in a Smart Home, Notes in Computer Science Publisher Springer Berlin / Heidelberg ISSN 0302-9743 (Print) 1611-3349 (Online) Volume 3983/2006

[8] K. J. Turner, L. S. Docherty, F. Wang and G A. Campbell, Managing Home Care Networks, in Robert Bestak, Laurent George, Vladimir S. Zaborovsky and Cosmin Dini (eds.), Proc. 8th Int. Conf. on Networks, IEEE Computer Society, 2009, pp. 354-359

[9] J. C. Augusto, J. Liu, P. McCullagh, H. Wang, and J.-B. Yang, Management of uncertainty and spatio-temporal aspects for monitoring and diagnosis in a Smart Home. International Journal of Computational Intelligence Systems, 1(4):361-378. Atlantis Press. 2008.

[10] K. Du, HYCARE: A hybrid context-aware reminding framework for elders with mild dementia, ICOST 2008.

[11] Integrating the Healthcare Enterprise, http://www.ihe.net (2010)

[12] B. Hayes-Roth, A blackboard architecture for control, Artificial Intelligence, 1985, 26(3):251–321

[13] International Health Terminology Standards Development Organisation, http://www.ihtsdo.org (2010)

[14] Preden, J.; Pahtma, R. Exchanging situational information in embedded networks . In: Proceedings of International Conference on Adaptive Science & Technology : IEEE International Conference on Adaptive Science & Technology ICAST 2009, Accra, GHANA , 14-16 December 2009 . IEEE Operations Center, 2009, 265 - 272.

PAPER 4

A. Kuusik, E. Reilent, I. Lõõbas, A. Luberg, A. Data Acquisition Software Architecture for Patient Monitoring Devices. Journal of Electronics and Electrical Engineering, Kaunas University, 105(9), 97 - 100. 2010.

ELECTRONICS AND ELECTRICAL ENGINEERING
ISSN 1392 – 1215 2010. No. 9(105)
ELEKTRONIKA IR ELEKTROTECHNIKA

T 115 MEDICINE TECHNOLOGY
MEDICINOS TECHNOLOGIJA

Data Acquisition Software Architecture for Patient Monitoring Devices

A. Kuusik, E. Reilent, I. Lõõbas, A. Luberg
ELIKO Competence Centre, Estonia,e-mail: alar.kuusik@eliko.ee

Introduction

Telecare (tele-home-care) provides a recognized solution to control an increase of medical expenses caused by an increasing proportion of elderly and chronically ill people [1]. As well described already by Doughty et al. [2], the modern telecare solutions should be able to monitor of slow deterioration of well-being and discover health risks early. It is believed that discovery of slight deviations in health condition requires, additionally to medical parameter measurements, *lifestyle-monitoring* as outlined by Barnes [3]. Long term patient, especially such requiring lifestyle monitoring, causes significant increase of amount of data gathered and its handling involution that is usually not addressed on system architecture level in complex way. From one side, the existing telecare solutions are designed as classical data acquisition systems not providing flexibility to add context information. From the other side – which is more important – if the context information is provided in human readable way or binary encoded, it is not machine processable outside of a single institution. Present paper describes content centric software architecture for telecare systems that simplifies machine processing of patient and context information through the semantic representation of data and semantic reasoning.

Previous work

For the patient lifestyle monitoring the use of Smart Home (SH) control platforms is a common approach. Monitoring of the duration, frequency, and patterns of daily activities, e.g. sleeping and training times, give an important context for acquired medical measurements data and can be used for discovery of emergency conditions [4, 5]. As presented by Chen et al. [6], the SH environments are producing massive amounts of data from sensors and, until the data is imbued with well-defined meaning, the potential use of the system for describing lifestyle context for patients is rather limited. Obviously, it is difficult to unify and organize the human lifestyle information using low, communication message level terminology system. The use of widely accepted ontologies (controlled, relational vocabularies) allows higher level interpretation and reasoning of information by a user. Therefore it is already proposed to use ontologies for describing (smart) environment context [7]. Within MATCH project, Turner proposes to use ontologies for data clustering [8]. However, while the works of both Chen and Turner show that semantic enrichment of context information simplifies its processing, they do not specify any practical ontologies to use and propose how the semantic data shall be handled within a real telemonitoring system starting from the information source (semantic sensor) up to the personal health record database.

Essential components for patient modern telemonitoring include patient profile and automated sensor handling. From a theoretical side, the policy (i.e. rule) based home care systems are promising for personalization and simple customization by Turner and others [8, 9]. Again, described solutions represent prototype implementations not compatible with actual health records and practical data acquisition systems by means of sensor integration and existing formal reasoning tools. Situation awareness and environmental condition detection is sometimes performed ubiquitously by several sensor motes [10] but semantic data exchange between those motes is not applicable for practical implementations.

Proposed software architecture for modern telecare monitoring systems

Having based on analysis of recent related work we can say that modern telecare solutions should support:

- Lifestyle monitoring, in addition to medical sensing;
- Personalization of patient policies;
- Simple sensor integration, automated service invocation and runtime reconfiguration;

Semantic content driven data processing and rule based reasoning solution should satisfy described requirements for modern telecare systems. However, the commercial telecare systems described above have conventional client server based data acquisition system architecture with fixed communication protocols and preknown set of supported sensing devices. Naturally, those implementations, even designed to be compatible with with HL7 v3 XML standard [11] are targeting single application use. The rule based and semantic approaches of telemonitoring described above present theoretical State-of-the-Art and do not propose any practical implementation frameworks for their realizations. Essential feature for telecare systems is the extendability by means of simple reconfiguration of hardware and introduction of new knowledge in form of rules, data processing executables, etc.

We propose an open agent based software architecture, which supports semantic data processing and (soft) real-time time reasoning with patient policies.

Multiagent system

The proposed architecture is a distributed multi-agent system of independent asynchronous processes that follows the classical blackboard communication model of Hayes-Roth [12] - the agents (executable processes) within the same hardware device (embedded controller, server) communicate by writing data to the central datastore (within the same hardware device) and every agent can access all data inserted to the datastore.

The main advantage of the blackboard (pull mode) communication based architecture is that, opposite to popular socket based communication, there is no need to specify target user processes (of local real-time data) in advance. There is always a possibility to add or modify content processing agents without the need to modify sensor and other hardware related agents.

The different Monitoring Device Hierarchy (MDH) software agents are running on intelligent sensors, Smart Home (SH) controller(s), hospital servers, PC clusters, etc. The simplest practical telecare software implementation contains telemonitoring gateway and hospital MDH levels both having their own blackboard datastore. Main agents by functionalities we propose:

- Sensor agents acquiring data from individual sensors, publish it on the blackboard and send the configuration info published on the blackboard back to sensor. Important is that there are no target user (agent) specified for the acquired content in advance. In the case the sensor agent receives raw data from hardware device it has to convert data into the semantic representation described below,
- Data processing agents performing variety of signal processing data inconsistency discovery tasks, essentially presenting data processing agents are (semantic) formal reasoners,
- Output agents communicating with host services (devices on higher MDH levels), for example an output agent running on SH central controller is responsible to export patient data to hospital system and download new configuration settings,
- HCI agents for displaying profile based selection of information on home screen or hospital web site.

Data presentation

All data in the system has to be presented semantically by using well-defined terms that are understood by different parties in the same way for supporting later processing, exchange and unified interpretation.

In our approach we stress to use the ontologies available on the web. That way we can guarantee that correct interpretation of information is always possible for content user (correct use still remains user responsibility). Especially important is the proper use of keywords in medical domain - for „heart rate“ for example, around 20 different terms are in use, some of them are equal, some have specific flavor. Inadequate labeling and later interpretation of sensor signals may lead to critical situations for patients. From the other side, if the used ontologies are published and accessible, automated conversion of information is a simple task. Good, worldwidely accepted ontology (vocabulary) for medical domain is proposed by SNOMED CT [13]. Right now we are using several different ontologies, further narrowing of selection will be done.

Different agents inside a system module (e.g., a set-top box or a aggregation server) are determined to share a common data model and data storage/exchange environment (blackboard). The schema of data representation within such system has to be universal and flexible. Therefore we propose to use RDF triples to present all the data, starting right from the sensors. The RDF-based data representation (and associated OWL-based ontology systems) for defining and describing relations and concepts is emerging for different computerized data processing applications using "the semantic web" standards and technologies. Using RDF data (knowledge) coding makes it possible to integrate *existing* formal reasoners and other knowledge processing tools. There are some XML-based semantic data presentation solutions developed for sensors like SensorML, Hydra middleware. However, those solutions are not fully RDF compatible which makes formal reasoning more complex. From the other side, XML format has a certain communication overhead, especially for wireless sensors. The additional advantage of using standard RDF allows using existing SPARQL tools.

A fact (a data item) on the blackboard in our system has the following fields:

- Subject: *id* of whatever has the *property*;
- Property: name of the property of the subject. In RDF terminology this would be the *predicate*;
- Value: value of the property. In RDF terminology this would be the *object.*

For example, PPG sensor (Bluetooth Nonin Onyx II device to gather and formalize pulse and blood saturation readings) could be described by following triples:

- *PPG_URI, registeredAs_URI, sensor_URI;*
- *PPG_URI, measures_URI, pulse_URI;*
- *PPG_URI, measuers_URI, saturationO2_URI;*
- *PPG_URI, modelNumber_URI, 9560BT.*

Realization of agents

The blackboard is implemented as a transparent custom shared memory datastore. There is no separate process for handling the datastore, agents in the system can use a set of API calls to insert and query data (triples) from shared memory. The datastore serves as a mid-term memory for storing data and as a postbox between different agents. It can be also seen as a deductive database for reasoner, using a rule language for rule-based generation of new facts.

We demonstrate the common model of an agent on the sensor adapter (agent) for the selected PPG sensor. The agent runs always in background, monitoring whether the sensor is online or not. When the sensor device is turned on the agent establishes connection immediately and receives all the output data generated by the device, which is inserted into the database in the form of triples. The agent also queries regularly the database for getting its configuration parameters or other input data.

The example output of a single measurement looks as following („#3920“ is a unique key for binding the particular set of different triples into one entity):

- *PPG_URI, sample_URI, #3920;*
- *#3920, pulse_URI, 73;*
- *#3920, saturationO2_URI, 98;*
- *#3920, timestamp_URI, 1264423298.*

With our architecture we can apply several formal reasoners as different data processing agents in parallel for the same physical datastore. For data processing and decision making, e.g. emergency condition detection, simple hardcoded agents can be installed, as well as full formal reasoners like Jena, Gandalf and others for handling thousands of patient and context related facts. For example, we can describe processing of semantically represented rules and data of SpO2 meter with a blackboard agent executing Prolog programs.

It is known that normal SpO2 value for healthy people is 96-99%, whereby the value < 95% indicates respiratory insufficiency and the value < 90% indicates hypoxia with the need of emergency treatment. However, for people with COPD (chronic obstructive pulmonary disease) a normal SpO2 diagnosed value is between 88-92%. Suppose we have a precalculated fact in our memory datastore being specific to the patients's critical SpO2 value: *patient_URI, critical_SpO2_URI, 90.* The respective agent could check whether the SpO2 value is critical or not by querying Prolog the following rule, which answers „false“ if no SpO2 sample is found below critical value and „true“ if all preconditions of the rule are satisfied (and variable X is instantiated to one particular SpO2 value which is under the limit):

```
spo2_problem(X) :-
fact('PPG_URI', 'sample_URI', Y),
fact(Y, 'saturationO2_URI', X),
fact(patient_URI,'critical_SpO2_URI', Z),
X<Z.
```

For uploading data to next MDH levels, we have a separate agent responsible for exchanging facts with a data aggregation server for sending relevant data items from the local memory database to the server and downloading/receiving new commands and configurations, which will be stored back into the memory database available for all other agents.

Transmitted data amounts could become relatively large, for example, one pulse-oxymeter sample as it was presented previously (if encoded into full human-readable string) takes 350 bytes. Compression or re-encoding of URIs containing triples is possible, though one must consider the trade-off for compatibility with potential data consumers in other MDH layers.

For reducing the load of communication channels, and especially central aggregation servers (can incorporate monitoring data from several hundreds of patients), a considerable amount of raw sensor data processing and analyzing is done locally on home telemonitoring gateway. Therefore, only the results/reports of the data processing and irregular individual values shall be uploaded by default, while other levels have possibility to query additional raw (gathered) data.

Implementation and testing

For testing the proposed architecture, we implemented the software for enhanced DVB receiver. This device is equipped with Bluetooth and Zigbee interfaces to support wireless medical and presence sensors. The device has 300MHz 32bit MIPS type CPU and non-RT Linux OS. Wireless Onyx II 9560 SpO2 sensor was used for testing, see (Fig. 1.). The processed information was sent to the hospital server using SOAP (Simple Object Access Protocol) messages.

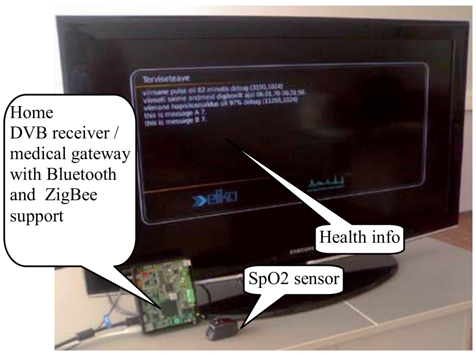
Home DVB receiver / medical gateway with Bluetooth and ZigBee support
Terviseteave
Health info
SpO2 sensor

Fig. 1. Instrumentation of home monitoring system

Blackboard on home controller is a custom memory database, reasoning agents use SWI-Prolog (Version 5.6.58) engine. Conventional Postgres database and Jena reasoner run on hosptial server. The implementation appears to fulfill requirements of simple customizability and can handle semantic content. The most critical issue is, as expected, the performance of the reasoner on smart home controller, because of weak hardware platform. In real tests we measured an average (of 750 tests) of Prolog reasoning time of 310 ms of 4500 facts and 420ms of up to 7500 facts. The measured worst case reasoning time was 2sec, which is clearly sufficient to discover patient emergency conditions quickly enough and the home factset should not exceed 1-2 thousand facts by our expectations.

Future work and conclusions

The further work will focus on optimization of the software implementation and integration of feasible set of medical and smart environment domain ontologies. The target is real life use of the developed telecare architecture and its software implementation. Experiments show that the proposed software architecture satisfies real life needs of modern patient home monitoring solutions, semantic reasoning and computerized personalization.

Acknowledgements

This research has been supported by European Regional Development Fund and Competence Centre program of Enterprise of Estonia.

References

1. The e–Health Innovation Professionals Group. Online: http://www.health–informatics.org/tehip/tehipstudy.PDF.
2. **Doughty K., Cameron K., Garner P.** Three generations of telecare of the elderly // Journal of Telemedicine and Telecare, 1996. – Vol. 2. – No. 2. – P. 71–80.
3. **Barnes N. M., Edwards N. H., Rose D. A. D., Garner P.** Lifestyle monitoring – technology for supported independence // IEEE Computing and Control Engineering Journal, 1998. – Vol. 9. – No. 4. – P. 169–174.
4. **Amaral T., Hine N., Arnott J. L.** Integrating the Single Assessment Process into a lifestyle–monitoring system // 3rd International Conference On Smart homes and health Telematic (ICOST 2005), 2005. – P. 42–49
5. **Sixsmith A.** An evaluation of an intelligent home monitoring system // Journal of Telemedicine and Telecare, 2000. – Vol. 6. – No. 2. – P. 63–72.
6. **Chen L., Nugent C. D., Mulvenna M. D., Finlay D. D., Hong X.** Semantic Smart Homes: Towards Knowledge Rich Assisted Living Environment, in Intelligence on Intelligent Patient Management (Ed. by McClean S., Millard P, El–Darzi, Nugent C). – Springer, 2009.
7. **Kim E., Choi J.** An Ontology–Based Context Model in a Smart Home, Notes in Computer Science. – Publisher Berlin/Heidelberg: Springer, 2006.
8. **Turner K. J., Docherty L. S., Wang F., Campbell G. A.** Managing Home Care Networks (in Robert Bestak, Laurent George, Vladimir S. Zaborovsky and Cosmin Dini (eds.)) // Proc. 8th Int. Conf. on Networks, IEEE Computer Society, 2009. – P. 354–359.
9. **Augusto J. C., Liu J., McCullagh P., Wang H., Yang J.–B.** Management of uncertainty and spatio–temporal aspects for monitoring and diagnosis in a Smart Home // International Journal of Computational Intelligence Systems. – Atlantis Press, 2008. – No. 1(4). – P. 361–378.
10. **Leonaite A., Vainoras A.** Heart Rate Variability during two Relaxation Techniques in Post-MI Men // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 5(101). – P. 107–110.
11. **Dimitrov D. Tz., Guergov S., Ralev N. D.** Multifunctional Adaptive System for Physiotherapy with Measurement Devices // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 5(101). – P. 111–114.
12. **Hayes–Roth B.** A blackboard architecture for control // Artificial Intelligence, 1985. – No. 3(26). – P. 251–321.
13. International Health Terminology Standards Development Organisation. Online: http://www.ihtsdo.org (2010).

Received 2010 02 15

A. Kuusik, E. Reilent, I. Lõõbas, A. Luberg. Data Acquisition Software Architecture for Patient Monitoring Devices // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 9(105). – P. 97–100.

Patient remote monitoring has continuously rising importance for aging countries. Computer based assisted living systems are too difficult to use by elderly people, therefore it has been proposed to extend home multimedia devices with patient monitoring functionalities. However, such implementations are usually platform and sensor specific making their extension and reuse complicated and time consuming. We propose an agent based software architecture for embedded patient monitoring devices that is built around a common database. The solution is open for adding any kind of medical sensors or signal processing software just by writing small software adapter. System can be reconfigured during operation. As one of the unique feature the system has built in formal reasoner to detect inconsistencies among sensor data and process patient unique safety rules in real time. The solution is prototyped. Ill. 1, bibl. 13 (in English; abstracts in English and Lithuanian).

A. Kuusik, E. Reilent, I. Lõõbas, A. Luberg. Duomenų surinkimo programinės įrangos iš pacientų stebėsenai skirtų įrenginių architektūra // Elektronika ir elektrotechnika. – Kaunas: Technologija, 2010. – Nr. 9(105). – P. 97–100.

Pacientų nuotolinių stebėjimo sistemų svarba ypač didėja „senstančiose“ valstybėse. Vyresnio amžiaus žmonėms sunkiau naudotis kompiuterių sistemomis, todėl buvo pasiūlyta pacientų stebėsenos funkcijas įdiegti į multimedijos sistemas. Tačiau jutiklių taikymas tokiose sistemose sukelia nemažai problemų. Esamiems pacientų stebėsenos įrenginiams pasiūlyta programinės įrangos architektūra, pritaikoma įvairių tipų jutikliams ar signalų apdorojimo programinei įrangai. Vienas iš esminių siūlomos sistemos pranašumų – paciento būsenos parametrų stebėsena esamuoju laiku lyginant jų tikroviškumą. Il. 1, bibl. 13 (anglų kalba; santraukos anglų ir lietuvių k.).

PAPER 5

A. Kuusik, E. Reilent, I. Lõõbas, M. Parve. Software architecture for modern telehealth care systems. Journal of Advances on Information Sciences and Service Sciences, 2011, 3(2), 141 - 151.

Software architecture for modern telehealth care systems

Alar Kuusik[1], Enar Reilent[1], Ivor Lõõbas[1], Marko Parve[2]
[1]*ELIKO Technology Competence Centre, Tallinn, Estonia*
[2]*East Tallinn Central Hospital, Tallinn, Estonia*
[1]*Firstname.Lastname@eliko.ee,* [2]*marko.parve@itk.ee*

Abstract

Results of several research groups indicate that modern home telehealth care systems should support patient personalization and context awareness. To deal with accompanying increase of data amount and processing complexity, a semantic reasoning approach is proposed. However, so far there are no practical, system level software architectures proposed to address all related issues within one complete solution. We describe a developed RDF blackboard based data processing solution for smart home telecare supporting off-the-self reasoning tools and existing ontologies.

Keywords: *telecare, patient monitoring, assisted living, smart home, semantics, agent software architecture, formal reasoning.*

1. Introduction

For the delivery of public healthcare, telecare (tele-homecare), an essential part of eHealth technologies, provides the cost effective way to manage burdens on public services caused by an increasing proportion of elderly and chronically ill people [1]. Additionally, to reduce demand on clinics and home visits for patients the long term human monitoring is believed to be an effective way for early discovery of health risks. Increased acceptance and adoption of preventative care regimes within 'wellbeing' programs is important, for example smoking cessation and weight reduction also require active patient monitoring at home. While the early patient telecare systems were designed just for acquisition and offline monitoring of health parameters, and were later extended with real-time monitoring of safety boundaries, then the modern patient home monitoring solutions essentially analyze the context and focus on investigation of long term trends. Certainly, to discover slight drifts in patient's data the measurement context has to be taken into account. Strong external factors like recent physical activities, stress, and irregular lifestyle may hide changes in patient's data over a long period. From one side, the existing telecare solutions are designed as classical data acquisition systems not supporting context information for detailed patient data analysis. From the other side, if context information is provided, it is usually not machine tractable (outside of the particular system) making large scale statistical data analysis virtually impossible. Present paper describes *content centric* architecture for home monitoring solutions supporting machine reasoning of semantically encoded sensor data and context information.

2. Previous work

2.1. Progress of home telecare services

Telecare and assisted living is an emerging topic of cost efficient health care. Average telecare savings from the community perspective that are mentioned in the literature or achieved in trials are about 15-30% [2]. As described by Doughty [3] et al., the first generation telecare technology solutions enable to summon help in emergency, the second generation provide automated detection of emergencies, and the third enable monitoring of deterioration of human well-being. The first generation solutions in the form of panic buttons are available to the whole well-developed world. The main disadvantage of such technology, though, is that patient may be unable to proceed with an emergency call, or one tends to use the emergency hotline improperly. Significant involvement of qualified medical staff is a disadvantage as well.

Commercial solutions of the second generation, e.g. Well@home, Zydacron, Docobo, and Philips Motiva, have been available for about 10 years allowing automated monitoring of patient's safety based on periodic measurements. One of the significant advantages of the second generation is the reduced need for professional medical assistance. However, existing practical solutions support only resting condition measurements because of complexity of processing dynamically changing context. On the other hand, trends clearly show improvements in the context awareness, e.g. use of accelerometer data in mobile telemedicine [4]. For example, recent physical activities detected prior to the heart rate measurements help to avoid faulty emergency actions while still maintaining high sensitivity of the system. For simple personalization and context awareness policy-, i.e. ECA rule-based home care systems are recommended by Turner [5] et al. Rule-based data interpretation for home care solutions has been investigated in various research projects [6, 7] but there are no standardized frameworks for simple reuse of the domain knowledge.

The third generation telecare solutions, also called lifestyle-monitoring by Barnes [8], essentially focusing on continuous monitoring and analysis of Activities of Daily Living (ADL), primarily including the duration, frequency and patterns of physical activities as described by Amaral [9] et al. giving *long term* context to medical sensor data. Wide deployment of micro-mechanical sensors on mobile phones just recently opened practical possibilities to allow ADL monitoring [10]. Continuous ADL monitoring carries essential information about degradation of well-being [9] and can be used for discovery of emergency conditions, even without medical parameter sensing [11]. Some recent telecare system prototypes with lifestyle monitoring are created by Amaral, Kaushik [12] et al., and others. However, there are several principal issues that are not addressed in such prototypes. Information describing patient daily activities may be considered very delicate and in some countries there are legal restrictions for centralized processing of such data. Amount of collected ADL information is remarkable and therefore on-site data aggregation and processing is feasible as well. Development of rules for processing broad range of ADL information is a complex and time consuming task stressing the needs for knowledge formalization and reusability among many telecare patients.

2.2. Smart Home technology targeting modern telecare

Smart Home (SH) systems are the most widely used technology platforms for third generation telecare targeting home based elderly and disabled people. Such improved SH systems have (wireless) interfaces for medical sensors, typically cardiovascular and respiratory monitors, weight scales, blood sugar meters, in addition to traditional micro-climate and entertainment control interfaces. Native SH components, like movement detectors, video cameras, and home appliance control circuits, efficiently provide required ADL context data. Recently, mobile communication devices equipped with micro-mechanical sensors became useful for cost-efficient monitoring of physical activities and emergency conditions like falling down. Therefore, importance of mobile telemedicine is rapidly increasing but it is significantly harder to take into account the ADL information collected at different locations and environments. In principle, the telecare software solutions described in present paper are also applicable to mobile telemedicine. However, specific issues related to handling dynamic ADL data sources will not be deeply analyzed.

Rule- and logic-based (including Fuzzy logic-based) control is the leading method for SH implementations [13]. Therefore, rule-based patient data handling and personalization is natural for ADL aware telecare. However, as stressed by Nugent and Chen [14] et al., even without telecare functionality the SH environments are producing massive amounts of data and, until supplemented with a well-defined meaning, the potential of smart homes assisting capabilities will not be fully achieved, and propose semantic data integration approach demonstrated in their SemanticsAtHome project. The importance of semantic context data fusion has also been stressed in AALIANCE telecare roadmap [15]. The main disadvantage of conventional, non-semantic approach is complexity of (automated) reuse of existing knowledge, e.g. interpretation of rules encoded within different terminology systems. Apparently, it is very hard to copy and reuse the information and knowledge derived from a particular SH installation because a) there are no widely accepted standards for presenting sensor-actuator data, and b) systems typically use low-level data formats and its conversion into human and machine understandable formats for wide reuse is weakly motivated and a hard task. Semantic assisted living projects claim that processing of formal semantically enriched content, its analysis, and decision support for intervention can be done more easily. As described by Redondo et al. [16], semantic representation simplifies SH service composition which is also important for adaptive telemonitoring. Additionally, semantically annotated data is more easily comprehensible for external services, e.g. medical decision support systems, Electronic Health Record (EHR), and global health statistic databases. Within the MATCH project Turner [17] proposed to use ontologies for data clustering. However, the works of both Nugent and Turner do not specify any real existing ontologies, essentially medical ones, to be used for achieving SH and telecare data interoperability in practice. Authors of [18, 19] propose different ontologies for describing a SH environment context. Comprehensive list of existing alternatives presented by W3C [20] show that is quite unlikely to agree on a single ontology to be used for smart home telecare.

3. Proposed software architecture for modern telecare systems

3.1. Functional requirements for the telecare system and its software

Based on analysis of recent related work we can say that modern telecare solutions should have the following capabilities:

- Support for ADL monitoring, in addition to the medical data acquisition.
- Support for personalization by means of customization of safety cutoff values and typical ADL behavior patterns.
- Interoperability with SH automation systems for home based patient context monitoring.
- Interoperability with mobile context sensing devices, e.g. mobile phones with positioning and acceleration sensing.
- Simple knowledge reuse and portability across different hardware based systems (sensors, communication infrastructure).
- Possible run-time renewal of medical domain expert knowledge from centralized repositories.

Semantic content-driven data processing solutions simplify satisfying described requirements for modern telecare systems. However, the existing commercial telecare systems are based on conventional client-server architecture and predefined communication protocols. Systems support restricted set of low-level messages and predefined set of hardware components. Naturally, those implementations, even though designed to be compatible with HL7 [21] patient information exchange standard, rely on predefined messages to be exchanged between the content source and the destination user.

The existing rule-based and semantic approaches of telemonitoring present theoretical State-of-the-Art and do not propose any practical implementation frameworks for their implementations. Essential feature for modern telecare systems is the extendability by means of simple reconfiguration of hardware and introduction of new knowledge in form of rules, signal processing algorithms, and services. We propose an open, agent-based software architecture for home and mobile telemonitoring which natively supports treatment of semantic content, including computationally feasible on-site formal reasoning and data aggregation.

3.2. Agent based software architecture

The proposed architecture (Figure 1) is a distributed multiagent system of asynchronous processes following classical blackboard model of Hayes-Roth [22]. The agents (typically executable processes) are running on the same hardware device (embedded controller, server, smartphone) writing data into a universal semi-realtime data-store (installed essentially on the same hardware device for high access speed) while every agent can asynchronously access *all* data inserted to the data-store. The complete system containing different hardware platforms forms a Monitoring Device Hierarchy (MDH) with several blackboard-agents sets. The first (lowest) MDH level software agents run on intelligent sensors implemented on micro-controllers. For example, a wired or wireless motion detector may be a hardware platform for the first level MDH agents. The 2nd MDH level agents and corresponding blackboard data-store operate on SH controllers or mobile devices. The higher, 3...n MDH level software agents operate on hospital servers and data clouds. The simplest practical telecare software implementation contains 2nd and 3rd MDH levels, SH controller and hospital server levels, both having their own blackboard data-stores.

Essential but not limited agent set for semantic telecare data processing includes:

- Sensor agents acquiring data from individual sensors and publishing data on the blackboard. Sensor agents are responsible for propagating sensor configuration information from the blackboard back to the sensor. It is important that there is no target user (agent) specified for the acquired content in advance.
- Data processing agents performing variety of signal processing, outlier detection, and aggregation tasks, as well as discarding obsolete data. Data processing also includes controlling the behavior of the system and decision making based on the given set of rules. Thus, present data processing agents are essentially formal (semantic) reasoners.
- Output agents communicating with host services (devices on higher MDH levels), for example an output agent running on SH central controller is responsible for exporting patient's data to the hospital system and downloading new configuration settings, profiles, commands, and messages.
- HCI agents used for adaption of user interfaces according to access profile on the home controller screen or the hospital web site, also for alerting and user feedback processing.

For communication between the second and higher MDH levels conventional SOAP messages may be an optimal solution. However, between (wireless) sensor devices and the gateway level device the custom communication protocols, given by sensor manufacturers, are difficult to avoid. The agents within one machine are built around a fast and transparent RDF [23] data-store implemented in shared memory for embedded home controllers, or in conventional database for servers. In the future it would be possible to use internal data-stores of reasoning tools for RDF encoded data [24]. Making use of the RDF representation for data encoding allows handling and saving different structures (like sensor readings, configurations, profiles, commands, reports, etc.) in the same universal manner. As mentioned, data written to the data-store is available to every process running on the same device. Thus, every write is like a broadcast. The RDF data-store serves three roles:

- A postbox between different agents within one controller device (including transparent external world communication with the aid of dedicated agents).
- A low-latency in-memory universal data-store for keeping data.
- A deductive database, using a rule language for rule-based generation of new facts.

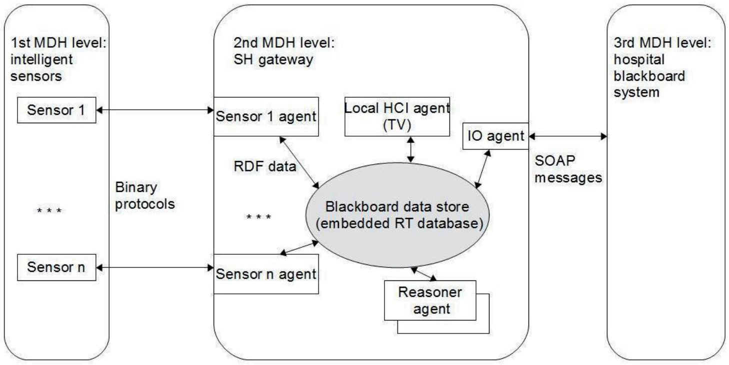
1st MDH level: intelligent sensors
Sensor 1
* * *
Sensor n
Binary protocols
2nd MDH level: SH gateway
Sensor 1 agent
Local HCI agent (TV)
IO agent
RDF data
Blackboard data store (embedded RT database)
* * *
Sensor n agent
Reasoner agent
SOAP messages
3rd MDH level: hospital blackboard system

Figure. 1. 3-level example of proposed telehealth care architecture

The main advantage of the blackboard based architecture in comparison to the popular socket based systems is that there is no need to specify target user processes (of local real-time data) in advance. Agents producing the data can work independently from the receiving agents regardless of their existence. For monitoring applications it is crucial not to miss any incoming events while data processing has weaker real-time constraints. By using the proposed architecture it is always possible to add or modify content processing agents without the need to modify sensor specific and other low-level, performance optimized, agents. It is possible to run (even simultaneously) different sensors that measure and output the same parameter. As no sockets are used in the implementation of the blackboard on the MDH level 2 device, certain load on the operating system (kernel) is reduced that is essential for acquisition of streaming sensor data, e.g. ECG. Benchmarking experiments show that writing data into the implemented RDF data store is at least 100 times faster in comparison with the conventional SQLite database.

4. Semantic content presentation

We propose using full RDF representation of data within the system, starting right from the sensors. The RDF format (and associated OWL-based ontology systems) for defining and describing relations and concepts is emerging for different computerized data processing applications using "the semantic web" standards and technologies. RDF data (knowledge) encoding simplifies integration with (different) existing formal knowledge processing tools. As noticed by Tian [25] et al., the multimodal reasoning has advantages for efficient processing medical domain information that natively contains both rules and behavioral data. There exist some XML-based semantic data representation solutions developed for sensors, like SensorML, and Hydra middleware. However, those solutions are not fully RDF compatible which makes formal reasoning more complex, which also applies to encoding and handling of structures from various different domains. From the other side, XML format has certain communication overhead which is especially problematic for wireless sensors. The additional advantage of using standard RDF is the possibility of using existing SPARQL tools.

In our approach we stress to use real existing ontologies available on the web. That way we can guarantee that correct interpretation of information is always possible for content user (while correct use still remains the responsibility of the user). The proper use of keywords in medical domain is especially important. For example for "heart rate" there are around 20 different terms in use, some of them are equal while some have specific flavor. Inadequate labeling and later interpretation of sensor signals may lead to critical situations for patients. From the other side, if used ontologies are published and accessible, automated conversion of information is a relatively simple task. The widely accepted good medical taxonomy is SNOMED CT (Systematized Nomenclature of Medicine - Clinical Terms) [26]. Wordnet linguistic vocabulary can satisfy the broadest range of information formalization needs. However, the large number of existing and competing semantic sensor ontologies [27] demonstrate that simultaneous support for multiple terminology systems is a must.

Tables 1 and 2 show representation examples of semantic medical sensor data encoded into RDF triplets. Every sensor agent (adapter) in the system is identified by a URI. Using this URI all relevant information concerning any particular agent is easy to find, for example agent's configuration data, description, and output data. Table 1 presents configuration data about the pulse oximeter sensor agent, a particular wireless instrument for taking photoplethysmographic (PPG, tissue transparency) measurements and outputting heart rate and blood oxygen saturation level (SpO_2).

Table 1. RDF coding of PPG sensor description:

Subject	Property	Value
http://www.eliko.ee/demo/ssg/schema#nonin_onyx2	http://www.eliko.ee/demo/ssg/schema#Type	http://www.csiro.au/Ontologies/2009/SensorOntology.owl#Sensor
http://www.eliko.ee/demo/ssg/schema#nonin_onyx2	http://www.csiro.au/Ontologies/2009/SensorOntology.owl#ModelNumber	9560BT
http://www.eliko.ee/demo/ssg/schema#nonin_onyx2	http://www.csiro.au/Ontologies/2009/SensorOntology.owl#measures	http://bioinfo.icapture.ubc.ca/subversion/SIRS/clinicalphenotype.owl#HeartRate
http://www.eliko.ee/demo/ssg/schema#nonin_onyx2	http://www.csiro.au/Ontologies/2009/SensorOntology.owl#measures	http://bioinfo.icapture.ubc.ca/subversion/SIRS/clinicalphenotype.owl#SaturationO2
http://www.eliko.ee/demo/ssg/schema#nonin_onyx2	http://xmlns.com/wordnet/1.6/Configuration	#3021
#3021	http://xmlns.com/wordnet/1.6/Sleep	1

The first triple says that the agent identified by the URI "http://www.eliko.ee/demo/ssg/schema#nonin_onyx2" is registered as a physical PPG sensor (with the dedicated software agent) in our system. The second fact records the model number of the sensor used and the two following facts indicate that the sensor measures heart rate and SpO_2. The last two rows demonstrate the usage of RDF hierarchy – one triple represents the configuration data of the agent and the other one (linked through the system internal ID #3021) is a PPG device agent parameter with a value (sleep time 1 sec). URIs for different concepts are taken directly from ontologies, if available, or are otherwise composed artificially. For example "`urn:snomed-ct:271650006`" represents SNOMED CT term "271650006 Diastolic blood pressure (observable entity)".

The same sensor adapter writes its output data (of one measurement) as shown by the four rows in the Table 2. The first triple defines the data to be a sample by the particular agent and links all triples together as one object. The remaining three triples describe the result of the measurement – indicating that the measured heart rate value was 73 beats per minute and the blood oxygen saturation was 98%. Also the timestamp of the moment when the measurement was taken is recorded.

Table 2. RDF coding of PPG sensor output data:

Subject	Property	Value
http://www.eliko.ee/demo/ssg/schema#nonin_onyx2	http://www.eliko.ee/demo/ssg/schema#Sample	#3920
#3920	http://bioinfo.icapture.ubc.ca/subversion/SIRS/clinicalphenotype.owl#HeartRate	73
#3920	http://bioinfo.icapture.ubc.ca/subversion/SIRS/clinicalphenotype.owl#SaturationO2	98
#3920	http://knowledgeweb.semanticweb.org/iriba/ontologies/ResultOntology#timestamp	1264423298

5. Semantic reasoner agent integration

Proposed software architecture with RDF data-stores natively supports semantic rule-based reasoning – dedicated agent(s) can read any data from the blackboard data-store and write output back there for any other agent to use. Some applications for reasoning agents can be outlined:

- Derivation of personal safety threshold parameters from potentially large set of expertise facts, which are difficult to handle correctly by humans.
- Continuous real-time validation of safety requirements of patient supporting dynamically changing rules.
- Aggregation of sensor data with semantically annotated output.
- Interpretation of semantic context information.
- Profile based selection of HCI information optimized for patient itself or medical professional, etc.

Through our architecture we can apply several formal reasoners as different data processing agents in parallel for the same physical data-store. Apparently, the feature of concurrent processing is crucial to maintain real-time service for incoming sensor data. For real-time sensitive decision making, e.g. emergency condition detection, simple hard-coded rule-processing agents can be installed, as well as formal reasoning tools like Jena, CHR, Gandalf, Otter (Prover9), and others, for processing thousands of patient and context related facts.

5.1. Example of semantic reasoning with Prolog

For better illustration of using formal reasoning in the system, we can examine an example of processing of semantically represented rules and medical data with a blackboard agent executing Prolog programs.

To demonstrate some issues of the patient's data processing in home telecare system let us consider the following case of monitoring SpO_2 level on people with lung diseases. Too low oxygen saturation level in blood may have lethal consequences for such patients. However, from the healthcare practice it is known that normal saturation level depends on diagnosis of the particular patient. For example, normal SpO_2 indication for healthy people is 96 – 99%, less than 95% indicates respiratory insufficiency, and less than 90% indicates hypoxia with the need of emergency treatment. On the other hand, for people with chronic obstructive pulmonary disease (COPD) the normal SpO_2 value can be as low as 88 – 92%. It is also possible that individual harmless level of SpO_2 for a given patient is assigned by a doctor and standard limits should be discarded.

Due to that essential personalization feature, our knowledge base inside a home telecare controller may contain contradicting data. For example, based on the described patient's illnesses the rule system assumes that the minimal SpO_2 lower threshold value is "96" while at the same time other records suggest the threshold to be "90" (e.g. said by the doctor). Therefore, a defeasible logic inspired [28] reasoning is used. Our rule system keeps additional records related to the genuine data indicating origin of data and respective priority system.

For reasoning on given knowledge-base we use SWI-Prolog which is executed as a child process spawned by the corresponding adapter agent. The adapter agent feeds Prolog with blackboard data and queries. The following rule examples assume the data items are presented as individual triples separately forming facts with three arguments, e.g.: "`fact(subject, property, value)`".

Due to the space constraints and better readability long URIs (as in Tables 1, 2) actually used in the rules are abbreviated in the following examples and replaced by intuitive short strings, e.g. "`ppg_sensor_uri`" stands for the identifier of the sensor agent managing the PPG sensor device (`http://www.eliko.ee/demo/ssg/schema#nonin_onyx2`). We can monitor all available SpO_2 data (as seen in Table 2) samples produced by the given agent and evaluate the emergency level with the given Prolog rule:

```
spo2_emergency_condition :-
     get_spo2_threshold(patient_uri, T),
     fact(ppg_sensor_uri, sample_uri, X),
     fact(X, spo2_uri, N),
     N<T.
```

The rule succeeds (Prolog answers 'true') if there exists at least one measured SpO_2 sample that is below the threshold value calculated by the rule "`get_spo2_threshold`" and fails in all other cases. The following rule "`get_spo2_threshold`" is meant to resolve all contradictions in the facts defining the threshold and return allowed minimal SpO_2 value for the given patient:

```
/* first alternative - if threshold given directly by a doctor, use
it, ignore others */
get_spo2_threshold(P, T) :-
     fact(P, profile_uri, Y),
     fact(Y, thresholds_uri, Z),
     fact(Z, lower_SpO2_limit_uri, L),
     fact(L, set_by_uri, W),
     belongs_to_class(W, medical_personnel_uri),
     fact(L, threshold_value_uri, T),!.

/* second alternative - maybe there exists a threshold computed
previously by the rule system itself */
get_spo2_threshold(P, T) :-
     fact(P, profile_uri, Y),
     fact(Y, thresholds_uri, Z),
     fact(Z, lower_SpO2_limit_uri, L),
     fact(L, set_by_uri, W),
     belongs_to_class(W, rule_system_uri),
     fact(L, threshold_value_uri, T),!.

/* third alternative - if no threshold records found (by doctors or
by rules), give some general default */
get_spo2_threshold(P, T) :- T = 97.
```

The real threshold values are kept as regular facts under the patient's profile and each of them has sub-record indicating whom it was written by. Therefore, one parameter (e.g. threshold value) may have any number of values assigned if set by different sources. As values like SpO_2 threshold do not change very often but are used quite frequently then they are usually calculated in advance and stored on the blackboard, however they could be derived also during run time every time they are needed by any other rule. Under usual circumstances the rule “`get_spo2_threshold`” returns a value calculated by the system itself. However, if there exists a SpO_2 threshold value given directly by a doctor then it overrules all other possible values. For the extreme case, where no threshold records are found from the set of facts the rule returns a hard-coded value of “97”.

Another (simplified) rule (written in three alternatives in Prolog syntax) demonstrates how the system might derive the current minimal SpO_2 value for the given patient. If the patient has COPD diagnosed the value will be “88”, in case of any other pulmonary disease “92”, otherwise “96”:

```
calculate_lower_spo2_limit(P,V) :-
    fact(P, profile_uri, X),
    fact(X, diseases_uri, Y),
    fact(Y, disease_uri, copd_uri),
    V=88.

calculate_lower_spo2_limit(P,V) :-
    fact(P, profile_uri, X),
    fact(X, diseases_uri, Y),
    fact(Y, disease_uri, D),
    is_subclass_of(D, lung_disease_uri),
    V=92.

calculate_lower_spo2_limit(P,V) :- V=96.
```

The acquired value is stored under the patient's profile facts and its source is set to “rule system” indicating that the data is not given by medical staff but was automatically generated by rules instead. Responsible medical professionals can add new data at any time and it becomes superior from the rules' point of view – the software rule system has to take such “super user” facts into consideration and cannot erase or modify them.

As the central blackboard incorporates contextual and environmental sensor data besides pure medical data it is natural to make use of the contextual data in rules to enrich medical knowledge of the patient’s condition. For example, if a heart rate sample is found to be below 50 beats per minute we can predict health (or device malfunctioning) problems. Alternatively, if there is a sample above 90, derivation cannot be made immediately. With sub-queries we try to justify the high pulse first and only if it is not possible then positive result (problem found) is returned. Say, if the patient has had significant physical movement activities 10 minutes prior to the pulse measurement, we have no reason to raise alarm. Contrariwise, if the patient has not moved around or has even been in bed then a heart rate over 90 indicates health problems. Contextual data needed for calculations in this case comes typically from the set of PIR sensors in the patient’s home environment and/or accelerometer sensors attached to the patient (mobile phone, pedometer, wrist watch). The rule of the main query in Prolog syntax to raise alarm is as follows:

```
problem_suspected :-
    fact(_, 'sample_URI', Y),
    fact(Y, 'pulse_URI', X),
    fact(Y, 'timestamp', T),
    (
          X<50;
          X>90,
          (
                not(was_active_between(T, T-(60*10)));
                was_in_bed_between(T, T-(60*10))
          )
    ).
```

6. Solution testing

For testing feasibility of proposed telecare software architecture and semantic data encoding we implemented the prototype solution containing a home controller (2nd MDH level device), set of wireless Bluetooth and Zigbee sensors and PC based server emulating hospital server (3rd MDH level). Communication between the home controller and the hospital server is based on SOAP messages always initiated by the home controller for network safety reasons. According to the proposals of Continua Health Alliance [29] the home controller is realized as an advanced DVB-T receiver equipped with Ethernet and wireless sensor interfaces. The home controller has standard Linux running on 300MHz 32bit MIPS type CPU. Its blackboard is an original RDF memory database following the ideas described. Reasoning agents use SWI-Prolog (Version 5.6.58) engine. The hospital server runs conventional Postgres database. The implementation fulfills the performance requirements and interoperability framed by the hospital personnel. The most critical issue, as expected, is embedded reasoner performance due to the relatively weak hardware platform and the real time demands of the sensor communication. In benchmark tests we measured an average Prolog reasoning time of 310 ms for 4500 facts, and 420ms for up to 7500 facts. 750 tests were executed. The measured worst case reasoning time measured was 2 seconds. Achieved real life reasoning response is clearly sufficient to discover patient emergency occasions quickly enough for realistic medical knowledge base of some thousand facts.

7. Conclusions

For efficient utilization of ADL information, required for modern telecare, the data driven approach has advantages over constrained predefined solutions. Semantics driven approach is well suitable for interoperability in medical domain having strong terminology standardization background. However, existing semantic telecare proposals did not offer *practical* solutions for flexible representation of data through the accepted medical code systems that are suitable for machine reasoning. In this paper we described a flexible agent based software architecture solution that is compatible with RDF data representation, unrestricted amount of domain ontologies and supporting existing reasoning tools. Feasibility and efficiency of proposed telehealth care system architecture and data encoding solution was successfully evaluated in tests on a real telemonitoring system. Further enhancement of the system will focus on development of specific data processing agents for data validation and aggregation.

8. Acknowledgments

This research has been supported by the European Regional Development Fund, Competence Centre program of Enterprise of Estonia.

9. References

[1] http://www.health-informatics.org/tehip/tehipstudy.PDF (2005).

[2] Paré G, Jaana M, Sicotte C. Systematic review of home telemonitoring for chronic diseases: The evidence base. Journal Am Med Inform Association 2007;14, pp. 269–277; pp. 2007

[3] K. Doughty, K. Cameron, P. Garner, "Three generations of telecare of the elderly", Journal of Telemedicine and Telecare, vol. 2, no. 2, pp. 71-80, 1996.

[4] L.C. Jatobá, U. Großmann, J. Ottenbacher, S. Härtel, B. von Haaren, W. Stork, K.D. Müller-Glaser, and K. Bös, "Obtaining Energy Expenditure and Physical Activity from Acceleration Signals for Context-aware Evaluation of Cardiovascular Parameters," IFMBE Proc. 18, Springer-Verlag Heidelberg, pp. 475–479, 2007

[5] F. Wang, K. J. Turner, Towards personalised home care systems, in Proc of the 1st international conference on Pervasive Technologies Related to Assistive Environments, Article 44, ISBN:978-1-60558-067-8, 2008

[6] J. C. Augusto, J. Liu, P. McCullagh, H. Wang and J.-B. Yang. Management of uncertainty and spatio-temporal aspects for monitoring and diagnosis in a Smart Home. International Journal of Computational Intelligence Systems, 1(4), pp. 361-378, Atlantis Press, 2008

[7] K. Du et al., HYCARE: A hybrid context-aware reminding framework for elders with mild dementia, in Proc of the 6th Int Conference on Smart Homes and Health Telematics (ICOST), Lecture Notes in Computer Science, Springer, Vol 5120/2008, Ames, IA, USA, pp. 9-17, 2008

[8] N. M. Barnes, N. H. Edwards, D. A. D. Rose, P. Garner, Lifestyle monitoring - technology for supported independence. IEEE Computing and Control Engineering Journal, vol. 9, nr. 4, 1998, pp. 169-174, 1998

[9] T. Amaral, N. Hin and J. L. Arnott, Integrating the Single Assessment Process into a lifestyle-monitoring system, in 3rd International Conference On Smart homes and health Telematic (ICOST), pp. 42–49, 2005

[10] G. Bieber, J. Voskamp, and B. Urban, Activity recognition for everyday life on mobile phones, in HCI (6), vol. 5615/2009 of Lecture Notes in Computer Science, pp. 289–296, Springer, 2009

[11] A. Sixsmith, An evaluation of an intelligent home monitoring system, Journal of Telemedicine and Telecare 6 (2), pp. 63-72, 2000

[12] A. R. Kaushik , B. G. Celler, Characterization of PIR detector for monitoring occupancy patterns and functional health status of elderly people living alone at home, Technology and Health Care, v.15 n.4, pp. 273-288, 2007

[13] D. Cook and S. K. Das, How Smart are our Environments? An Updated Look at the State of the Art, Journal of Pervasive and Mobile Computing, 2007

[14] L. Chen, C.D. Nugent, MD. Mulvenna, D.D. Finlay DD, X. Hong; Semantic Smart Homes: Towards Knowledge Rich Assisted Living Environment, in book Intelligence on Intelligent Patient Management; Ed by S. McClean et al., Springer, ISBN 978-3-642-00178-9, pp. 279-296, 2009

[15] G. v. d. Broek et al. (Ed by) AALIANCE Ambient Assisted Living Roadmap, IOS Press 2010, doi:10.3233/978-1-60750-499-3-62 , 2010

[16] Rebeca P. Díaz Redondo, Ana Fernández Vilas, Manuel Ramos Cabrer, José Juan Pazos Arias, Jorge García Duque, Alberto Gil Solla, Enhancing Residential Gateways: A Semantic OSGi Platform, IEEE Intelligent Systems, vol. 23, no. 1, pp. 32-40, 2008.

[17] K. J. Turner, L. S. Docherty, F. Wang and G A. Campbell, Managing Home Care Networks, in Proc. 8th Int. Conf. on Networks (ICN), IEEE Computer Society, NY, pp. 354-359, ISBN 978-1-4244-3470-1, 2009.

[18] E. Kim and J. Choi, An Ontology-Based Context Model in a Smart Home, Lecture Notes in Computer Science, Volume 3983/2006, Springer, Heidelberg, ISSN 0302-9743, DOI: 10.1007/11751632_2, 2006

[19] Context Inferring in the Smart Home: An SWRL Approach, in Proc of the 21st Int Conf on Advanced Information Networking and Applications Workshops, Vol. 02, ISBN:0-7695-2847-3, pp. 290-295, 2007

[20] http://www.w3.org/2005/Incubator/ssn/wiki/Review_of_Sensor_and_Observations_Ontologies (2011)

[21] http://www.ihe.net (2011)

[22] B. Hayes-Roth, A blackboard architecture for control, Artificial Intelligence, 26(3), pp 251–321, 1985

[23] www.w3.org/RDF (2011)

[24] B. Parsia, An Introduction to Prolog and RDF, XML.com, http://www.xml.com/pub/a/2001/04/25/prologrdf/, 2001

[25] J. Tian et al., Multi-Modal Reasoning Medical Diagnosis System Integrated With Probabilistic Reasoning, Int Journal of Automation and Computing 2, pp. 134-143, 2005

[26] http://www.ihtsdo.org (2011)

[27] M. Compton, C. Henson, L. Lefort, H. Neuhaus and A. Sheth, A Survey of the Semantic Specification of Sensors, In Proc of the 2nd International Workshop on Semantic Sensor Networks, 8th Int Semantic Web Conference (ISWC), Washington, pp. 17-32, 2009

[28] D. Nute, Defeasible Logic. In Web Knowledge Management and Decision Support, Springer, ISBN: 978-3-540-00680-0 , pp. 151 – 169, 2003

[29] R. Carroll, R. Cnossen, M. Schnell, D. Simons, An Interoperable Personal Healthcare Ecosystem, IEEE Pervasive Computing, Vol. 6, No. 4, pp. 90-94, 2007

PAPER 6

E. Reilent, A. Kuusik, M. Puju. Real-time data streaming for functionally improved eHealth solutions. 2012,International Conference on Biomedical and Health Informatics (BHI2012), Hong Kong and Shenzhen, China, 2-7

Real-time data streaming for functionally improved eHealth solutions

E. Reilent, A. Kuusik *Member, IEEE,* M. Puju

Abstract – Present eHealth solutions developed for EHRs lack for motivation of use for competing healthcare enterprises and attraction for citizens. Proposed Universal Health Repositories supporting both clinical and home-measurement data related to telemedicine and wellness applications may force both institutions and individuals for active data sharing and exchange. However, HL7 protocol based communication solutions and installable software traditionally used in professional healthcare does not suit well for rapidly changing user driven application field. In the paper we describe a developed mobile data streaming and completely web based real-time access solution. The software has been developed for demonstrating novel services within an existing nation-wide eHealth system. The long term goal deploying similar applications is engage more citizens and healthcare enterprises to use eHealth services.

I. Introduction

Despite the effort of eHealth standardization bodies like Certification Commission for Health Information Technology (CCHIT) and IHE in US or European Commission, there are significant interoperability problems between health information solutions of different healthcare enterprises. However, as stated in global analysis the interoperability and standardization are crucial to allow widespread use of the emerging technologies like telemedicine, to enable them to benefit from the uniformal markets and to contribute to its completion [1]. Beside of that, as agreed by European Commission in 2008, European Union member states should have developed national regulations for telemedicine as an enhancement of conventional eHealth solutions by the end of 2011 [2]. As seen by an example of Estonian eHealth solution eTervis [3], which was the first launched nationwide EHR solution [4], operational since 2008, relatively small amount of existing electronic patient data has been (a) made accessible for the patients through the web portal, (b) accessed and reused by the patients. Limited amount of available EHR data and weak popularity of the website among the patients restricts third parties to create novel services for citizens as targeted for the eTervis system. Apparently, the healthcare enterprises are not motivated sharing *high quality* patient data with other hospitals acting as direct competitors for their customers (patients). Based on extensive analysis by Fontain et al. there is so far no documented evidence of direct savings through the EHR data *exchange* in primary care [5]. In particular case Estonian national legislation is forcing hospitals to share certain EHR data. However, in real life, legal requirements cannot guarantee the equal quality and, especially, force to publish *all* meaningful personal health information available within each single healthcare institution. Interoperability issues of different IT solutions used by hospitals, uncertainties of HL7 v3 messages used, require manual work prior data publishing add additional complexity and reduce the content provider side motivation even more. At current implementation eTervis does not handle patient created content. Quite remarkable popularity of (unfortunately discontinued) Google Health and Microsoft Health Vault indicate that, from the user perspective, there are clear benefits for interoperable EHR solutions to make health service competition really present.

The real life situation with eTervis leads to an opinion that state control and purely formal requirements (at particular stage of the existing EHR exchange infrastructure) cannot fundamentally improve utilization of the solution among citizens and healthcare enterprises. Discontinuation of Google Health indicates that similar basic EHR data exchange is not sufficiently vigorous business model for repository keepers. eTervis and similar EHR data warehouse maintainers need novel approaches to admit buildup of new applications for better attraction of both healthcare institutions and citizens.

II. From EHR to UHR

Possible applications where both hospitals and citizens can profit from exploiting common data stores are *user controlled* (a) wellness monitoring and (b) telemedicine services. Such rapidly interest gaining applications may overrule protectionism of institutional content owners and definitely can attract more users to consume eHealth services including existing feature set. Technically speaking, extended proposed eHealth repositories supporting telecare and wellness information shall be compatible with UHRs (Universal Health Records) combining both imprecise home-

Manuscript received November 10, 2011. This research has been supported by European Regional Development Fund, Competence Centre program of Enterprise of Estonia.

E. Reilent is with the Eliko Competence Centre, Teaduspargi 6/2, Tallinn 12618, Estonia

A. Kuusik is with the Eliko Competence Centre, Teaduspargi 6/2, Tallinn 12618, Estonia (corresponding author, e-mail: alar.kuusik@eliko.ee, phone: +372-659-9881)

M. Puju is with Tallinn University of Technology, Ehitajate tee 5, Tallinn 19086, Estonia

based measurement and high quality EHR data. Similar idea of supporting user generated data is presented in Australian eHealth standard IT-014 [6]. To justify the idea of common nationwide UHR repository is popularity of fitness services from Runkeeper, Sportlyzer, Garmin, Polar. Similarly to telehealth services from Docobo, Doc@home and other there is insufficient confidence for users that the sensitive and safety related data has been handled by professionals and data retention is guaranteed throughout the human lifetime. Besides of our main target of user attraction eHealth driven telemedicine is a promising way for healthcare cost reduction [7].

A. Extending functionalities of the eHealth solution

In context of existing eTervis system we started to develop novel (demo) telecare services to be realized on top of existing IT infrastructure and web portals. The demo applications shall attract both citizens and institutions to evaluate in practice the potential of user application driven eHealth data warehouses in the further. Additional agreed prerequisites were generic web access i.e. no need for special software installation and functionality that is technically novel for traditional eHealth solutions. As a motivated and interesting application the mobile devices based streaming and real-time monitoring of cardiac signals was selected. Practical applications kept in mind include postsurgical monitoring of heart disease (CVD) patients and risk pregnant condition (fetal monitoring). The current state of art in RT ECG monitoring over mobile devices is proprietary protocols and special workstation software [8]. According to the European Society of Cardiology (ESC), it is expected that by 2015 12 million Europeans will have a heart failure [9]. A review paper [10] analyzing scientific publications issued between 1966 and 2006 on telemonitoring of chronic heart failure concluded that telemonitoring might be an effective strategy for disease management of high-risk heart failure patients. Good overview of recently developed mobile telecare solutions by Kang et al. [11] demonstrates clearly that the development focus is on data acquisition, data presentation – absolutely essential for carers and clinicians - has been left out of attention. On the same time, overview of Standing et al. demonstrates that eHealth software interoperability and use issues are important restrictions for the users [12]. Attempts to deploy web based and handheld device technologies for clinical applications and data access have not been successful because of performance issues [13]. Today the computing power of handheld devices exceeds capabilities of desktop computers five years ago. We believe that due to the rapid evolution of mentioned technologies such components may be key enablers for UHR applications. It has to be stressed, that the technological solution for heart data streaming was implemented separately from eTervis to avoid possible failures of public service. For certain testing public PHRBox [14] data repository was used. The patient experiments are just about starting.

III. Mobile RT Monitoring Solution for eTervis

A. Existing functionality and interoperability of eTervis

Currently the Estonian eTervis solution is offering EHR services for cross-institutional patient data exchange. System is organized as distributed data cloud with universal access control system. Image archive as most actively used data service is based on conventional Picture Archiving and Communication System (PACS). Health records are stored in relational databases. Data exchange between software modules is based on HL7 v3 messages. SNOMED CT nomenclature is used for content annotation. HL7 XML protocol selection is completely proper for hospitals and healthcare enterprises but may be too complex for personal use. For example, home nursing epicrisis contains of 1500 XML lines. For RT streaming signals (e.g. ECG) current solution is too much resource consuming.

Goal of the current project is to create and evaluate streaming data upload and web based real-time access interfaces for eTervis data store suitable for user driven mobile telecare applications.

B. Mobile data streaming use case and requirements

Similarly to mentioned mobile telecare solutions the post-infarction or postsurgical patients are given simple (1-lead) ECG monitoring devices on their leave from hospital. Sensor connects via the Bluetooth link to the user's gateway device (typically smartphone) forwarding measurement data to server. ECG measurements are encoded as standard EDF [15] files which together with additional knowledge formulates a proper EHR entry. Especially important are the activity of the patient during the ECG recording (e.g. lying, walking, sitting, exercising) and feeling or complaint (good, bad, weak, pain, etc.).

Both ends of the monitoring link can be assumed to be mobile. Another key feature for the system is to satisfy both the live view possibility during the measurement and access to complete recordings afterwards. While a regular EHR/PHR web environment has no time-critical constraints and is usually implemented by using full scale of contemporary technologies the live view solution, on the other hand, should be as much platform-independent as possible and not rely on any special frameworks – therefore it makes use of simple HTML and Javascript on the client side and fast lightweight CGI programs and in-memory buffers on the server side. The overview of the data flow in the system is presented in Figure 1.

C. Data flow

Doctor's live view interface works asynchronously from the patient's gateway device. Bluetooth sensor devices are connected to and driven/controlled by patient's gateway (e.g. phone) which decodes proprietary sensor protocols and forms standard EDF file with filled out header fields and real data values in records as they are received in real time. On the other hand, these EDF records (segments) have to be sent to the central server over the Internet (WLAN, mobile communication) as they are produced during the measurement. The segments of the EDF file and supporting annotations (e.g. context) are encapsulated into Google's Protocol Buffer [16], a faster and smaller substitute for XML encoding.

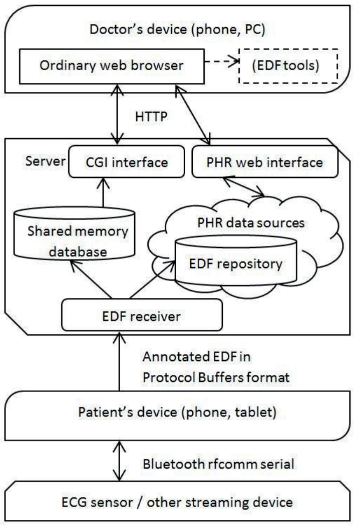

Fig. 1. Principal scheme of data transfer from sensor to doctor's screen.

On the server side the decoded EDF file is stored into the proper repository as is and also entry about the measurement act is made to the patient's profile in PHR with annotations not fitting to EDF header. This entry also contains links to the EDF file and to a dedicated web page for live view. However, we use a separate channel for live view and do not access the data through the general PHR web application, therefore, the software agent who receives the streaming EDF data writes all incoming samples also to the shared memory database.

Patients and doctors can use web interface to access PHR profiles and follow the links of specific measurements to be able to download EDF file (for opening with standard EDF tools), or open live view web page for seeing the real time stream flow. The lightweight live view page fetches recent samples form the server and renders the live graph by making calls to server side CGIs which provide latest measurement results found from the memory database.

D. *Data structures for serving live view web interface*

The central component for fast data handling in purpose of enabling live view with web page is the shared memory database described in [17]. For every signal in an incoming EDF file one entry is created into the memory database that is updated during the measurement. After the measurement has finished and no more live stream exists the corresponding entry is removed from the memory. Signals are separated in memory in contrast to EDF file structure with respect to simpler access at request handling.

Each entry has several fields for representing all the information of the EDF header (e.g. unit, sampling rate, signal name etc.), a circular buffer for containing recent samples of the last n seconds, and supporting structures for internal stream identification and bookkeeping of data flow. When a next patch of samples comes into the server it is immediately inserted into the circular buffer of the relevant entry in memory database with minimal effort and is henceforth available for serving to the viewing clients (several clients can view the same signal simultaneously).

User can follow the links given in the PHR web environment and open the live view page for the selected measurement, this page however, starts loading recent data from the server based on the parameters specified in the link. Also user can adjust several parameters (scaling, displacement, filtering). The live view page requests samples of the signal as well as static attributes (unit, starting time, range, etc.) and updating attributes (how often data is received from the patient's device, how much time passed since last data income) from the server by accessing dedicated CGI scripts/programs which in turn gets samples and other information directly from the memory database where it is easily locatable, especially these particular samples that the client is needing. The alternative to using memory would be reading the continuously growing EDF file and seeking relevant data from that which has several inefficiencies (operating system's buffering and delays, slow access time, locking issues).

E. *Experiments and RT performance evaluation*

The performance and smoothness of live graph of the viewed signal depends on the network throughput and ability of the current browser to render the samples. While the data amounts to be transferred from server to browser are quite small the network times are still varying for individual batches as are inconsistent times for processing by browser due to the multitasking operating system. The Table I compares different setups of devices, networks and data volumes. The numbers represent the moving average of the amount of signal's samples retrieved and rendered by the viewer in a loading cycle. If one cycle takes longer time than usual, the following cycles have to catch up and put more effort to rendering. The moving average varies notably, thus intervals given in the table. No data loss is allowed, smaller numbers mean smoother graph flow on the viewing interface.

IV. Solution Deployment in Practice

The system is currently put into use for ECG monitoring for post-infarction rehabilitation patients. The sensor device exerted is MegaEMG ECG [18] (Figure 2 a and b). By furnishing health records' environment with the live streaming support, that does not require special software or some certain device from the doctor, more complete EHR system is given to users. Doctors are given possibilities for convenient (CVD) patient monitoring they lacked so far. Henceforth we hope to attract more eHealth users and healthcare professionals.

TABLE I
TEST RESULTS OF RT STREAM DISPLAYING

Viewing device	*Number of samples shown per second*		
	50	*500*	*1000*
PC*, network 1	4 to 6	35 to 65	70 to 80
PC*, network 2	4 to 12	60 to 70	70 to 150
PC*, network 3	6 to 10	50 to 120	150 to 200
Phone, Symbian Anna, CPU 680MHz, WLAN	16 to 20	220 to 300	500 to 700
Phone, Android 2.3, CPU 1.2GHz dual core, WLAN	4 to 5	30 to 50	70 to 80
Phone, Android 2.3, CPU 1.2GHz dual core, 3.5G/ HSPA	6 to 7	90 to 100	150 to 200
Tablet, Android 2.2, CPU 1GHz, WLAN	9 to 10	110 to 130	200 to 300
Theoretical optimum	4	40	80

* The same hardware, different network service providers and testing locations

The overall usability of the live view web page is satisfactory on larger screens like of PCs and tablet computers, and also quite feasible but not generally convenient for the ECGs waveform on screens of typical phones due to their modest dimensions (e.g. 4 inch). Figure 3 c shows three different screens displaying the live viewer's test page. However, browser technologies, including Javascript engines, as well as overall capabilities of mobile devices (phones, tablet computers) have been recently developed to fairly good levels to allow browser-based solutions, instead of custom (platform-dependent) applications, for live signal visualization, compared to the situation some years ago [19].

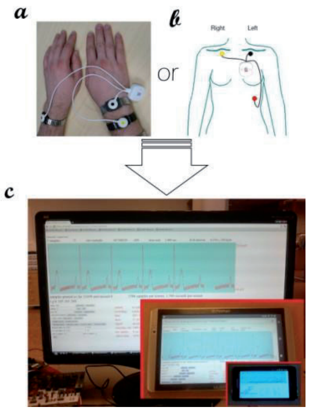

Fig. 2. MegaEMG ECG sensor a: emergency attachment using bracelets, b: conventional placement. c: Live EDF signal viewer (web page) on 22" PC monitor, 10" tablet computer, and 4" smartphone respectively.

V. CONCLUSIONS

We suppose that enabling telemedicine and wellness applications within eHealth systems may motivate healthcare enterprises and citizens to pay more attention to all kind of eHealth services. Currently eHealth solutions are EHR centric which makes integration and use of new applications complex or impossible. We propose streaming data upload solution based on widely accepted EDF and Google Protocol Buffers data standards. For simultaneous real time data access fully web based solution for clinicians or carers was developed and successfully benchmarked on desktop and mobile devices.

REFERENCES

[1] Health Information Network Europe (HINE), 2006 - European eHealth forecast (report)
[2] European Commission. Communication from the Commission on telemedicine for the benefit of patients, healthcare systems and society. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2008:0689:FIN:EN:PDF (accessed 09/11/11)
[3] e-Tervis SA at http://www.e-tervis.ee/ (accessed Nov 2011)
[4] Tiik M, Ross P. Patient opportunities in the Estonian Electronic Health Record System. Stud Health Technol. Inform. 2010;156:171-7. PMID: 20543352.
[5] Fontaine P, Ross SE, Zink T, Schilling LM. Systematic review of health information exchange in primary care practices. J Am Board Fam Med. 2010 Sep-Oct;23(5), pp 655-70
[6] http://www.e-health.standards.org.au (last accessed 10.10.2011)
[7] Omre, A.H.: "Reducing Healthcare costs with wireless technology",Bus. Dev., Nordic Semicond. ASA, Oslo, Norway., 2009, ISBN 978-0-7695-3644-6.
[8] Guang-Zhong Yang:" Body Sensor Networks", SpringerVerlag New York, Inc., Secaucus, NJ, USA., 2006, ISBN 978-1-84628-272-0
[9] Cleland J, Habib F. Assessment and diagnosis of heart failure (Minisymposium: Heart failure). J Intern Med 1996;239: pp 317–25, online 2003, DOI: 10.1046/j.1365-2796.1996.462801000.x.
[10] Chaudhry SI et al. Telemonitoring for patients with chronic heart failure: a systematic review. J Card Fail. 2007 Feb;13(1):56-62
[11] Kang K., Bae C., et al., UHaS: Ubiquitous Health-assistant System based on Wearable Biomedical Devices, IJIPM: International Journal of Information Processing and Management, Vol. 2, No. 2, pp. 114-26, 2011
[12] Standing S., Standing C., Mobile technology and healthcare: the adoption issues and systemic problems. Int J Electron Healthc. 2008;4(3-4): pp 221-35. PMID: 19174359
[13] McAlearney, A. S., Schweikhart, S. B., & Medow M. A. (2004). Doctors' experience with handheld computers in clinical practice: Qualitative study, British Medical Journal, 328(1162), 1-5.
[14] www.phrbox.com
[15] B. Kemp, A. Värri, et al., "A simple format for exchange of digitized polygraphic recordings" Electroencephalography and Clinical Neurophysiology, 82 (1992): 391-393.
[16] http://code.google.com/apis/protocolbuffers/docs/overview.html
[17] Reilent, E.; Lõõbas, I.; Pahtma, R.; Kuusik, A. Medical and Context Data Acquisition System for Patient Home Monitoring. In: The 12th Biennial Baltic Electronics Conference BEC2010, Tallinn, October 4-6, 2010. , 2010, (1; 1), 269 - 272.
[18] http://www.megaemg.com/
[19] Ros, M., D'Souza, M. and Postula, A. J. (2008). Wireless interactive system for patient healthcare monitoring using mobile computing devices. In: B. J. Wysocki, Signal Processing and Communication Systems 2008. 2nd International Conference on Signal Processing and Communication Systems, 2008 (ICSPCS 2008), Gold Coast , Australia, (1-6). 15-17 December 2008.

PAPER 7

I. Lõõbas, E. Reilent, A. Anier, A. Luberg, A. Kuusik. Towards semantic contextual content-centric assisted living solution. In: Proceedings of 12th IEEE International Conference on e-Health Networking Applications and Services (Healthcom 2010): 12th IEEE International Conference on e-Health Networking Applications and Services, Lyon 1-3 July 2010. IEEE Operations Center, 2010, (1; 1), 56 - 60.

Towards semantic contextual content-centric assisted living solution

Ivor Lõõbas, Enar Reilent, Andres Anier, Ago Luberg, Alar Kuusik
ELIKO Technology Competence Centre
Tallinn, Estonia
{ivor.loobas, enar.reilent, andres.anier, ago.luberg, alar.kuusik}@eliko.ee

Abstract—We present a content-centric software architecture of home monitoring solution designed to support universal semantically described context information and formal reasoning for automated profile generation and data aggregation. At the core of each data processing node in the system is a semantic RDF-based datastore to which a peripheral agent cloud is connected. An agent cloud is a collection of independent programs that are used to perform various tasks including sensor integration and semantic reasoning to derive new knowledge from existing semantic data. A way of communication between the agents is also described with a simplified data aggregation example. The presented architecture is a first iteration of assisted living research platform for further research.

Home care monitoring, semantic, context-aware, reasoning, data acquisition, data management.

I. Introduction

For the delivery of public health care, remote patient monitoring provides a cost-effective way to manage burdens on public services caused by an increasing portion of elderly and chronically ill people. Additionally, to reduce demand on clinics and home visits for doctors, the long term human monitoring is believed to be an effective way for early discovery of health risks. Increased acceptance and adoption of preventative care regimes within "well-being" programs is also important, for example smoking cessation and weight reduction are also requiring active patient monitoring at home.

Early patient home monitoring systems were designed only to acquire medical data to be analyzed by doctors off-line. Later ones were enhanced with automated warning generation and personalization features leading towards the increase of complexity in instrumentation and processing software. From one side, the existing telecare solutions are designed as classical data acquisition systems with static configurations of devices, patients and data processing algorithms. From the other side, which is more important, the context information which can be understood by the end-user might not be easily interpreted by machines. However, in order to address the issue of the rapidly increasing amount of collected data, it is crucial that context processing is automated.

In the present paper we describe a content centric architecture of home monitoring solutions designed to support semantically described universal context information and formal reasoning for automated profile generation and data aggregation. In the second chapter of the paper we take a deeper look at what has already been done in telecare R&D topics. In the third chapter we present our multi-level data acquisition architecture which makes use of a semantic RDF-based datastore concept we call whiteboard, and a peripheral agent cloud for data acquisition and processing. In the fourth chapter we describe semantic communication within the system and present a simplified data-aggregation example to illustrate a way of dealing with massive amounts of incoming raw sensor data within the system to derive higher-level information. Finally, in the last chapter we outline our plans for future developments.

II. Previous Work

As the population ages, telemedicine and home monitoring is an emerging topic of cost-efficient health care. As already well described by Doughty [1], the first generation telecare technology solutions enable patients to summon help in case of an emergency, the second generation provided automated detection of emergencies, and the third enabled monitoring the deterioration of well-being. The first generation solutions are available practically for the whole well-developed world. Commercial solutions of the second group, e.g. well@home in US [2], Zydacron [3] and Docobo [4] in Europe, have been around for 10 years as well. The third generation, also called lifestyle-monitoring by Barnes [5], essentially includes continuous monitoring of physical and social activities, also called Activities of Daily Living (ADL) by Chen [6], sleeping times, etc., for early discovery of health problems through indirect impact and providing contextual information for medical measurements. Some recent prototypes worth mentioning are by Amaral [7], Kaushik [8], and others.

As outlined in [6], the interpretation of ADL information has to be personalized making data processing quite a complex task. Typically measurement data pattern recognition algorithms are applied allowing one to later operate with predetermined (logical) states like health conditions and activities. Per Turner [9] "one size fits all" solution is unsuitable for patient monitoring as the system must be easily customizable by non-technical people, which apparently is not the case in real life.

Policy- (i.e. rule-)based home care systems are promising [9] in making it easier for end-users to modify the response behavior (e.g. set triggers) of a home care system. Rule-based home care has been investigated in various research projects [10, 11]. This approach is well in-line with Smart Home (SH) solutions because rule- and logic-, including Fuzzy logic-, based control is the leading control method in this area [12]. SH platforms are the leading technological solutions in providing cost-efficient assistance and monitoring for the elderly and disabled people. Logical reasoning is also the most natural method to derive additional knowledge expressed with ontological relations.

However, the SH environments are producing massive amounts of data from sensors and other devices around people and, until enriched with a well-defined meaning, the potential of SH-s assisting capabilities will not be fully achieved [6]. The main reason is complexity of reuse of acquired knowledge due to the lack of high-level uniform data representation. Apparently, the information and knowledge derived within a particular SH installation is very hard to copy and use at different locations because a) there are no widely accepted ontologies for presenting sensor-actuator data; and b) systems typically use low-level data formats, and conversion into higher level presentation and publishing is weakly motivated, which makes converting it into universal presentation a difficult task.

Chen's activities related to SemanticsAtHome [13] and other projects show that if SH data is semantically described and this semantic content is machine-readable then processing for analysis and decision support for intervention can be done more easily, and possibly with distributed computing power. Similarly, as shown by the Roboswarm project [14] deploying semantic web technologies for mobile (service robot) sensing for public indoor areas, significantly simplifies information reuse and service decomposition.

While there have been various proposed solutions to deal with issues of data structure, automatization and personalization arising in home care, not many attempts have been made to make use of many of them together. While not targeting medical home care, Xue and Pung [15] have addressed the issues presented above with their middleware solution and have even described a semantic P2P cluster overlay. Similarly addressing the presented issues and making heavy use of the experience gained in the Roboswarm project [14], we are applying the ideas deployed there to the medical home care system.

III. Architecture

Modern home monitoring solutions involve numerous amounts of various sensors that generate a lot of output information. To address the issue of massive incoming sensor data, we propose a hierarchical multi-level architecture where each node consists of similar components that follow similar processes while each node and component may have a different implementation. Through hierarchical multi-level architecture we intend to achieve low-level sensor data aggregation into high-level knowledge for end-user, and *vice versa* propagating high-level system management through the levels to low-level node-specific management, thus reducing the amount of raw data on higher levels. The system is intended to behave in a uniform way on all nodes and all levels. However, while sharing similar architecture, different implementations may specifically be tailored to suit various needs and possibilities of different computing platforms.

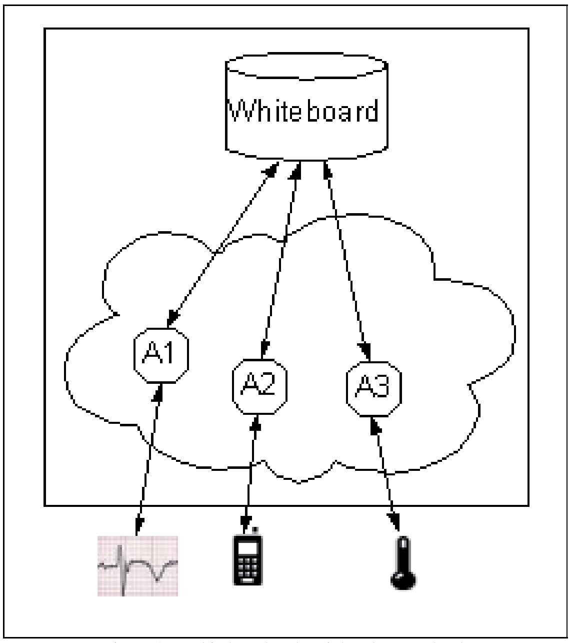

Figure 1. Whiteboard and peripheral agent cloud.

Each node consists of a whiteboard and a peripheral agent cloud. What we call an agent cloud is a collection of independent small programs that perform specific tasks on the given node. An agent cloud includes computational, communication, as well as wrapping agents for sensor integration. What we call a whiteboard is a concept that stands for a semantic datastore which is readable and writable by any agent at any given time. Whiteboard serves two objectives at the same time. Firstly, it is a persistent database for agents. Secondly, it is a communication medium for information exchange between the agents. Information exchange between the nodes on different levels is achieved through the specific communication agents which communicate information from whiteboard on one level to the whiteboard on another level. This makes a lower level node appear to a higher level node as yet another agent in the peripheral cloud. Similar whiteboard-centered approach with various agents for different tasks has been previously used in the Roboswarm project to make a mobile service robot operational. Various small agents were used for different tasks and the whiteboard was used as a central datastore and communication medium. As this architecture with central whiteboard has been successfully deployed in the Roboswarm project [14], we intend to build on this and make use of the previous experience.

In order to make our data understandable to other components in the system as well as outside of the system, we treat whiteboard as a semantic datastore. We make use of

Resource Description Framework (RDF) [16] which is based on making statements about resources in expressions in the form of subject-predicate-object called triples. Thus, our whiteboard datastore implementation can be viewed as a database having 3 columns: the subject, the predicate, and the object (or: the subject, the property, and the value). Which looks very similar to the N-Triples notation [18].

The subject of an RDF statement is used to identify resources. It can either be a Uniform Resource Identifier (URI) or a blank node, in which case it denotes an anonymous resource. To identify each node uniquely, a whiteboard within any node is assigned a URI which is stored on the whiteboard. Each agent running within the node has a unique identifier. Since our data representation is semantic, each agent is also assigned a unique URI which is formed by appending the agent's unique identifier to the URI of the whiteboard.

```
whiteboard URI: http://www.eliko.ee/ssg/wb/1

agent's unique ID: MyAgent1

resulting URI: http://www.eliko.ee/ssg/wb/1/MyAgent1
```

Figure 2. URI formation.

By similar approach we can also build URI-s to identify nodes (whiteboards) of the lower level, whose agents' URI-s in turn are formed by appending unique identifiers to the URI of the whiteboard. This enables us to identify any component within the system uniquely, which in our estimation should be beneficial for system management.

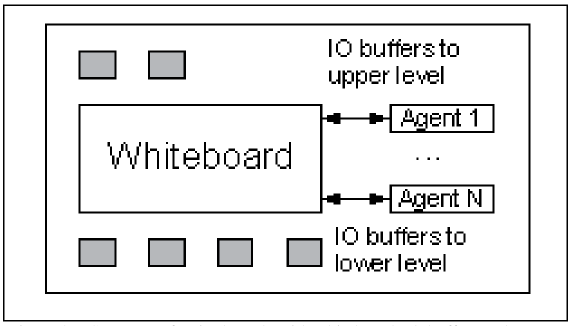

Figure 3. Structure of a single node with whiteboard, IO buffers and agents.

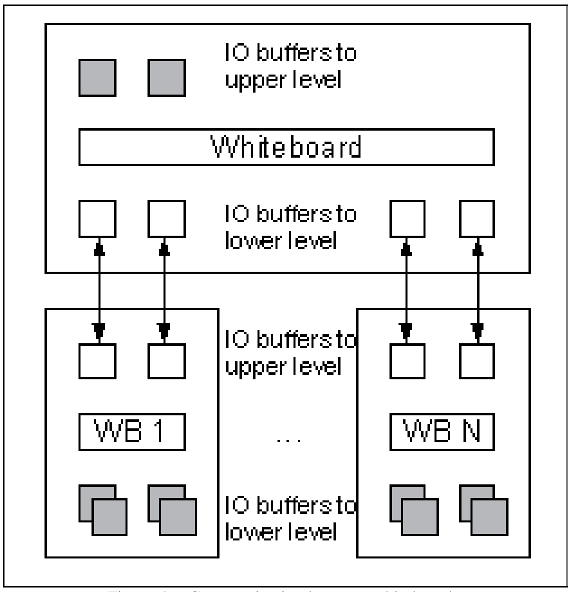
IO buffers to upper level
Whiteboard
IO buffers to lower level
IO buffers to upper level
WB 1
...
WB N
IO buffers to lower level

Figure 4. Communication between whiteboards.

Agents in our system can be divided roughly into two types by the way they acquire necessary data. There are agents that query data they need from the whiteboard, and there are consumer agents whose data is provided by the provider agents. Moreover, while reading from the whiteboard is open to all, writing to the whiteboard is a more complicated task if data collision and corruption is to be avoided. This all has lead us to introduce the concepts of input and output buffers which means that every agent has a dedicated areas for reading and writing which we currently identify through a RDF property.

```
@prefix e: <http://www.eliko.ee/ssg/schema> .

<http://www.eliko.ee/ssg/wb/1/MyAgent1>

        e:InputBuffer _:1 .

_:1 e:Property1 "value1" .

_:1 e:Property2 "value2" .
```

Figure 5. Input buffer example in Notation3 [17].

Output buffer is a similar concept. Everything written by any agent to the whiteboard is first stored in the respective agent's output buffer where it remains until it is picked up by another agent responsible for processing output buffers. This agent then decides where to move corresponding data.

Introduction of input and output buffers opens up a possibility of chaining agents together for data processing. For example if agent Agent1 outputs heart rate in ontology O1 while the system works with heart rate in ontology O, an additional agent Agent2 can be built to convert heart rate from ontology O1 into ontology O, thus making heart rate understandable to other agents in the system.

Semantic RDF-based representation of data makes it possible to apply predicate calculus and make use of rule-based reasoners (e.g. Jena) and programming in logic (e.g. Prolog) in the system for deduction of new knowledge. One way to include such reasoners is to write a specific agent for the reasoner that would mediate data from the whiteboard to the reasoner and *vice versa*. However, if reasoning on larger amounts of data is to be considered then this will prove to be ineffective. Because of this we also plan to introduce a reasoner within the whiteboard with either direct access to the whiteboard data or synchronizing changes made to whiteboard data to the reasoner database. This will result in data duplication but will allow the reasoner to have all the necessary data readily in memory when the task requiring reasoning is called.

IV. Communication and Data Aggregation

Whiteboards serve as a data exchange medium between the agents. For this purpose every whiteboard provides a communication interface. Depending on the whiteboard implementation in the given node the protocol, syntax and implementation of the interface may be different while the behavior remains the same. In addition, more than one implementation of the communication interface may be exposed. For example, communication between the nodes is based on web-services while communication within the node may be implemented using a whiteboard client library and direct function calls.

Communication is based on semantic data representation. Currently for handling simplicity, everything exchanged through communication interface is presented in N-Triples notation, or a notation similar to that. For example, a message sent from one node to another may be encoded in N-Triples notation enclosed in a SOAP message envelope, while a direct function call within the node may make use of an array of structs with separate fields for the subject, the predicate, and the object.

Any message exchange follows N-Triples notation where RDF blank nodes have only meaning within the message exchanged. In the example that follows a blank node identified by _:113 should not be considered as having meaning outside of the message. When whiteboard receives anything to be stored, all blank nodes are iterated through and replaced so that no data collision happens with data already stored on the whiteboard. Hence, _:113 may easily be replaced by _:217.

```
<http://www.eliko.ee/ssg/wb/1/MyAgent1>
<http://www.owl-
ontologies.com/nullontology.owl#Sample>_:113 .

    _:113
<http://bioinfo.icapture.ubc.ca/subversion/SIRS/clinicalphe
notype.owl#HeartRate> "70" .

    _:113
<http://bioinfo.icapture.ubc.ca/subversion/SIRS/clinicalphe
notype.owl#SaturationO2> "97" .
```

Figure 6. A sample message.

Whiteboard communication interface exposes three methods, two of them for reading, one for writing.

- list – a method for listing data stored on the whiteboard, as well as a method for searching by the subject, the predicate, or the object values. The search results return matches including sub-properties. In the example above, if search is made by the predicate <http://www.owl-ontologies.com/nullontology.owl#Sample> all of the triples above will be returned.
- listBuffer – a method for listing contents of the input buffer of the agent querying.
- storeBuffer – a method for storing data to the output buffer of the agent.

To deal with massive incoming amounts of sensor data, we intend to make use of multiple agents and multi-level architecture. What may initially be a computationally heavy task to achieve on a central node for numerous patients in home care, may become considerably simpler if raw sensor data is aggregated and analyzed on the home node, thus making use of the distributed computing power.

Let us consider a simplified example where a number of home care patients are requested to take Electrocardiogram (ECG) readings daily. In order to analyze long term heart rate variability trends to assess conditions of the cardiovascular system.

1. On a home node we have an ECG agent that takes raw samples coming from the ECG amplifier, enriches them semantically, e.g. with meta data like sensor maker, pre- and post-, low- and high-pass filters applied, pre- and post-processors, sampling rate, etc., and stores it on the local whiteboard.
2. A second agent on the home node picks up the stored ECG samples of the measurement, extracts heard beat period (RR interval) values and stores them on the whiteboard.
3. A third agent on the home node picks up the stored RR interval values of the measurement, calculates heart rate variability (HRV) value and stores it on the whiteboard.
4. A communication agent on the home node picks up the HRV value and sends it to the parent node.
5. On a parent node there can be one or more agents which analyze stored HRV values, calculate long-term trends, compare them with the trends from other patients under the observation while taking into account data not available on the lower level nodes, like information about the environment, activity history, medical history, etc.

While home nodes hold all the raw sensor data, there is no actual need for it on the higher level to calculate HRV trends and trend comparison. Thus, the actual sensor data remains on the lower level node for some predefined period. As long as the communication framework defines ways to retrieve this data when the need should arise, such scheme should ease the data load on central nodes and make use of distributed computing power.

V. Conclusions and Further Work

At present time we have implemented an assisted living research platform for further research. As outlined in this paper, we have used experience gained in the Roboswarm project and extended the Roboswarm architecture to fit our vision of the home care infrastructure.

We have created two implementations for different nodes, one representing a lower level node with sensors attached to it, the other a higher level node without sensors. On both nodes we have a whiteboard with input and output buffers and a number of agents. On the lower level node there are agents for specific sensors (e.g. PPG), communication agent that links the lower level node to the higher level node. On both nodes we have also implemented a reasoning agent, using Jena reasoner on the higher level node and Prolog on the lower level node, for deduction of new knowledge. Moreover, higher level node is implemented in Java and web-services as communication interface while lower level node is implemented in C and using direct function calls for communication with the whiteboard.

We have made use of the Roboswarm architecture experience and have confidence it will scale in a home care solution as well. Our further work involves refining current implementations, defining data that is needed for operation of the nodes, optimization of data storage, exchange and transfer, implementation of various agents for different sensors and computation tasks. Furthermore, it is necessary to design rule sets for automated profile generation and automated data aggregation.

Acknowledgment

This research has been supported by European Regional Development Fund.

References

[1] K. Doughty, K. Cameron, P. Garner, "Three generations of telecare of the elderly," Journal of Telemedicine and Telecare, vol. 2, no. 2, pp. 71–80, 1996.

[2] well@home, http://wellathome.com/.

[3] Zydacron, http://www.zydacron.com/.

[4] Docobo, http://www.docobo.co.uk/.

[5] N. M. Barnes, N. H. Edwards, D. A. D. Rose, & P. Garner, "Lifestyle monitoring – technology for supported independence," IEEE Computing & Control Engineering Journal, vol. 9, no. 4, pp. 169–174, August 1998.

[6] L. Chen, C. D. Nugent, M. D. Mulvenna, D. D. Finlay, X. Hong, "Semantic smart homes: towards knowledge rich assisted living environment," Special Issue on Studies in Computational Intelligence on Intelligent Patient Management (Edited by S. McClean), Springer, 2008.

[7] T. Amaral, N. Hine, and J. L. Arnott, "Integrating the single assessment process into a lifestyle-monitoring system," 3rd International Conference On Smart homes and health Telematic (ICOST 2005), pp. 42–49.

[8] Alka R. Kaushik, B. G. Celler, "Characterization of PIR detector for monitoring occupancy patterns and functional health status of elderly people living alone at home," Technology and Health Care, vol. 15, no. 4, pp. 273–288, December 2007.

[9] F. Wang, K. J. Turner, "Towards personalised home care systems," Proceedings of the 1st international conference on PErvasive Technologies Related to Assistive Environments, Article 44, 2008, ISBN:978-1-60558-067-8.

[10] J. C. Augusto, J. Liu, P. McCullagh, H. Wang, and J.-B. Yang, "Management of uncertainty and spatio-temporal aspects for monitoring and diagnosis in a Smart Home," International Journal of Computational Intelligence Systems, 1(4):361-378, Atlantis Press, 2008.

[11] K. Du "HYCARE: A hybrid context-aware reminding framework for elders with mild dementia," Smart Homes and Health Telematics, vol. 5120/2008, pp. 9–17, 2008.

[12] D. Cook and S. K. Das, "How Smart are our Environments? An Updated Look at the State of the Art," Journal of Pervasive and Mobile Computing, 2007.

[13] SemanticsAtHome, http://www.infc.ulst.ac.uk/cgi-bin/infdb/resprojview?projid=1275.

[14] T. Tammet, J. Vain, A. Puusepp, E. Reilent, A. Kuusik. "RFID-based communications for a self-organizing robot swarm," In S. Brueckner, P. Robertson, U. Bellur, eds., Proc. of 2nd IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems, SASO 2008 (Venice, Oct. 2008), pp. 45–54, IEEE CS Press, 2008.

[15] W. Xue, H. Pung, W. L. Ng, and T. Gu, "Data Management for context-aware computing", accepted for The 2008 IEEE/IFIP International Conference On Embedded and Ubiquitous Computing (EUC 2008), December 17–20, 2008 Shanghai, China.

[16] Resource Description Framework (RDF), http://www.w3.org/RDF/.

[17] Notation3, http://www.w3.org/DesignIssues/Notation3.html.

[18] N-Triples, http://www.w3.org/2001/sw/RDFCore/ntriples/.

PAPER 8

E. Reilent, A. Kuusik, I. Lõõbas, P. Ross, P. Improving the data compatibility of PHR and telecare solutions. In: 5th European Conference of the International Federation for Medical and Biological Engineering 14 - 18 September 2011, Budapest, Hungary: (Toim.) Jobbágy, Á.. Springer, 2011, (IFMBE Proceedings; 37), 925 - 928.

Improving the data compatibility of PHR and telecare solutions

E. Reilent[1], I. Lõõbas[1], A. Kuusik[1] and P. Ross[2, 3]

[1] ELIKO Technology Competence Centre, Tallinn, Estonia
[2] East-Tallinn Central Hospital, MD, Tallinn, Estonia
[3] Institute of Clinical Medicine, Tallinn University of Technology, Estonia

Abstract— **Personal Health Records (PHRs) and wellness telecare systems are emerging and shall enable individuals to take more responsibility of maintaining their own health. However, practical data interoperability with existing IHE systems has been not achieved yet due the wide range of different PHR information collected by users. We describe an HL7 v3 protocol based data interoperability solution that enables seamless integration of telecare data with an existing Estonian nation-wide Electronic Health Record System targeting nation-wide Universal Health Record (UHR) data repository.**

Keywords— **Telecare, universal health record, e-health**

I. Introduction

Personal Health Record (PHR) solutions are targeting safe, high quality and cost-efficient proactive healthcare. Health and wellness information collected by telecare systems should essentially be recorded into PHR for long-term monitoring and evaluation of trends of well-being. The most significant benefits of PHRs can be achieved through the interoperability of different existing patient data stores and IHE systems [1]. However, as stated by Lähteenmäki [2] most PHR systems do not communicate with other healthcare information systems well enough yet. Integration of telecare and subjective feeling data in meaningful and formalized way suitable for computerized processing is even more complex task. 2008 saw Google adapt Continuity of Care Record (CCR) for Google Health [3] thus showing high potential of the CCR to become leading standard for interoperable PHR systems. However, since CCR has been developed in accordance with the hospitals' requirements to Electronic Health Records (EHRs), there are certain limitations to include telecare and lifestyle information into CCR based PHRs.

As stressed in [4] there is significant lack of interoperability among health, rehabilitation and care information systems. Modern telecare systems essentially support lifestyle monitoring [5] and provide context information for measurement and event data. While such data carries important information about degradation of well-being [6] it shall be considered as essential part of PHR.

In some countries, for example in Estonia [7], national eHealth systems are developed for simple access to patient EHR data collected by any kind of healthcare organizations. Natural way is to enrich such healthcare databases with personal wellness and telecare content resulting true UHR repository. From one side, such long term infrastructure projects raise more deeply the issues of feasible interoperability of PHR, EHR and telemedicine systems with manageable data access right system. From the other side, due the increased demand for flexible data presentation standards leaded by telecare, wellness equipment manufacturers and end users, such interoperable UHR systems are more like to become available.

Current paper presents proposals how to integrate personal data from telemonitoring systems into Estonian nation-wide Electronic Health Record System – eTervis (eHealth) portal. For our analysis we rely on existing knowledge developing interoperable, semantics driven telemonitoring systems [8] to be used by East-Tallinn Central Hospital.

II. Proposed centralized repository for EHR and telecare data

For most of the countries there exist several competing and typically incompatible legacy repositories for EHR and PHR data. However, in some countries, for example in Estonia, exists a public, state controlled and centrally maintained infrastructure for citizens' EHR data. Such infrastructure offers (controlled) a single service to keep and access patient data. Since privacy and data retention issues for state driven data warehouses are carefully addressed and legally validated, such repositories are suitable for storing personal welfare information. Possible extension of centralized EHR system with telemonitoring capabilities is shown on Figure 1. Dashed lines present proposed data flows of telecare information.

Designed capable enough for storing high resolution image content and real time data access, the particular eTervis EHR repository can technically support telemonitoring services including recording narrow bandwidth (10kbps) streaming data, e.g. ECGs. Context enriched ECG monitoring can be considered as most demanding and modern *practical* telecare application for EHR/PHR systems to date [9].

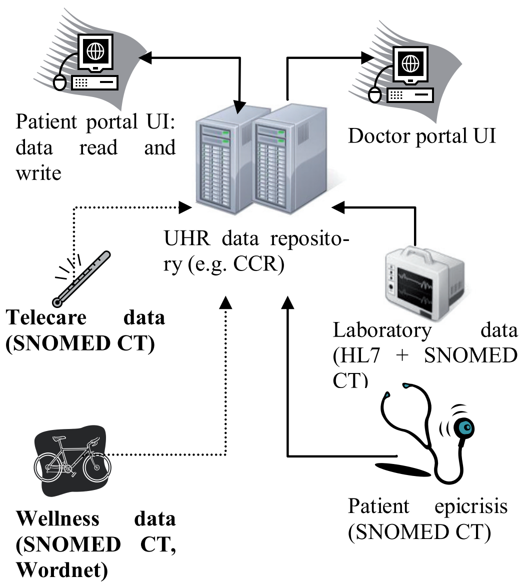
Patient portal UI: data read and write
Doctor portal UI
UHR data repository (e.g. CCR)
Telecare data (SNOMED CT)
Laboratory data (HL7 + SNOMED CT)
Wellness data (SNOMED CT, Wordnet)
Patient epicrisis (SNOMED CT)

Fig. 1 eHealth EHR system extended with telecare and wellness monitoring functionality

III. Requirements for expandable PHR system

Despite the existence of many PHR systems there is a growing interest to expand centralized EHR infrastructures by accepting and keeping data coming directly from the patients. The motivation to manage PHR data alongside with the clinical data collected by the healthcare professionals in the same system is to provide physicians with better access to their patients' nonclinical information. Because centralized EHR repositories are already responsible for keeping track of all medical data provided by different medical establishments, about one patient in the same place, they could as well concentrate private home-made measurements, training diary, drug taking history, etc.

However, if it is decided that PHR data has to be sent directly to EHR holder where the ordinary clinical data is, it cannot happen just out of the box. Typically, EHR systems have complex proprietary input protocols which do not suit well for PHRs. For example, eHealth has a large number of HL7 v3 XML message formats for storing and requesting predefined number of specific rather large and well-defined documents like medical case histories, ambulant cases, medical bills, referral letters, prescriptions, etc., while PHR data entries are usually small but vary from patient to patient.

The situation is similar with standalone PHR implementations in systems like described by [2], Google Health, Microsoft HealthVault [10], etc., which have actually very fixed formats for certain events, for example home-made blood pressure measurements. Even though many of them reference external vocabularies like SNOMED CT for creating interoperability on conceptual level, they fail to adapt great diversity of possible PHR entries beyond the given formats of a particular coding system.

The goal is to find the practical way to represent data gathered by the patient. The focus is to add expandability and refine-ability where fine-tuned formats and schemas lack certain flexibility and need considerable intervention when new data types are introduced or old ones require expansion. For example, when a patient uploads weight measurement data it is important in some cases to add meta-information, like the type of scale used, but if the current schema has no slot for this kind of information, there is no seamless solution to enable recording of such data. On the other hand, we cannot force every entry to have the device field. Therefore, some data organization methods will not fit the best. We must also not forget that data still has to be automatically manageable by expert systems and rule engines which are usually present in EHR systems (e.g. decision support algorithms Map of Medicine [11] in eHealth).

In the realm of PHR there are many different behavior models of users, some want to keep diary of physical activities, others track their health parameters like blood pressure daily, or at random times. In general, patients want to send not very precisely defined wellness data as well as results of medical measurements to their PHR. While clinical data is unanimous, PHR data can be medical as well as describing subjective wellness feeling, or data not even directly connected to the patient but rather contextual, like outdoor temperature at jogging time, etc.

As an example of volatile nature of PHR data schema we could consider the case of keeping workout diary. While one patient just logs the history of exercises taken at the gym with duration data, the other patient wants to enrich her training diary with data output by the pulse watch. Suppose the PHR repository is capable of accepting all that but if the person wants to add some conclusion of how she felt after the training session, this might not be possible to accomplish in the system. Additionally, in some cases person would like to add aggregated data of particular measurements to the PHR.

IV. Data Model

Hereby the underlying core data structure has rather important role for effective connectivity of versatile data sources and platforms for creating expandable PHR ecosystem. Tightly fixed formats of recordable events can be relied on to some extent, but rather general and universal basic data schema could perform better for the patient's point of view even if it poses inconveniences for technical side.

PHR builds up in a natural way as a diary of entries which are typically rather short and based on a single measurement, test, activity etc. In most of the cases complex HL7 v3 messages with lots of required field as regular in the EHR system eHealth can be avoided but still easily introducible if necessary. There are not many fields in PHR entries that have to be present in every case, perhaps patient identification data and the timestamp of the entry are most common, still not omnipresent as the patient ID can be transmitted outside of the contents of the entry (already needed during the creation of communication channel and authentication) and the timestamp might be omitted with some static personal data (name, genetic information, disabilities). Therefore no compulsory fields are required on the general level. Also grouping entries to components or dividing them to certain number of subtypes is dropped.

According to the proposed data model a PHR entry shall be a list of *key = value* pairs, which is actually covering lots of possible cases of usage. The *key* is always terminal node and refers to an external ontology or vocabulary concept (SNOMED CT, LOINC, WordNet, etc). As it is publicly accepted to use many coding systems simultaneously for enabling semantic interoperability all identifiers split to root and extension (HL7, Google Health, [2]). For example the key *weight* in pseudo code takes the form of *<key extension = "363809009" root = "2.16.840.1.113883.6.96" codeingSystemName = "SNOMED CT" displayName = "Weight">* in an arbitrary XML representation.

The *value* node can, however, be either a single terminal (string, number) as *42* or a list of elements as *[element1, element2, enementn]* where a list element can be terminal node or another *key = value* pair, thus making the schema hierarchical. Still, the encoding is more intuitive than the core model of entity-role-participation-act from the HL7 v3 and compared to the *<TestResult>* node from Google Health it can express list of values, for example recorded data stream of a handheld ECG device (CardGuard SelfCheck ECG [12]) like *1_lead_ECG=[6918, 10246, 10246, 9734, …]*.

For better illustration we could consider a minimalistic example of an entry of how a patient could evaluate her current feeling in the following pseudo code:

```
entry = [
    feeling = tired,
    timestamp = 14 Feb 2011 17:52
]
```

Another entry represents the usage of nested lists of *key = value* pairs and depicts the results of one blood pressure measurement. It is worth of noticing that as the measurement device outputs also the pulse rate reading besides the blood pressure, it is very natural and simple to encode that data into the same entry:

```
entry = [
    type = measurement,
    blood pressure = [
         systolic = 120,
         diastolic = 80,
         unit = mmHg
    ],
    pulse = [67, unit = ppm],
    timestamp = ...
    user comment = ...
    device = ...
    ...
]
```

For considering the case of enriching data the following entry demonstrates a log of a training event where the patient got the data from a pulse watch:

```
entry = [
    training = [burned calories = 570],
    device = [polar watch, model = 1]
]
```

But if the same patient uses different model of pulse watch later which outputs more parameters, the data stream of that patient could be enriched with backward compatibility and corresponding user applications adopt the new data schema with reasonable effort:

```
entry = [
  training = [
    burned calories = 570,
    duration = [120, unit = minutes],
    sport = [cycling, avg speed=25 km/h]
  ],
  device = [suunto watch, model = 2]
]
```

V. Proposed enhancements to eHealth ecosystems

For making use of the potential of eHealth system to accept PHR type data coming directly from patients either by inserted manually in dedicated web-applications or automatically from telemonitoring installations which connect (e.g. using Blutooth) to physical sensor devices and process incoming sensor data, some additional development is needed to be done on the EHR's side.

As soon as the concrete transferring carrier is picked for the proposed hierarchical entry format and authentication procedures are agreed on there are no major obstacles on uploading and saving the PHR data to eHealth. In general, access interfaces to the system do not have to be differently designed from the existing web-services used for managing ordinary clinical EHR data encoded in HL7 v3 XML messages.

The PHR entries could be expressed in XML and parsed into the tree of objects for accustomed processing and managing on the server side or be encoded into RDF triples and handled respectively. While precise storage or querying issues and displaying PHR data streams on the user interface are not in the scope of this paper, experiments show that it is feasible to encode the data given in the proposed schema into ordinary relational databases as well as into dedicated XML databases (e.g. [13]) for hierarchical records.

VI. Conclusions

PHR systems are quickly winning popularity among the competitive healthcare users. On the same time, vogue of different fitness and training data exchange services is increasing even more rapidly. It has been expected a breakthrough in telemedicine utilization to manage the aging Western populations. There are some attempts, even in large scale – for example eTervis (eHealth) in Estonia - to merge all such information into one public UHR cyberspace. However, present PHR data encoding standards developed from hospitals' perspectives have insufficient flexibility to present quite loose and context dependent telecare or lifestyle information. We propose to extend existing CCR, CCD and proprietary PHR data formats with refinement potentiality for storing specific detailed records. Proposed data encoding solution was successfully tested with cardiological telecare data and context information to be fitted into strictly predefined schema of existing eTervis EHR system.

Acknowledgment

This research has been supported by European Regional Development Fund, Competence Centre program of Enterprise of Estonia.

References

1. Kaelber D, Pan E C, et al. (2008) The Value of Personal Health Record (PHR) Systems, AMIA Annu Symp Proc. 2008: 343–347, PMCID: PMC2655982
2. Lahteenmaki J, Leppanen J, Kaijanranta H (2009) Interoperability of personal health records. in Conf Proc IEEE Eng Med Biol Soc. 2009:1726-1729 PMID: 19964259
3. Google at (2011) http://code.google.com/intl/et-EE/apis/health/ccrg_reference.html
4. AALIANCE Ambient Assisted Living Roadmap Ger van den Broek et al. (Eds.) IOS Press, 2010 doi:10.3233/978-1-60750-499-3-62
5. 6 Nugent C D, Mulvenna MD et al. (2008), Semantic Smart Homes: Towards Knowledge Rich Assisted Living Environment, Special Issue on Studies in Computational Intelligence on Intelligent Patient Management (Edited by McClean S.), Springer, Heidelberg
6. 7 Amaral T, Hine N, Arnott J L (2005), Integrating the Single Assessment Process into a lifestyle-monitoring system. In 3rd International Conference On Smart homes and health Telematic ICOST 2005, pp 42–49
7. e-Tervis SA at http://www.e-tervis.ee/
8. Kuusik A, Reilent E, Lõõbas I, Parve M (2010) Software architecture for modern telehome care systems; in proc 6th International Conference on Networked Computing INC 2010, ISBN 978-89-88678-20-6, IEEE Catalogue number CFP1084J-ART, pp 326-331
9. Schmidt S, Schuchert A, Krieg T, Oeff M (2010), Home Telemonitoring in Patients With Chronic Heart Failure, Deutsches Ärzteblatt International, 107(8), pp. 131-138
10. Microsoft HealthVault at http://www.healthvault.com/personal/index.aspx
11. Map of Medicine at http://www.mapofmedicine.com/
12. PMP4 Self Check ECG at http://www.lifewatch.com/telehealth_monitors
13. eXist at http://exist.sourceforge.net/

Use macro [author address] to enter the address of the corresponding author:

Author:	Enar Reilent
Institute:	ELIKO Technology Competence Centre
Street:	Teaduspargi 6/2
City:	Tallinn
Country:	Estonia
Email:	enar.reilent@eliko.ee

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. **Vineeth Govind**. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. **Pavel Grigorenko**. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. **Anna Rannaste**. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. **Sergei Strik**. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. **Rain Ottis**. A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.

61. **Natalja Sleptšuk**. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. **Martin Jaanus**. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. **Argo Kasemaa**. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. **Kenneth Geers**. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. **Riina Maigre**. Composition of Web Services on Large Service Models. 2011.

66. **Helena Kruus**. Optimization of Built-in Self-Test in Digital Systems. 2011.

67. **Gunnar Piho**. Archetypes Based Techniques for Development of Domains, Requirements and Sofware. 2011.

68. **Juri Gavšin**. Intrinsic Robot Safety Through Reversibility of Actions. 2011.

69. **Dmitri Mihhailov**. Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models. 2012.

70. **Anton Tšertov**. System Modeling for Processor-Centric Test Automation. 2012.

71. **Sergei Kostin**. Self-Diagnosis in Digital Systems. 2012.

72. **Mihkel Tagel**. System-Level Design of Timing-Sensitive Network-on-Chip Based Dependable Systems. 2012.

73. **Juri Belikov**. Polynomial Methods for Nonlinear Control Systems. 2012.

74. **Kristina Vassiljeva**. Restricted Connectivity Neural Networks based Identification for Control. 2012.

75. **Tarmo Robal**. Towards Adaptive Web – Analysing and Recommending Web Users` Behaviour. 2012.

76. **Anton Karputkin**. Formal Verification and Error Correction on High-Level Decision Diagrams. 2012.

77. **Vadim Kimlaychuk**. Simulations in Multi-Agent Communication System. 2012.

78. **Taavi Viilukas**. Constraints Solving Based Hierarchical Test Generation for Synchronous Sequential Circuits. 2012.

79. **Marko kääramees**. A Symbolic Approach to Model-based Online Testing. 2012.