Rapid Prototyping Infrastructure for Wearable Computing Applications

A dissertation submitted to
Faculty 3 (Mathematics and Computer Science) at the
University of Bremen
for the degree of
Dr.-Ing.

by
Dipl.-Inf. Hendrik Iben
born January 2nd, 1981, Bremerhaven

Vorgelegt am / Date of submission: 22.04.2014
Datum des Promotionskolloquiums: 19.06.2015
/ Date of defense

1. Gutachter / Examiner: Prof. Dr. Michael Lawo
2. Gutachter / Co-Examiner: Prof. Dr. Paul Lukowicz

Publication Changes

The following changes have been made to this dissertation for publication:

- Appendix D that contained previously published papers has been removed.
- All figures have been converted to grayscale.
- Some table headers in the evaluation chapter have been edited and/or annotated for a better understanding.
- Various spelling mistakes were fixed.

Änderungen zur Publikation

Zur Publikation wurden an dieser Dissertation folgende Änderungen vorgenommen:

- Anhang D, welcher bereits publizierte wissenschaftliche Veröffentlichungen enthielt, wurde entfernt.
- Alle Abbildungen wurden in Graustufen umgewandelt.
- Einige Tabellenköpfe im Evaluationskapitel wurden zum besseren Verständnis bearbeitet und/oder annotiert.
- Diverse Rechtschreibfehler wurden korrigiert.

Acknowledgements

I want to thank my academic supervisor Prof. Dr. Michael Lawo for his support especially during the final phase of my thesis work. I also thank Prof. Dr. Malaka and Prof. Dr. Lukowicz for their willingness to examine my work.
In addition I also want to express my gratitude to Prof. Dr. Otthein Herzog who helped me in many discussions to find the right direction when starting my thesis work.
I also want to thank my former colleague Dr. Hannes Baumann who worked with me on the SiWear project. Our common goals in the project helped to drive the progress of our academic work.
Working on the SiWear project brought up many opportunities for fruitful discussion with people from many different institutions. I would like to thank everyone involved for their contributions.
I want to thank all past and current members of the Wearable Computing Group at the TZI for creating a nice working atmosphere where an interesting discussion can occur by just walking up to the coffee machine.
Finally I want to thank my parents and my two sisters who always encouraged me to pursuit my academic career even though it was not always easy to share my fascination for this field with them.

Contents

Publication Changes **iii**

Änderungen zur Publikation **iii**

Acknowledgements **v**

Abstract **xi**

Zusammenfassung **xiii**

1 Introduction **1**

2 Building a Wearable Computing Application **5**

2.1 Wearable Computing Context Needs 5

2.1.1 The Need for Simple Systems 6

2.1.2 Mock-Up Testing and Iterative Development 7

2.1.3 Scientific Study Support 9

3 Related Work **11**

3.1 Roulette Wheel Prediction 12

3.2 Implicit Human Computer Interaction 13

3.3 Active Maps 13

3.4 MIThril Enchantment Whiteboard 14

3.5 wearIT@work 16

3.6 Context Recognition Network Toolbox 18

3.7 Evaluation of Findings 18

4 Evaluation of the Context ToolKit **23**
4.1 Context ToolKit . . . 23
4.1.1 Design Concept . . . 24
4.1.2 Architecture . . . 25
4.1.3 Context Information Model . . . 27
4.1.4 Context Information Delivery . . . 27
4.1.5 Usage in Applications . . . 28
4.1.6 Code Metrics . . . 29
4.1.7 Discussion . . . 33

5 The TZI Context Framework **37**
5.1 Motivation . . . 38
5.2 Distributed Communication Schemes . . . 38
5.2.1 D-Bus . . . 39
5.2.2 IRC . . . 39
5.3 Describing Context in an Abstract Way . . . 40
5.4 General Context Handling . . . 43
5.4.1 Infrastructure for Context Distribution . . . 45
5.4.2 Data Structures for Context Information . . . 47
5.4.3 Differences between TCF and CTK . . . 53
5.4.4 Code Metrics . . . 54
5.4.5 Communication Protocol . . . 59
5.4.6 Short Context Mode . . . 66
5.4.7 Transfer of large datasets . . . 67
5.4.8 History Queries . . . 69
5.4.9 Storing Context Histories in Relational Databases . . . 71
5.4.10 Scripted Subscription . . . 74

6 Evaluation of the TZI Context Framework **77**
6.1 Comparing CTK to TCF . . . 77
6.1.1 Application Complexity . . . 78
6.1.2 Transmission Performance and Efficiency . . . 83
6.1.3 Conclusion . . . 87
6.2 Creating High-Level Context . . . 88
6.2.1 Context Aggregation . . . 89
6.2.2 High-Level Example . . . 89

6.2.3 Limitations . . . 90
6.3 Framework usage in the SiWear Project . . . 91
6.3.1 Context for Picking . . . 92
6.3.2 Test-Scenario . . . 94
6.3.3 Evaluation-Scenario . . . 96

7 Conclusion and Outlook **99**

References **103**

List of Figures **109**

List of Abbreviations **111**

A Server Commands **113**
A.1 Connection Maintenance Commands . . . 113
A.2 General Queries . . . 114
A.3 Context Manupilation Commands . . . 117
A.4 Subscription Management . . . 117
A.5 Context Setting . . . 119
A.6 Context History . . . 119
A.7 Large Transfers . . . 120

B UML Diagrams **121**

C Code Examples **127**
C.1 Context Subscription and Processing . . . 127
C.2 Context Generation . . . 128
C.3 Temperature Demo CTK . . . 128
C.3.1 CTKTempReader.java . . . 129
C.3.2 TempReadUI.java . . . 135
C.3.3 CTKTempProvider.java . . . 138
C.3.4 TempUI.java . . . 141
C.4 Temperature Demo TCF . . . 143
C.4.1 CTXTempReader.java . . . 143
C.4.2 TempReadUI.java . . . 147
C.4.3 CTXTempProvider.java . . . 149
C.4.4 TempUI.java . . . 151

Abstract

Supporting workers by the use of computer systems is widely found throughout the world of work. While the benefits are apparent for office work, process monitoring and similar fields, where computers are used as tools to carry out a task, other fields can not benefit of them at first sight.

When a task demands high mobility or the use of hands, computers cannot directly make it easier. The manual work is in focus here and cannot be carried out by the computer as a tool. In these fields workers can be supported by the use of *wearable computing* applications where using the computer is not the focus but the automatic detection of on-goings in the environment.

These *context-sensitive* applications consist of a body-worn system - the *wearable* - together with numerous sensors distributed in the environment. By analyzing sensor-data in conjunction with known work structures the system is able to support the wearer at his or her task. Support can come up from displaying preprocessed information relevant to the task or by triggering processes in the environment.

Typical wearable computing applications are found in an industrial environment. Possible examples are order picking, car-manufacturing or maintenance work. The challenge when developing a supporting application here is to improve an already optimized process. When designing a wearable computing application the user, whose present task has to be structured more efficient, is in focus.

For the development of these applications user studies are vital where a possible solution can be evaluated in a practical environment. For a wearable computing application this often means to test a sensor-system resp. a possible way of detecting specific events in spite of not knowing in advance if it will be beneficial. These kind of tests are often performed with simulated systems in so called *Wizard-of-OZ* studies. Here a human operator takes over the role of the component in question and sends corresponding signals to the rest of the system. If a simulated system turns out to be unusable no resources have been consumed for a technical realisation. If results are promising the component is implemented and a next test is performed with the real system.

When using sensors in the environment this approach demands that the application needs to work with simulated and later with real values. For this a generic framework is desirable that enables an easy replacement of components. While there exist generic methods in software engineering to allow replacing components they do not incorporate the technical needs of sensors.

This thesis deals with the implementation of a framework to build context-sensitive wearable computing applications that enables a rapid-prototyping approach during development. Special technical demands for such a framework in an industrial setting are worked out and an abstraction for modelling information in the environment is introduced. An existing framework for context-aware applications in a *pervasive computing* environment is examined where problems arise when transferring it to the wearable computing domain. This motivates a specialized approach customized to the conditions in an industrial environment. As a last point an evaluation of the framework by its use in the development of a wearable computing solution is performed leading to a final assessment of its value.

wearable computing, mobile computing, pervasive computing, rapid prototyping, software infrastructure

Zusammenfassung

Die Unterstützung von Menschen durch Computersysteme ist weit verbreitet in der Arbeitswelt. Während die Vorteile für Bürotätigkeiten, Prozessüberwachung oder ähnliche Arbeitsfelder, in denen Computer als Werkzeug für die Umsetzung einer Tätigkeit verwendet werden, offensichtlich sind, besteht in anderen Bereichen auf den ersten Blick keine Verwendungsmöglichkeit.

Tätigkeiten, die eine hohe Mobilität oder den Einsatz der Hände erfordern, können nicht direkt durch ein Computersystem erleichtert werden. Hier steht die körperliche Arbeit im Vordergrund, die nicht durch den Computer als Werkzeug erledigt werden kann. In diesen Arbeitsfeldern kann eine Unterstützung durch den Einsatz von *Wearable Computing* Anwendungen realisiert werden, bei denen der Einsatz des Rechnerssystems nicht im Vordergrund steht, sondern die automatische Erkennung von Vorgängen in der Umgebung.

Diese *kontext-sensitiven* Anwendungen bestehen aus einem am Körper getragenen System - dem *Wearable* - sowie aus einer Vielzahl von Sensoren, die in der Umgebung verteilt sind. Durch die Analyse von Sensordaten in Verbindung mit den bekannten Arbeitsstrukturen, kann das System den Träger bei seiner oder ihrer Arbeit unterstützen. Eine Unterstützung kann dabei die Aufbereitung und Darstellung arbeitsrelevanter Informationen sein oder auch das Auslösen von Prozessen im Umfeld.

Das typische Einsatzfeld für Wearable Computing Anwendungen findet man im industriellen Umfeld. Beispiele sind die Kommissionierung, die Fertigung im Automobilbereich oder Wartungstätigkeiten. Die Herausforderung bei der Entwicklung einer unterstützenden Anwendung ist es hier, einen bereits optimierten Prozess zu verbessern. Beim Design von Wearable Computing Anwendungen steht daher der Nutzer im Vordergrund, dessen bisherige Tätigkeit effizienter gestaltet werden soll.

Bei der Entwicklung dieser Anwendungen sind Nutzerstudien unverzichtbar, in denen eine mögliche Lösung in einer praxisnahen Umgebung evaluiert werden kann. Für eine Wearable Computing Anwendung bedeutet dies häufig, ein Sensorsystem bzw. eine mögliche Art der Erkennung von Ereignissen testen zu müssen, obwohl ein möglicher Nutzen nicht von vornherein sicher ist. Derartige

Tests werden häufig mit simulierten Systemen in sog. *Wizard-of-OZ* Studien durchgeführt. Dabei übernimmt ein Mensch die Rolle der zu testenden Komponente und gibt entsprechende Signale an das restliche System weiter. Stellt sich ein simuliertes System als nicht brauchbar heraus, wurden keine Ressourcen durch eine technische Realisierung verbraucht. Sind die Resultate vielversprechend, wird die Komponenten technisch umgesetzt und ein nächster Versuch mit dem echten System durchgeführt.

Bei der Verwendung von Umgebungssensoren bedeutet dies, dass die kontextsensitive Anwendung einmal mit simulierten und später mit echten Werten funktionieren muss. Hierzu ist ein generisches Framework wünschenswert, welches eine einfache Austauschbarkeit ermöglicht. Es existieren zwar allgemeine Methoden in der Softwareentwicklung, um Austauschbarkeit von Komponenten zu ermöglichen, jedoch beziehen diese die technischen Anforderungen von Sensoren nicht mit ein.

Die vorliegende Arbeit beschäftigt sich mit der Implementierung eines Frameworks für kontext-sensitive Wearable Computing Anwendungen, durch das ein *Rapid-Prototyping*-Ansatz bei der Entwicklung ermöglicht wird. Die speziellen technischen Anforderungen an ein solches Framework im industriellen Umfeld werden heraus gearbeitet und eine Abstraktion zur Modellierung von Umgebungsinformationen wird eingeführt. Ein bestehendes Framework für kontextsensitive Anwendungen in *Pervasive Computing* Umgebungen wird betrachtet wobei Probleme bei der Übertragung auf die Wearable Computing Domäne sichtbar werden. Dies motiviert einen spezialisierten Ansatz der an die Bedingungen eines industriellen Umfeldes angepasst ist. Als letzter Punkt, wird eine Evaluation des Frameworks durch seinen Einsatz bei der Entwicklung einer Wearable Computing Lösung durchgeführt und eine abschliessende Bewertung erstellt.

Wearable Computing, Mobile Computing, Pervasive Computing, Rapid Prototyping, Software Infrastruktur

Chapter 1

Introduction

Wearable Computing describes body worn computing devices that provide useful information relevant to a physical real world task. Unlike mobile devices the focus is on the task and the device should not unnecessarily distract from it. To fulfill this expectation the wearable device cannot depend on direct interaction with the wearer but needs to process information from the environment to infer the next helpful action to be taken.

This relevant information is the *context* of a wearable computing application. There are many sources for contextual information: One can use sensors that measure physical properties like location, lighting conditions, the presence of digital markers to name a few. Other possibilities are known aspects of the work-flow and the resulting mental model of the user to take pro-active actions, software generated information from databases, current time, simulations or implicit interaction by analyzing the actions of the wearer or co-workers.

One problem with the wearable computing approach is the ambiguity of information gained through any system and the conversion into usable context-information. This problem is an open research question and many solutions for limited aspects have been proposed (e.g. [Dey00, LTGLH07, ST94]).

Wearable Computing is a field that shares many challenges with Pervasive Computing but also has its own problems to solve and other constraints.

It is commonly applied in industrial settings to support workers at a specific task. Additional challenges arise from the industrial setting as typically devices have to be running throughout the whole shift of a worker and need to be connected to some form of information system that keeps track of the state of the task at hand.

Even though many advances have been made to reduce energy usage and increase battery capacity, a full shift of eight hours is still a challenge that can be approached by designing a system for reduced energy usage. Unlike a Smartphone, that can save energy by turning off components until needed, a wearable computing device is virtually always on to support the user. Any software that runs on such a device needs to make a best effort at saving resources that consume a significant amount of energy - notably CPU time and wireless network transmissions.

When it comes to designing systems that can make use of information from the environment, pervasive and wearable computing approaches face a common problem. Information needs to be transmitted from devices in the environment to the application. Currently two approaches are found. The first approach is creating an application from scratch, e.g. selecting the needed information providers and designing the application to directly use these sources. While this allows to create very efficient applications, making changes at a later stage becomes problematic or impossible. In the second approach, a framework is used as a layer between the application and information from the environment that allows the application to retrieve information in an abstract way. While this approach obviously introduces an overhead, it makes introducing changes later easier and also enables quicker development since a transport scheme is already in place.

However, even though the benefits of having an abstraction layer are a valid concern, many applications are still developed from scratch and not many of these abstraction layers have been created. A notable implementation of such a layer is the Context ToolKit (CTK) [SDA99] that provides a complete framework to make environmental information available to applications. It has been

successfully used to create smart environments and is based on extensive research on best ways to provide information to smart applications.

It is however hard to apply the CTK to a wearable computing system due to technical problems and design decisions. CTK was designed for smart environments where all components are accessible over the same network. This allows for an elegant discovery mechanism but can typically not be applied to restricted networks found in industry settings. Additionally, it is not applicable to current even more restricted mobile networks.
Even though these technical problems are severe, they could be solved by stripping some of the elegance from the system. The problem with the design however remains. The design of the CTK is decentralized making each component of the system independent of each other. This also means that each component is responsible for communicating with other components that need information from the device. If this is applied to a small sensor in a wearable computing application (e.g. a temperature sensor), the computation needed to be part of the CTK framework is very high compared to the actual computation power needed to process the sensor.

A wearable computing scenario therefore needs a more fitting framework that cares equally for the information providers, such as sensors, and information consumers. While the consumers will typically be more sophisticated systems their energy is still limited if they are wearable. The producers might be at a fixed place in the environment and therefore have access to virtually unlimited energy but might also be a part of a mobile system or a component of the wearable system that additionally drains energy. In any case, a low computing and transmission power demand is desirable.

On the other hand, information distribution from a producer to potentially many consumers is a key factor of a framework for smart applications. This also applies to the ability to locate suitable producers at the runtime of the application.

To provide these features, a wearable computing information framework has to shift needed computing power away from the individual devices and concen-

trate it on one dedicated system. While the CTK uses a distributed approach to create an information distribution framework for pervasive computing, wearable computing needs a centralized approach. A single, powerful computer in the environment will serve all other components. Information producers send data only to this server while consumers are notified of this data by it. This approach minimizes computational and transmission effort for the components by putting this burden on a single system. In addition to this, a centralized approach is also more easy to integrate into a typical industrial network as only the server has to be reachable in a controlled fashion.

In order to show the beneficial character of a centralized system for wearable computing, it has to be compared to the other approach. At the time of this writing the CTK is the only usable system for creating a smart application with a framework. As no centralized approach exists, it will be implemented as part of this work. The implementation will show the concept of centralized information distribution and how applications can make use of it. To show the fitness for the wearable computing domain, an examplary application will be created using the framework. To further show, that the decentralized system saves energy and resources for the clients, a technical comparison will be done where both frameworks perform the same task. By analyzing the needed transmission effort a direct relation to the energy needed by both systems can be derived.

Chapter 2

Building a Wearable Computing Application

The motivation for the creation of a new framework for contextual support in wearable computing applications emerges from the experiences during the design of such an application. Evaluating existing frameworks in the light of the task at hand showed a missing link between modelling contextual information and the technical aspects of making this information available to wearable clients. This chapter provides an overview of the encountered problems during the design and how they influenced the creation of a practical approach to context information.

2.1 Wearable Computing Context Needs

In the BMWi-funded project SiWear[1][BGH+07], the consortium was tasked with the creation of a wearable computing solution for applications in picking and service for the automobile industry. The goal was to create a solid foundation for wearable computing applications suitable for the needs of this field by evaluating and extending the state of the art in wearable computing.

During the course of this project many studies were conducted in laboratory and also real world environments to evaluate the effects and potential of supporting workers with wearable computing technology. While the initial studies did not

[1] http://www.siwear.de

employ very sophisticated systems, they were driving the technical work in terms of requirements and software architecture design. During this phase, previous work in wearable computing, especially the results of the wearIT@work[LHW07] project, where examined for potential use in the project. While many of these previous approaches had been designed to be reusable in later projects, with the intention of speeding up development, many technical problems were encountered that ultimately lead to the development of new solutions for the task at hand.

Existing systems were developed to suit a particular set of requirements that may be different from the specific requirements of a new project. When this is the case, the existing system was not generic enough to be truly reusable. On the other hand, a truly generic system may be too complex for starting a new project with it as developers may not even know what they need at first. Both situations lead to the infamous *not-invented-here-syndrome*[KA82], where developers neglect existing software in favour of a solution tailored to their exact needs.

2.1.1 The Need for Simple Systems

As stated before, re-inventing software solutions is often the result of existing software being to complex to use quickly. Also, not knowing exact requirements during initial prototype development can make it impossible to select an appropriate solution from the beginning. When development started in the SiWear project a mix of both situations was in effect. While some systems for creating wearable computing applications were known, initial investigations showed that it was not clear how to use them correctly while they also were very complex systems that required a significant amount of time to set up. Furthermore it was unclear, what kind of information had to be processed and how the complete system would need to be integrated into an existing infrastructure.

To create applications more quickly it was decided to start with the creation of user interface tests that needed to respond to direct input via common user interface elements. Additionally, external events from sensors or other sources should be able to modify the state of the interface. The only known constant for development was that external events would be sent via a network connec-

tion. Using these requirements and constraints a very rudimentary client-server approach was built where a wearable client would retrieve information from the environment via textual messages from a server in the environment. The interpretation of these messages was done by the client itself, providing a first step in the creation of a content agnostic server for context information.
Even though this system was very limited and non-structured, it provided developers with a common tool to quickly send arbitrary messages to wearable clients. The nature of the client-server approach also allowed distribution of events to multiple clients in a very easy way.
In conclusion, a very simple system enabled developers to rapidly explore new ideas on dealing with contextual information.

The first area where the use of wearable computing technology was applied in the SiWear project was warehouse picking. In the beginning, only two types of commands were needed: *application control* and *pick detection*. The interesting observation here is that the broad category of application control is not easy to grasp using terminology associated with context processing, since it is up to the application developer what control can mean. Nevertheless, using (simulated) sensor information and arbitrary control messages proved to be no problem to manage for the developers. Sensor information became *'just another kind of button press'*.

2.1.2 Mock-Up Testing and Iterative Development

The use of mock-ups, that is, the use of partial or make-believe systems, lacking important features, is a valuable tool for getting feedback on systems without having to invest many resources in development. While mock-ups are often used for the user interface of a system, many aspects can be studied with this approach. Potential users can (pretend to) use or interact with these systems and form an opinion. From observations, developers are also able to identify potential problems with the design, e.g. users struggling to understand the system.

Initial studies in SiWear were focused on providing an alternative to paper based picking. In paper based picking, the worker needs to carry a piece of paper that lists items to be picked from some form of structured storage system, e.g. labelled shelfs. Alternatives to paper are the use of audio messages

or computer controlled displays. Switching away from a physical piece of paper is not as simple as transmitting the same information over a different medium. It is easy to understand that reading out the complete list via speech-synthesis will not help a worker much in keeping track of picked items. This is also true for visual systems that should only provide a minimum amount of information in a clearly visible manner, to allow a quick reception by the worker. While the information could be navigated by some form of direct input scheme, the use of contextual information represents an interesting option. If the system can in some way detect correct (or incorrect) actions by the worker, an automatic transition of displayed information and providing additional information is possible.

At this point, mock-up-testing becomes an important tool. Developing actual mechanisms for detecting picking actions is a complex task that requires a significant amount of resources. Without knowing the benefits from having this kind of information starting development is hard to justify. When deciding to use a mock-up, the detection functionality can be emulated by a human operator. This kind of setup is also known as a *Wizard-of-OZ*[Kel83] approach, where the operator (wizard) observes the environment from a suitable spot (behind the curtain) and controls part of the system without interacting with the user of the simulated system. A human operator is an easy to use detector for human-computer-interactions but has to maintain the needed discipline when operating the system (e.g. sticking to set rules and no anticipation of user actions).

When the intention of the mock-up system is to not use it for further development anymore (throw-away prototype), the implementation of the simulated input is of no importance. If however the system is used to develop and improve other components of the system, such as the user interface, one has to think of the phase where the mock-up functionality will actually be replaced by a real implementation. In the best case, the real implementation would be a drop-in replacement for the mock-up that requires no further changes to the components. While this can be solved in various ways for general software development, e.g. by defining interfaces, the domain of environmental context acquisition has also a physical component. When a suitable trigger to navigate an information system can come from a human observer or a sensor in the envi-

ronment, a suitable abstraction has to be found. From a software development standpoint, this abstraction should not be created ad-hoc for every encountered need but come from a generic framework. The use of a framework allows developers to find common patterns in their project to apply the framework on. While developers have to learn the framework at first, re-applying the knowledge later leads to faster development.

When the first tests were done in SiWear, only very simple events in the environment were of interest. One thing to note though is that different means of controlling were handled via the same mechanism, that is, direct interactions from the user, simulated speech recognition and simulated sensor information were transformed into a common representation to control the application. This way of input abstraction was inspired by the work done in the wearIT@work [LTGLH07] project that showed benefits of a common control infrastructure. As a result of this abstraction real systems could later be used to replace simulated functionality as the application was not tied to any specific source of input.

In [Sti08] a system for identifying user activities related to the wearIT@work project is shown. An interesting parallel can be drawn to the design of this system and the applications in SiWear. The activity detection system made use of an existing framework for integrating the sensors and detection algorithms that allowed replacing components developed in special software with more efficient direct implementations after testing. In SiWear, simulated sensor events were later replaced by real sensor events. While not directly related, both approaches show how a framework can accelerate development by abstracting components and thereby making them interchangeable.

2.1.3 Scientific Study Support

While successful wearable computing solutions have been developed and are in use [Sta02] these projects often originate in a scientific environment. Conducting studies and analyzing performance are important for those in the process of the design. An often needed feature, not only in the wearable computing domain, is the possibility of monitoring the internal processes of the application and making the same information the user sees available to observers. A framework for context distribution should support developers in this special use case without

much burden. The studies performed in the SiWear project benefited from such a feature by being able to provide a mirrored image of the user view to a camera for recording while also making environmental data available to an operator for inspection. Inspection of the environment was possible using a generic client that used monitoring features provided by the implemented context distribution framework. This application could be re-used throughout many studies without changes.

Chapter 3

Related Work

Over time there have been various approaches to wearable computing. While many of them were one-time experiments exploring specific properties of the field, some tried to approach the general problem of designing a wearable application by creating software frameworks.

One of the technical challenges involved in every approach is creating suitable technical descriptions of the information needed from the environment or the user of the system - the *context* - in order to make it available to the system. Focusing on this part of the wearable computing field, creating a framework for distributing context information, one first has to define the boundaries of this information by creating a model of context information. Every model used by a computer system is just that, a simplification of a process or entity from the real world. And while such a model will never be able to encompass every situation from the real world, a suitable model will work within a defined set of circumstances. A model should also be as simple as possible (but not more) to be used effectively and efficient by a designer. If a model can be easily understood, its limits can also be understood easily.

To create a model for context information several projects from the pervasive and wearable computing domain will be evaluated in this chapter for their information models to find a general pattern. This model will then be biased towards the anticipated use in a wearable computing scenario but should still

be functionally able to serve pervasive computing applications.

3.1 Roulette Wheel Prediction

The first wearable computing system, built with the intention of supporting a real world task, can be found in the experiments on Roulette wheel prediction by Thorp[Tho98] as early as 1955. While the core idea is quite simple it shows some general needs of wearable applications. To predict the outcome of Roulette wheels, a shoe-integrated computer was used where timing information about the current game was input with the toes. The result of the computation was transmitted to another device[1] worn by a (potentially different) person and mediated by audible signals. Analyzing the needed information here can be quickly done. The computer system needs timing information that is abstracted by keeping track of the time when a button is pressed. The transmitted result is a single symbol that either specifies where to place the bet or not to bet at all. While the setup in this early experiment is technically very direct it can be logically abstracted. On the one side is an action from the user such as pressing a button that is not used directly as a means of control but indirectly used to setup a timespan very similar to the control scheme of a stopwatch. On the other side, an abstract output symbol is computed that needs to be conveyed to the wearer. While this example is very simple it shows one important property. Neither the actual physical source of the input, nor the actual physical representation of the output are important for the application and can be replaced by other means. This has actually been done with the Eudaemonics' shoe [Eud98], a project directly related to the work of Thorp. In this recreation of the project instead of an audio output a tactile feedback device was used to convey the computed symbol to the wearer.

Outcome 1: Context information comes in the form of events and values.
Outcome 2: (Physical) source of context not always important.

[1] The receiver here is actually just an audio playback device

3.2 Implicit Human Computer Interaction

In [Sch00] Schmidt defines the term *implicit human computer interaction* and how sensor devices can be used to create context information for applications. Three key requirements are identified to create software systems that can make use of this kind of interaction (from [Sch00]):

1. *the ability to have perception of the use, the environment, and the circumstances,*
2. *mechanisms to understand what the sensors see, hear and feel, and*
3. *applications that can make use of this information.*

The first requirement is fulfilled by identifying appropriate sensor devices and for the second and third part, an abstraction layer is introduced that allows a uniform description of sensor events and the actions that should be taken by the application. The proposed model behind this layer consists of treating possible events that are determined by sensor evaluation as *boolean* variables that can be used to form complex expressions. The evaluation of these expression can then in turn *trigger* corresponding actions in an application.

The concept requires that sensor readings can be transformed into an application specific meaning and removes the any need to process sensor data at the application level.

Outcome 3: Smart applications do not necessary need raw sensor data but evaluations of it.
Outcome 4: An abstraction that assigns meaning to sensor data allows replacing actual sensors.

3.3 Active Maps

Active map services provide clients with information on located objects. In [ST94], Schilit and Theimer investigate dissemination methods for mobile hosts.

They are approaching the problem of transmitting information over bandwidth limited connections from the perspective of location based services for mobile applications. As a possible solution they evaluated the idea of serving different needs of mobile receivers in terms of wanted information and bandwidth limitations. While they did not specifically create an implementation - at least not for public use - they created an elaborate architecture design to help in designing information distribution systems. The expressed ideas are very interesting from a technical point of view but the proposed systems are specialized to mainly location based information distribution. General handling of context information for more abstract information sources is not covered in this work. A core realization that also applies to general handling of information is however present (from [ST94]):

> *The information that clients are actually interested in is a subset of all the information that might be available to them.*

Outcome 5: Clients are only interested in a subset of available information.

3.4 MIThril Enchantment Whiteboard

As part of the MIThril project for wearable computing at the MIT the Enchantment Whiteboard system is used for communication in general and especially for transmitting sensor information[DSGP03]. The main purpose of the system is to accelerate the development of distributed applications. It follows the paradigm of a whiteboard where a central component can be seen from any possible client who in turn can read and write information from and to it. Additionally, clients are able to subscribe to portions of the whiteboard thus receiving automatic updates. For the transmission of high bandwidth data a secondary system exists that allows direct communication between clients be negotiating a connection via the whiteboard. As a curious fact, the framework was created to serve as a communication mechanism for the context aware window manager Anduin[2] but was soon extracted as a separate project to solve the more general communication problem. This is another example of the observation that general frameworks for wearable computing are not available.

[2] `http://www.media.mit.edu/wearables/mithril/anduin`, accessed 16. Dez. 2013

The main idea of the whiteboard approach is to reduce the complexity of communication. For N clients, instead of having $N * (N - 1)$ communication channels (in the worst case), a whiteboard limits communication to N as each client is only connected to a central point. However, the project contains a small contradiction by providing a non-transparent way of direct communication via its *Enchantment Signal* extension. To use this form of communication two clients negotiate a connection setup via the whiteboard and then engage in direct information exchange. This puts a burden on the application designer to choose the appropriate way of communication for specific information.
While the whiteboard approach to communication seems well suited for information source discovery and transmitting sensor data this implementation is not used widely outside of the MIThril project. Possible reasons for this are, that its use was not well communicated in the wearable computing community and also technical problems. While it was meant to be portable and therefore usable on many platforms it needs to be build for each platform first. This puts another burden on potential users of the software. They do not only have to understand how to use the software but also need to make it work on their desired platform.
Additionally, while the sources are freely available, documentation is very sparse and the project seems to be mostly not maintained anymore. These problems make the framework a niche choice for people that have the required skills to deploy and use it.
The framework is written in the C/C++ language with the intention of being portable. While applications written in these languages have the tendency to be more resource efficiently, especially on embedded or otherwise small systems, compilation introduces a higher effort to setup a system in comparison with interpreted software packages that need no compilation for the target system.

Outcome 6: Efficient communication is a building block for distributed applications.
Outcome 7: Automatic pushing of updated data reduces communication effort.
Outcome 8: A framework must be easy to use to be widely adopted.

3.5 wearIT@work

The wearIT@work project [LTGLH07, LHBK11] aimed at providing a complete solution to the development of wearable computing applications with a strong focus on applications for industrial settings. The project followed a modular software architecture approach. At the center of this approach is the Spring Framework[Spr13], a generic framework for modular systems using the *dependency injection* (DI) approach. This very abstract concept allows specifying dependencies between components at runtime and therefore changing components transparently to the application. The handling of context information is also implemented via DI in a way that the application does not need to know available sources of information in advance as long as the injected components are compatible.

While this approach is interesting from a software engineering point it creates an additional burden on the side of the application developer. The specific aspects of the needed context component need to be defined for all sensors or other sources of context. Only with a consistent definition different modules can be injected later as replacements. Since the meaning of information can vary greatly this approach is not very effective in combination with DI. Furthermore the DI approach shifts the problem of detecting available sensors into the generic framework. In the case of the Spring framework this means that all needed sources of context have to be configured before the application starts and cannot be changed later. A component that detects sensors in the environment while the wearable application is running can of course be added as a component but would defeat the purpose of a central configuration via the framework. The implemented context sources in wearIT@work however are quite simple and can easily be handled by the framework. There are implementations for temperature and light sensors, that transmit their readings as numeric values and switches that have a boolean status.

A big focus of the project was the evaluation of adaptive user interfaces that can use contextual information to change their properties. Handling of user interaction is done by the Wearable UI Toolkit (WUI-TK)[WNK07] with a detailed description in [Wit08]. Following the architectural ideas of the project the

user interface specification itself was highly modular and a designated rendering component is responsible for visualizing an abstract model of the current user interaction possibilities. This component in turn depends on a *context manager* component that provides sensor information in an abstract way.

While the architecture itself is very modular only a few and very simple sensors have been evaluated. A specification of the interface between rendering and context manager component has not been defined leading to some direct dependencies in this project. However, the design envisions the use of distribution systems like the Context ToolKit (discussed in chapter 4) combined with modular systems for data acquisition.

In [WNK07], Witt describes needed architectural support for wearable user interfaces:

> *Adaptive user interfaces have to deal with lots of information to implement some kind of intelligent behavior. Therefore, a suitable architecture for such kind of interfaces has to include access to many different information sources (preferably in a modular way).*

This further underlines the need for a robust way of discovering and making use of sensors in wearable applications and a separation of context handling from the logical transfer of information. The Context ToolKit is directly mentioned as a suitable communication component for this problem. A technical issue with the approach used in this project is that it is virtually not available to third parties. While the goal was to create a common framework that would be of benefit for later projects in the wearable computing domain the required resources have never been formally released and can only be obtained by asking the involved working groups.

Outcome 9: Context information can be optional for a wearable system (e.g. used for optimization).
Outcome 10: Information is not necessarily processed at only one point in the application.
Outcome 11: Distribution systems should not make assumptions on their con-

sumers or sources.

3.6 Context Recognition Network Toolbox

In [BKLA06] a modular system for context recognition is built. The goal of this system is not to provide a solution for making context information available to distributed applications but to create this information easily in the first place. The CRN Toolbox is a set of modular components that provide access to raw sensor data and also to machine learning algorithms that can be used to extract meaning from this data. The authors of the toolbox themselves see it as a complementary tool that can be used together with a distribution system such as the CTK.

While the software has been used in several projects, including the wearIT@work project, it is not widely used in recent projects. The project website is at the time of this writing unmaintained and the provided links to the software are not functional. This is a similar situation as with the MIThril Enchantment Whiteboard and the wearIT@work project in general. The knowledge and resources to actually use the developed software are not available to the public. This is the result of depending on services that have been discontinued for some reason and no mechanism of transferring the information to other services.

Outcome 12: Gathering of context information can be separated from its distribution.
Outcome 13: Separation of concerns makes adaption easier.

3.7 Evaluation of Findings

It is obvious that communication has a vital role in the creation of smart applications. When the number of external or not logically connected sources of information increases having a framework for information distribution reduces the needed engineering effort for the application designer. The remaining question is how to structure information best and how to design an appropriate

communication scheme for such a framework. In [Dey00] many existing frameworks for context distribution have been evaluated to find common needs among smart applications. The type of support for context in these frameworks was evaluated with respect to the type of supported information and the features provided to enable context awareness. From [Dey00]:

> *There are certain types of context that are, in practice, more important than others. These are* ***location****,* ***identity****,* ***time*** *and* ***activity****. Location, identity, time, and activity are important context types for characterizing the situation of a particular entity.*
>
> [...]
>
> *Our proposed categorization [...] is a list of the context-aware features that context-aware applications may support. There are three categories:*
>
> 1. ***presentation*** *of information and services to a user;*
> 2. ***automatic execution*** *of a service; and,*
> 3. ***tagging*** *of context to information for later retrieval.*

The result of this evaluation did show a high diversity in the support for these features in the various frameworks. They have been created to support a specific type of task and are not suitable for general use. This lead to the creation of the CTK with the goal of creating a framework that supports all needed features from the evaluated domains. While designed for general use its main goal was supporting smart applications in a pervasive computing setting. In this field, constraints from the environment only have a minor impact on the application while they are much more limiting in the domain of wearable computing.

By evaluating frameworks that have been designed with wearable computing applications in mind more fitting criteria will be found. This evaluation will evaluate not only the logical needs in terms of context information but also technological needs arising from the special conditions. While the outcomes from the first examples show that context information can be used as an abstraction for direct input (outcome 1,2) further examples from the field show the need also for implicit interaction in a wearable scenario (outcome 3,4). A

context information framework therefore needs to be able to process both kinds of information. Furthermore, a common problem with context information is finding an efficient way of transferring the information across many devices (outcome 5,6,7,8). Here a difference can be found between wearable and pervasive computing applications. Wearable applications will also often encounter a change in available sources for information and generally need to be able to discover and select them according to the current needs of the user.

The wearIT@work project also shows a very complex interpretation of context information that affects the wearable application in various ways. Not only explicit and implicit interaction has to be taken into account but also other information sources can be used to further control the wearable application, for example, by changing the representation of information to better match the current situation (outcome 9,10,11). Finally, a general trend can be observed in favor of modular systems (outcome 12,13). A context distribution system does not necessarily need means of interaction with the sources of information but provide only a distribution scheme.

As a non-technical observation the few frameworks that are created for solving general problems for wearable computing applications so far seem to become unmaintained (or even unavailable) after a few years for various reasons. In general, the required knowledge to operate and enhance the provided software seems to only be present among very few developers and to vanish when they leave the field of wearable computing. When a new wearable computing application is to be developed this situation will become the first road-block for the developers. One could argue that the availability of a framework is equally important as its suitability for the task at hand. Even if the description of a framework looks promising a lack of resources to actually use it will encourage developers to start writing their own solutions. This is however true for software systems in general and not a specific problem of the wearable computing field. Applying proven methods of keeping software available to a larger community would be beneficial for new projects.

A long lasting solution for wearable computing should therefore make use of public software distribution and documentation systems to ensure that main-

tainers that are not necessarily part of the original development team can take over projects easily. Therefore the need for modular systems does not only apply to the actual software design process but should also be adopted on a higher level. There exist various independent online services for collaborative software development that are not affected by changes in working groups and have the added benefit of providing easy ways for interested third parties to contribute to a project. The software developed during this thesis has been published on the GitHub software collaboration platform and can be found at:
`https://github.com/wearlab-uni-bremen`
This popular platform allows free access to the code for any interested party and will hopefully exist for a long time.

Chapter 4

Evaluation of the Context ToolKit

4.1 Context ToolKit

In the previous chapter various needs for context information in applications and possible methods of transmission have been examined. Many of these transmission concepts have been developed for a specific use and are therefore only to some extend reusable. This chapter will focus on an existing framework for context information transmission that was explicitly designed to be generic. Previous systems have been analyzed for their needs to provide a solution that can be applied to all scenarios. While the framework is a very elaborate solution to this problem, it was not designed with wearable computing applications in mind and it will be shown that its approach may be unfitting for this domain.

The Context ToolKit (CTK)[Dey00] is a very elaborate system that deals with creating an infrastructure for context sensing devices (sensors) and applications using the gathered information. It is engineered following an object oriented paradigm were a common base class enables communication between objects. Subclasses are implemented for sensors, interpreters and other participants in a context aware application. On a very low level the system relies on the HTTP protocol[HTT99] for communication and each object in the system acts as an individual HTTP server and client for communication. It uses a network broad-

casting approach to find components in the environment which enables context aware applications to be used almost without configuration. However, this approach only works if the underlying network structure supports this kind of technique which is not always the case, especially for many mobile networks.
An interesting aspect of CTK is the use of the *widget* metaphor for working with context information. It allows the programmer to make use of an external sensor as if it were normal type of user interface element. This approach abstracts the use of the context information from the technical means of getting the information.

4.1.1 Design Concept

The central design aspect of the CTK has been to create a system that is easy to use for the application designer. It takes up the *widget* metaphor from graphical user interface design to provide a similar mechanism to work with context information. An application designer can therefore apply existing knowledge on graphical user interfaces to working with context. Making use of this knowledge is an interesting approach since a lot of effort has been made to create consistent schemes for user interface programming as shown in [MHP00]. A GUI widget like a button serves as a mediator between the user and the application by providing a known element to the user that has a defined meaning and behaviour. The inner workings of how input devices act with the button and how the widget triggers the associated program function are of no concern to the user and the application designer.

There are however differences between this approach and standard GUI programming. While GUI widgets are created by an application for a specific purpose context widgets are shared among applications since the same context information can be of interest for more than one application. From the application design view this is only a minor difference since the important part of triggering a defined function still happens for each application independently. Having an easy to understand metaphor for context widgets is not only meant to provide an easy to use framework but also to encourage building additional widgets that can be reused. From [Dey00]:

> *[...], we need to provide support so that the building of these components is actually easier than ad hoc implementation within the application.*

In addition to the context widgets the CTK also introduces the concepts of *Interpreters*, *Aggregators*, *Services* and *Discoverers* that correspond to other needs of context aware applications. A discoverer in this framework is a component that acts as a registry for other components and provides services to find remote components. Figure 4.1 shows the core of the object hierarchy used to model these concepts where the arrow direction indicates the specialisation of classes. The common ancestor of all objects is the *BaseObject* that provides the communication infrastructure used by all components.

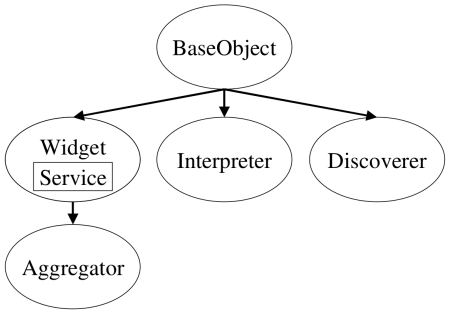

Figure 4.1: component model in CTK (from [Dey00])

4.1.2 Architecture

The framework has been implemented in an object oriented approach using the Java programming language. All concepts have been modelled to include a common parent class that provides functionality for communication and all other common tasks shared among the concepts (such as storage, timekeeping etc.). While the components in the framework are decentralized and therefore do not provide a central point for finding other components the discoverer concept provides this kind of functionality. Applications do not need an explicit discoverer component but can instead use network broadcast messages to find one. When

a discoverer component has been found, it can be queried for other components in the network that in turn have registered with this discoverer component.

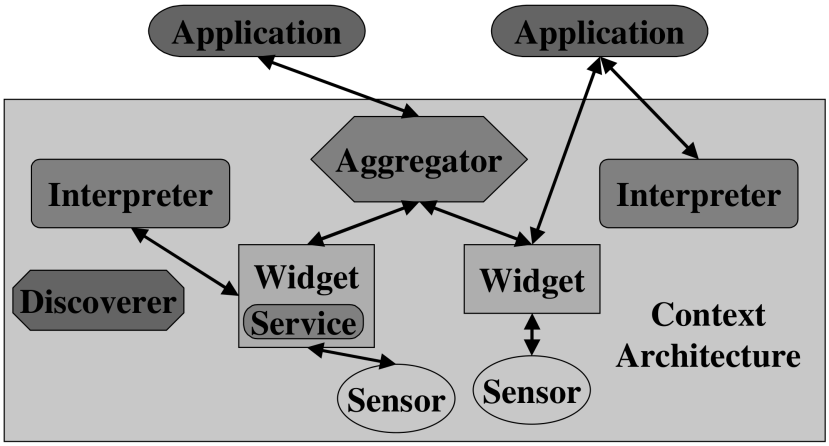

Figure 4.2: CTK architecture (from [Dey00])

Figure 4.2 shows an example constellation of components that can provide contextual information to applications. The component structure of the framework suggests using a container framework like Spring[Spr13] to provide a flexible system for starting the main part of a smart application. Distributed sensors however will more likely be implemented as stand-alone applications communicating with the rest of the system.

The communication protocol used by the components has been designed to be independent of the used programming language and operating system and makes use of XML for data representation and HTTP as a transport mechanism. It is therefore possible to have independent implementations of various parts of a context aware application. By using a defined set of messages encoded in XML components can transparently exchange information.
By using HTTP for information transport each component has to act as a HTTP client and also as a (simplified) server. This implies that each component needs to open a network port on its local system to be reachable by other components. The combination of the host name (or IP address) and the used port can

therefore be used as a unique identifier for components[1].

4.1.3 Context Information Model

Context information is represented by *attributes* of context widgets. These attributes are complex data types and are identified by a given name. The value of an attribute depends on its type that can either be one of many standard primitive types (boolean values, character strings, numerical types, etc.) or a nested structure of attributes and their values. This data model for context information permits to encode virtually any information that can be found in applications or databases.

There is however no meaning attached to the information. External conventions have to be made to provide a common understanding of attribute names and the meaning of the value. When an application is searching for a provider of context information it can send a description of the needed data structure (in terms of attribute name and type) to a discoverer that will eventually provide suitable components.

4.1.4 Context Information Delivery

Analog to graphical user interface widgets context information is delivered as an event to the application. While normal input events are sequential and typically handled in a single thread of execution different context events can happen while others are still being processed. Widgets in the CTK therefore deliver context events in individual threads of execution.

While this has some benefits it puts an additional burden on the application developer in terms of potential problems of parallel access to resources and also increases needed resources by creating additional threads. In case of high frequency events from sensors the creation (and dispose) of threads may require a lot of system resources. The creators of the CTK are aware of this problem and propose a selectable means of delivery (e.g. single threaded) but so far have only implemented the multi-threaded behaviour.

[1] at least in local area network (LAN) structures

4.1.5 Usage in Applications

While applications can make use of the CTK framework by implementing the used protocol themselves reusing the provided software as a library is a more convenient approach. Application developers will mostly make use of the common base class provided by the framework that implements the communication protocol. Relying on the existing methods retrieving the context widgets and subscribing to events is a straightforward process. The only thing left to the developer is implementing the application logic that performs tasks corresponding to the received information. Developers that want or need to to create new sources of context information will make use of the *Widget* class itself that provides the needed mechanisms to define attributes and distribute values to subscribers.

Receiving Information

When an application wants context information it needs to create a subclass of the base class to have access to the communication mechanism. This allows it to automatically search for discoverers in the network. To find a suitable widget a query is sent to a discoverer through a method of the base class. No direct communication with the discoverer is needed. The query contains a description of the needed attributes and types. The discoverer will return a list of matching components that can be queried for further information. If the application decides to use a component it subscribes for receiving updates. In case that no matching component is found an application can also subscribe to the discoverer to be notified on the discovery of new components to eventually find a matching widget later.

When new data is available from the widget a handler method is called where data can be parsed from a supplied data object. The data object represents the hierarchy of nested attributes and values and needs to be traversed to reconstruct the needed information as an application defined data type (if needed). This data handling is also done when waiting for new components from the discoverer. Specialised methods transform the data object to a component representation that can be used to subscribe to the corresponding widget. There is only a single handler method for all subscriptions and the application has to

keep track of the information source by comparing a supplied identifier.

Providing Information

To provide context information an application has to create a subclass of the *Widget* class to inherit needed functionality. For simple information cases (e.g. temperature readings) the class only needs to specify the provided attributes and a way to obtain this information. By creating an *update* notification the new value is automatically sent to all current subscribers of the widget. Per default a widget will try to locate a discoverer and register itself with them to make it available to other components.

4.1.6 Code Metrics

The framework has been designed using an object oriented approach. This is characterised by the introduction of abstract base classes and subsequent specialisations of these classes that inherit a certain set of common functionality. In addition to the encapsulation of functionality in classes, packages are used to form a logical grouping of classes inside the structure of object oriented software. The package concept is not only found in object oriented programming languages but is also a method of logical separation of functionality in general. When building software systems reducing the dependencies between packages and therefore also between classes from different packages in the object oriented approach is desirable from a software quality viewpoint. In addition, the complexity of individual methods should be kept low to ensure that their behaviour is well defined and can be tested automatically. There exist tools that can automatically analyze code and generate quality metrics. Using these metrics developers can set acceptable limits to their own work and receive automatic feedback on the current state of their development. If a limit is exceeded a developer has to decide whether this result is still acceptable for the given method or package or if changes need to be introduced to maintain the desired goals in quality.

Code metrics can also be applied to foreign code to get a quick overview on potentially problematic areas in terms of maintainability or methods with a potentially complex behaviour. In case of the CTK framework code metrics have

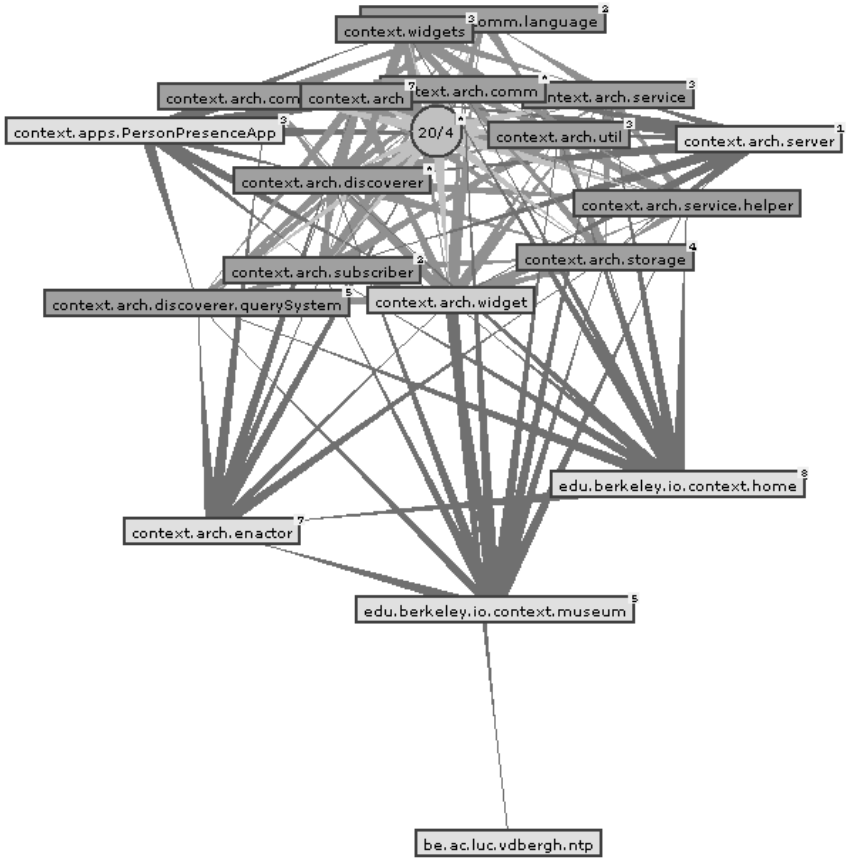

Figure 4.3: dependency graph of the Context ToolKit

been applied to get an understanding of its structure and dependencies between packages. Figure 4.3 show the result of an analysis using the *metrics2* [Met14] plugin for the Eclipse Java IDE[Ecl14].

The nodes in the graph represent packages where a number in the upper right corner shows the number of sub-packages not shown. An edge in the graph represents a dependency where the thinner side connects to the package that is being used by another package. The central package of the graph that is used as the origin for the dependency analysis is colored in a grey shade and normal packages are colored lighter. During the dependency analysis some packages

may be found to form circular dependencies with each other. These packages are called *strong packages* or *tangles* and are colored darker. A grey circle is associated with each tangle and shows the number of involved components and the length of the shortest path from one component to another one in the cycle. Analyzing the CTK shows that 202 classes inside 28 packages are present where about 30 of these classes are not strictly part of the framework but can be used to demonstrate some functionality of the framework. The dependency graph is therefore quite complex and does not intuitively provide an insight into the framework. The graph does show however a very high dependency between packages in the framework as indicated by the single tangle that almost connects all packages. On the outer side of the graph packages containing demonstration code are visible that are also logically separated by a different base package.
A feature of the code metrics plugin is the inspection of individual tangles that can provide a better view on the packages involved in a cyclic structure. This detail view can be seen in figure 4.4.

For an application developer the creation of widgets is the central entry point into the framework but the high amount of dependencies between packages can be confusing. Apart from human confusion each dependency between packages increases the *instability* of software. Instability, as defined by [Mar03], describes the effect of changes outside a package on the package by the ratio of *efferent coupling* (dependency on external packages) to the sum of *efferent* and *afferent* (external packages depending on this package) coupling. In an ideal case, a package does not depend on classes from other packages and is therefore not affected by change. Such a package would have an instability of 0. The most unstable package, a package that only uses external packages, would have an instability of 1. A cyclic dependency can therefore have a great impact on a software system where a change in one package requires changes in all other packages inside the cycle.

The whole CTK source code has been analyzed to have an average instability of 0.403 ($\pm$0.292) but this value cannot be directly used to judge the framework as it also contains the demonstration code. The demonstration code has naturally a high instability as it only uses other packages and biases the overall result. More appropriate is to look at the package containing the base class to

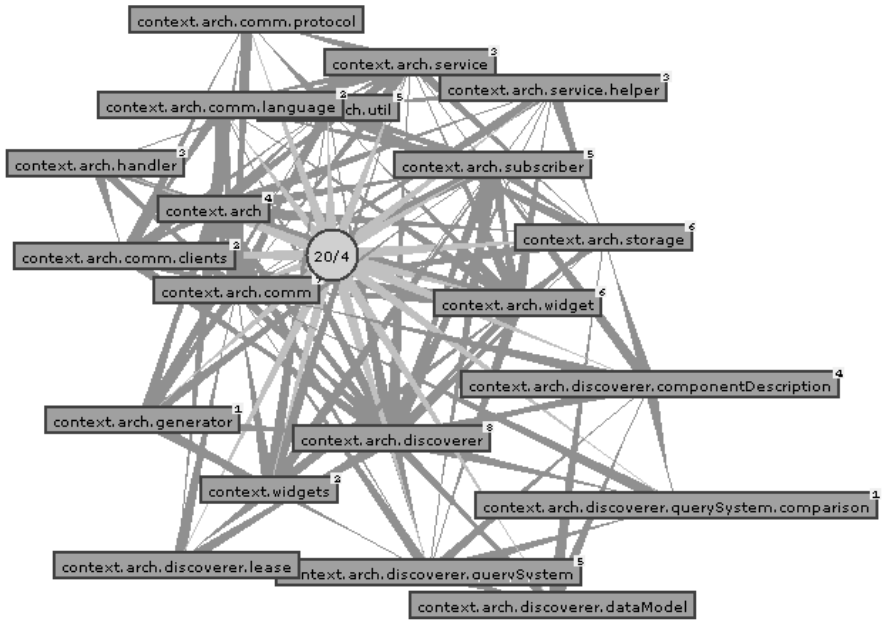

Figure 4.4: dependency graph of the Context ToolKit core

derive new context widgets. The *context.arct.widget* package has an instability of 0.278 derived from an efferent coupling of 5 and an afferent coupling of 13. Further generated metrics show that most of the instability in packages is found in internal packages that deal with the protocol and widget discovery and do not directly affect the developer of context widgets.

To get an understanding of the complexity of the framework classes the *Cyclomatic Complexity* measurement[McC76] of class methods is calculated. While this metric only counts potential control flow paths in methods it can be used to find methods that are more complex than others in a software system. The generated metrics show an overall complexity of 2.061 ($\pm$2.570) for the complete framework (again including the demostration code) with a maximum value of 54. This highest value is found in a class dealing with the storing of context information. Other methods with very high complexity deal with internal discovery mechanisms. For the *BaseObject* (and therefore any *Widget* implementation)

the methods dealing with querying the framework for context widgets have a higher complexity (9-11) which comes at no surprise.

4.1.7 Discussion

The idea and implementation of the Context ToolKit are very well suited for providing context information in a distributed system. The used widget metaphor and object oriented approach make it easy to create applications using the system without knowing details about the implementation. While it is very straightforward to create a context providing widget the handling of received information is more difficult. The framework supports the developer by providing a callback mechanism that is similar to the response to user interface elements. Extracting the actual information however requires dealing with a complex abstract data type. When looking at the actual implementation it is unclear if a better approach to the data extraction issue exists, e.g. if the framework could provide methods to automatically extract requested data types but the provided examples are showing only the manual way. Related IPC frameworks such as the Java Remote Method Invocation (RMI) or D-Bus proxy objects allow an automatic handling of abstract data types by a compatible local wrapper to lift the work of manual extraction from the developer. This functionality may or may not be already present in CTK. Documentation that clarifies this point has not been found. Seen from a high level perspective the design of the framework and the model of context information is suitable from the information distribution aspect for smart applications, but shows drawbacks when used in a wearable computing scenario.

The choice of a decentralized communication scheme makes the system flexible but also more resource intensive. Each component has to act as an individual server to process HTTP requests and each request is handled in an individual thread. While this effect has only a small impact on a typical desktop or server system a small or embedded system may have difficulties providing the resources while still being responsive. As a side effect of using HTTP as a transport mechanism a lot of information needs to be send for every request. This is a result of the stateless nature of the protocol that treats each connection independently and therefore needs instructions from the client to perform an operation. This is a common problem with Web Services[Web04] where appli-

cations request information that depends on a previous state. While there exist techniques to mitigate the information overhead such as persistent sessions or the use of WebSockets[Web11] they have been retrofitted into the stateless nature of the protocol to be useful when mixed with applications running inside a web browser. Implementing these measures can reduce information overhead but adds an additional burden on the server side. In case of the CTK, where all components implement a HTTP server, this would mean to trade transmission costs for computation costs. Overall this could be a benefit but would at least make the system more complex.

The used method of component discovery may also pose a problem for wearable computing. A broadcast message is typically only visible in the same network but will not be transmitted, for example, from a wired network into a wireless network. Therefore a component in the environment (LAN) will not be visible from a mobile client (WLAN, GPRS). The components of the CTK can communicate directly if their address is known in advance but doing so is a contradiction to the discovery mechanism. Using XML to encapsulate data is a popular choice but also wastes a lot of data that is mainly useful to allow human inspection. While wireless networks become faster they still offer only a fraction of the bandwidth found in a wired network. And even if network speed is not an issue transmitting more data results in higher energy use. Energy is a very valuable resource for a mobile client and therefore a more compact transmission format would be desirable.

From a software development view the framework is robust but very complex. There is no technical separation of the framework code and the code used for demonstration. This supplies a new developer with examples on how to use the framework but adds unneeded resources to a project. The overall structure of the project shows cyclic dependencies that create potential burdens on further maintenance of the framework. While a developer will typically not deal with all aspects of the framework and the packages that are directly used only have a few dependencies it complicates the understanding of the framework.

The goal of the CTK framework for context aware applications was *to provide support so that the building of these components is actually easier than*

ad-hoc implementation[Dey00] but it seems that this goal is not fully reached. While the creation of components is very simple and a discovery mechanism is in place the usage of the provided information can still be difficult.

Chapter 5

The TZI Context Framework

Context information is any information relevant to the current use of the wearable computing system. This includes information on the user and the system itself (c.f. [Dey00]).

To make context information available to an application it has to be input into the system by some technical means. Many approaches have been shown in the past where each can be used to solve particular problems in this domain. In this chapter the design and implementation of the TZI Context Framework (TCF) is portrayed. This framework is a direct result of the requirements of the wearable computing approach developed in the SiWear project and was developed at the Center for Computing and Communication Technology (TZI). Its main focus is the integration of contextual information into a typical network found in industrial environments and limiting the needed resources on the wearable clients.

After outlining the motivation and discussing related technology the representation of contextual information is presented. Following this high level concept the actual implementation in an object oriented approach is shown. A comparison between the TCF and the CTK reveals the strengths and weaknesses of the individual frameworks and verifies the fittness of the proposed framework for wearble computing applications. A discussion of the concept of handling high level context and the evalaution of the framework in the SiWear project conclude the chapter.

5.1 Motivation

Unlike typical applications for desktop environments, wearable applications are in general not self-sufficient (c.f. [SS02]). In order to work at their full potential they need access to systems that are not worn on the body. At a very low level this dependency on other systems results in the requirement of wireless networking for the wearable system that can come in different flavors. For industrial settings a wireless network may simply be a connection to wireless LAN access point that provides access to the rest of the available network infrastructure, possibly including access to the internet itself. For mobile wearable applications a wireless network will likely be realized by connecting to the internet via a cellular network were bandwidth and connectivity are far more limited compared to a WLAN connection. Between these two extremes are many possible variations, e.g. in some scenarios a wearable system may rely on available public WLAN hotspots to access data when possible.

Some existing systems for context distribution have been designed to implement smart objects in a fixed environment[Dey00]. For these approaches bandwidth and connectivity issues are virtually not existent as these have to be solved once when the smart object is installed into the environment. When these systems are to be used in a mobile scenario, several problems arise. Apart from dramatically reduced bandwidth network techniques for discovery, such as broadcasting, may also not be available in such a network.

The context distribution system that was developed here has been built to be usable in a limited network environment - especially in cellular networks. Key ideas here were the reduction of transmitted information, explicit handling of sporadic disconnections, easy protocol handling on the client side (for resource-limited devices) and robust discovery of information sources.

5.2 Distributed Communication Schemes

As previously discussed, the communication approach used in the Context ToolKit (CTK) is suitable for making information available in distributed systems but has high demands on bandwidth and processing power. Other ap-

proaches focus on special fields where only limited types of information are to be distributed as shown in [RLRT11]. The distribution of information is a common problem in many computer systems and many specialized systems have been developed. To motivate the development of a suitable communication scheme for wearable computing applications a few selected systems haven been investigated.

5.2.1 D-Bus

The D-Bus[DBu14, Lov05] system is a framework for inter-process communication (IPC) and is part of many Linux-based operating systems. Applications can register themselves with the D-Bus service to provide services to other applications. Messages can be either method calls that interact with an application or signals that notify applications of a certain event.

This idea can be extrapolated to sensor systems where sensor information is distributed in the same way. However, D-Bus is limited to a local infrastructure and while it would be technically possible (to some extend) to transfer messages in distributed systems, it was not designed for this purpose. The system uses a *forest*-structure to group information where each *tree* is constructed from textual identifiers. This way of grouping by textual prefixes is commonly found in many aspects of programming (e.g., nested data-structures, package hierarchies, file systems, etc) and is therefore easy to grasp for application programmers while it puts additional burden on the technical system. In other words, the D-Bus system provides abstract information in a commonly understood structure.

5.2.2 IRC

Internet Relay Chat[IRC93, Wer96] is a protocol that solves the problem of offering a global system for textual communication that uses a distributed network. The structure of the protocol is designed to be human readable and easy to implement for different systems. While not designed for carrying sensor information the design can be adapted to fit other needs. A special client-to-client protocol (CTCP)[IRC94] exists to create an isolated communication channel between two users that can be used to implement file exchange and other services. The concept of people talking in different rooms can be seen as an analogy to

the use of sensor information. The same utterance in different rooms can have a different meaning. That is, the room an IRC conversation takes place provides a context for interpreting what was said. By joining a room a user expresses interest in data concerning a specific topic. IRC therefore models a way of structuring information flow by letting users choose where they say something and also from where they want to receive information.

5.3 Describing Context in an Abstract Way

One of the problems when designing context aware systems is finding a proper description of the needed information that is not too specific to the used sensors and also not too complex for practical use. The meaning of *context* is also not fixed and may be different in many application domains. There have been many attempts at providing formal descriptions for context information ([KWN05, Sch00]) but they have not found a wide use for various reasons, e.g. being too abstract, too complex or just not suitable for a given task. It is unlikely that one framework will be able to suit the needs of every possible use. On the other hand there are not many applications for end users that make use of very complex information from the environment. The design of the general framework for handling context will therefore focus on being easy to use from a developers point of view. Easy use of a framework will encourage experimentation which will hopefully lead to improvements. In other words: since little is known about how to use complex information in applications, a context framework should concentrate on providing easy to use means of dealing with simple context information.

This leads to another question: *What is simple context information ?*

A simple context does not necessarily mean that it consists only of simple information. A prominent example is the use of location information in mobile applications. Getting coordinates for positioning is a complex task but the use in the application is actually simple. The application does not have to deal with the *how* of getting the information but just incorporates that information for other purposes like location based services or just plainly to show the users location.

In this work, simple context information will be characterised by the low complexity of the contained information. Location information for example simply consists of coordinates in a common coordinate system and some kind of error estimation. In other words: a simple location context does not provide all means to infer the meaning of the values (e.g. coordinate pairs). It is expected to be used in a common way, e.g. coordinates are implicitly expected to be used for terrestrial navigation and not as markers for craters on the moon. Another example are temperature readings. A simple numerical value is not enough to infer the used system (Fahrenheit, Celsius, Kelvin, ...) but is sufficient if the system is clear from the application domain.

Assumption 1: Simple context information provides values but does not convey any meaning.

One could argue that this kind of simplification will lead to incompatible systems due to misinterpretation of readings. But from a practical point of view the change of coordinate systems and units will occur very seldom. Instead of providing information for late inference the framework should rather make it easy to write wrappers for sensors that use a different system. And in cases where creating a wrapper is not possible it is also unlikely that the application would be able to use the data even if it could infer information on the semantics of the data, e.g. adding a gyroscope to a system that has no ability to process rotation is futile. If an additional sensor is intended to stabilize or improve another sensors reading, the output of this system can be used instead when wrapped appropriately.

Assumption 2: A system for handling simple context information is enough to create context aware applicaitons.

Even among sensors that provide simple information in the previously mentioned sense a logical structure has to be created to allow uniform treatment of different sources of information. While an explicit sensor can be accessed in a specific way by an application (e.g. reading coordinates from a GPS receiver) allowing it to interpret the values correctly, this method does not deal well with

the replacement of sensors. For some applications defining classes of sensors among their domain may be appropriate but it suffers from a similar problem that is found in object oriented programming. Here a specialized instance of a class may provide useful properties for the application to exploit but these are not visible if it is used as a general (less specialized) class instance. A practical approach to this problem is checking for known specializations but these have to be known to the programmer. Sensors can be similar to this. For example a light sensor is expected to provide the level of lighting. More specialized sensors may be able to also provide their limits (e.g. maximum reading) or the resolution of the sensor (e.g. a very simple sensor may only be able to differentiate between 'no light' and some defined threshold before switching to 'light').

The underlying problem is the same as with the object oriented programming: An application can only make use of properties where their meaning is known. But unlike OOP, a set of necessary and optional attributes can be defined for many sensors by common understanding of the application domain. Missing attributes will probably degrade the overall performance of an application but it can still fulfil its purpose.

Assumption 3: Sensors can provide optional information besides the required readings. Due to the nature of these additions, a formal description would be very complex. A simple, common-sense based approach will have a positive effect on using this kind of information

Regardless of information being optional or necessary a need for grouping information concerning a physical or logical object arises from these observations. A framework for handling sensor information needs to provide some kind of abstraction to relate sets of values to a particular sensor and also to support multiple instances of the same sensor (e.g. providing support for multiple acceleration sensors mounted on tools, the body, etc.). The most important decision is the handling of values from a multidimensional sensor. The longitude and latitude information from a location sensor are mostly useful together, as are the values from a tri-axial acceleration sensor. From a technical point these values can be handled as an atomic set or as individual entries. An atomic set means that the values in this set can only be transmitted or received in one operation in

contrast to multiple operations used for individual values. This distinction can matter when information is transfered in real time. For one reason or another, individual values may not always arrive in a determined sequence and have to be combined by the application. On the other hand, individual entries can be retrieved selectively and more important (from a developer point of view) avoid the need for complex data types.

The proposed framework choses the approach of using individual values. In section 6 it will be shown that this can also have a practical benefit on bandwidth consumption. However: if atomic value sets are needed, the framework should support them by letting the application designer take care of the details (e.g. wrapping and unwrapping multiple values from a single transmission).

Assumption 4: Avoiding complex datatypes in the framework allows easier general use.

The following chapter will outline the logical structure for dealing with simple context information by following these observations.

5.4 General Context Handling

Previous approaches concentrated on either the access to context information or on the means of recognizing context from given signals. Also distribution of context was addressed. In order to create a truly reusable system for context use it has to be operating system, programming language and interface neutral. The less assumptions on the system are made the higher the chances are for a successful reuse.

This comes at a price however. Neutrality means that such a system is unable to convey the meaning of information, that is, semantic information can not be part of the information and has to be created at the application level. On the other hand this approach allows for injecting arbitrary data into the system without bending or breaking semantic rules. It therefore allows rapid prototyping of context aware applications without explicitly specifying semantics that may change at a later point for various reasons.

The proposed framework has been built on the following assumptions:

- Information in a context is given a semantic meaning by the context.
- There can be more than one context in a domain allowing for different meanings of the same information.
- Information has a source. That is a physical or logical object to which it belongs.
- Sources have one or more properties that carry information.
- Each information is assigned a time-stamp indicating the time this information became valid.
- Context information can have a persistent or transient character. That is, persistent information is valid until it changes while transient information is valid for a short time or only punctual.
- While semantic information cannot be transmitted certain classifying elements (tags) can be attributed to information.

A simple example for this view on context information is given in figure 5.1. A room has two sensors that monitor the state of a window (open or closed) and the temperature in the room. The state of the window is an example for persistent information as it will not change over time. The temperature sensor provides transient information as the measurements are taken periodically and may become invalid very quickly.

This example also shows the logical grouping of context information. The physical room has two properties: the state of the window and the temperature. Inside a given context the room is a source of information. Both properties contain values that are related to the room and can be interpreted according to the context of the room.

This kind of separation represents a simple hierarchical model of three layers as shown in figure 5.2. Information is first grouped by a context in which the information is to be interpreted. Secondly, physical and logical items inside the

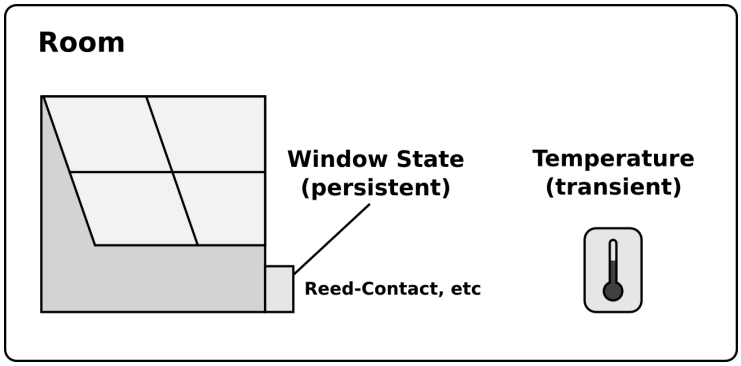

Figure 5.1: room context example

context form sources of information. Inside these sources an arbitrary number of properties hold values from the environment.

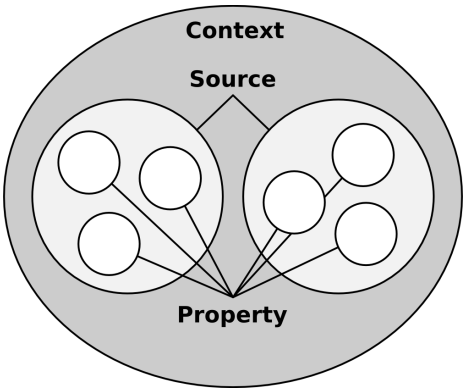

Figure 5.2: context hierarchy

Properties themselves hold the corresponding value and also other information such as the validity of the information (persistent or transient) and a time-stamp. Furthermore, an optional set of tags can be associated with a single piece of information. This structure is visualized in figure 5.3.

5.4.1 Infrastructure for Context Distribution

A common scenario in wearable computing is the availability of numerous context sources and computing devices that require specific types of information

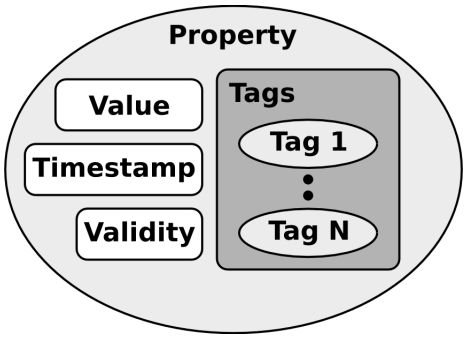

Figure 5.3: property details

from these sources. The problem of providing information to these devices is best solved with a *Client-Server-Architecture* were a server acts as an aggregator of context information and clients provide or request specific information to or from the server. Seen from a software development side this follows both, the mediator and the observer pattern as found in [GHJV95]. A mediator is needed as context aware applications do not necessarily know the specific type of sensor found in the environment and therefore need a discovery technique. The observer pattern reflects the need of applications to be notified of changes in the sensor readings or the occurrence of events in the environment.
The server side of the framework is designed to run on a computer with high processing power and availability thus being able to serve many clients simultaneously. Clients only need very little computing power and may stop and start participating in communication frequently.

Clients providing contextual information send it to the server where it is stored. If a client needs access to context information it can either actively request information from the server or subscribe to certain types of information which are then pushed to the client on arrival at the server (*push*-approach). Figure 5.4 shows a diagram of the client-server structure. This design additionally allows chaining together several server systems by having a server act as a client to another server and mirroring arriving context information or selected parts. This allows for building redundant systems or specialized sub-servers only providing specific information to reduce processing load. See figure 5.5 for an illustration of this kind of filtering.

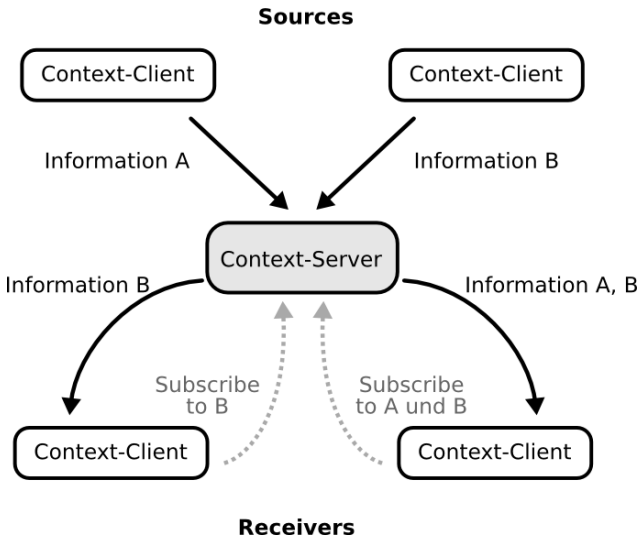

Figure 5.4: client server diagram

5.4.2 Data Structures for Context Information

While the fixed structure of context information as defined by the framework allows a very simple abstract data type for modelling definitions to transfer this information over a network connection have to be defined. When clients receive information or send an update of a sensor value it has to be encoded into a data stream that represents the internal model.

Updates of context information can originate from two different components of the framework. The first component is the server that manages all sources of information in the environment and needs to keep track of changes sent by connected clients. The clients are the second component in the framework that have to deal with changes sent by the server. Following an observation from the needs of both, context providers and context aware applications, clients may require only very little functionality in terms of context management. While the server needs to keep track of new context sources or their removal, a client might only be interested in getting updates on a very specific set of information. In that case, replicating the tracking of information on the client would waste resources. On the other hand, some clients might be interested in this kind of information, e.g. a context aggregator might depend on the presence of several

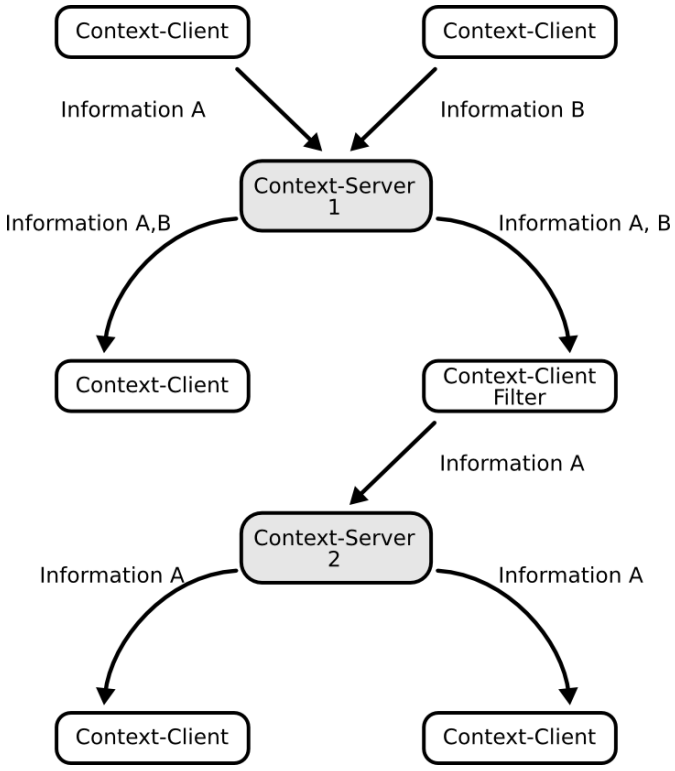

Figure 5.5: context filtering for a second server

information sources and has to react on changes of the environment.

Storing of context information on the server is separated into the maintenance of an environment model that keeps track of existing sources of context information and the notification of clients when requested information changes. The environment is modelled very closely to the proposed data structure for simple context information but implements a design that allows resource efficient access and changes to the components inside the model. Figure 5.6 shows a simplified UML diagram on the implemented data structures but leaves out details. See figure B.1 in appendix B for a detailed version. The diagram shows how the framework models an active environment of information sources. The *Environment* class serves as a central control structure that encapsulates the management of logical elements in the context model. Only one instance of the environment is present at a server or client. The actual model of context infor-

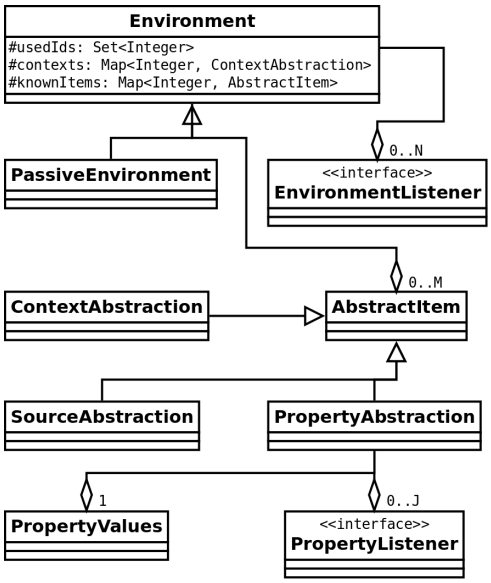

Figure 5.6: simplified environment model

mation inside an environment is represented by the classes *ContextAbstraction*, *SourceAbstraction* and *PropertyAbstraction*[1]. All of these classes are derived from the common base class *AbstractItem*. The software development approach behind this allows the environment to treat all of the abstractions in the same way as the basic *AbstractItem* class. Inheriting from this base class all items have an assigned numeric id that is unique inside the environment. This enables the protocol to reference any item inside the environment by this number and in turn allows a very efficient lookup of items. The environment class provides methods to perform different lookups that can be either based on the given names of items or their numeric identifiers. In addition to the management functionality, the environment representation implements a callback functionality that can be used to notify interested objects of structural changes in the environment such as the addition or removal of elements.

While the representation of context and source elements is straightforward, the representation of properties is more specialized. The *PropertyAbstraction* class

[1] by convention, Java class names are written in *CamelCase* where each word begins with a capital letter

does not contain the context information but only a reference to a *PropertyValues* object that serves as an abstract data type encapsulating the information. This separation of concerns allows to add or remove elements to the property model at a later state if needed without influencing the management of the environment. In addition to the value reference the *PropertyAbstraction* class implements a callback functionality that can inform registered listeners of changes in the values. Using this mechanism updating values inside the environment will automatically create an appropriate event. This is used inside the server to process the subscriptions to context information by individual clients. While the environment model is the central part of the server system clients might need the ability to mirror a subset of the environment to perform special operations. For this case the *PassiveEnvironment* class exists that is functionally identical to the *Environment* class in terms of managing changes in the environment. The difference between these two implementations is that the passive environment does not assign unique identifiers to items but requires them when items are added to it. Using this special class a client can keep track of an environment by using the same identifiers as given by the central server component and make use of the efficient lookup mechanisms.

While the environment model is a very sophisticated approach for dealing with a changing environment it can become rather complex. Simple clients that only request a defined set of information might not want to invest the needed resources required by the model. The framework provides a second model that still provides some convenience methods for dealing with context information but does require only a low amount of resources. The UML diagram in figure 5.7 shows the involved classes and interfaces. The class *Context* represents a complete logical context inside an implicit environment and manages its sources and properties on its own. The *ContextElement* class in this model serves as an abstract data type that references values of a specific property in a given source. Information on the context of the source is not present in this class. The *Context* class implements a callback functionality that allows classes that implement the *ContextListener* interface to be notified of events in this specific context. This includes structural changes such as the addition or removal of sources and also provides a method of processing updates to the values of a property. All methods that deal with changes to a context are supplied with

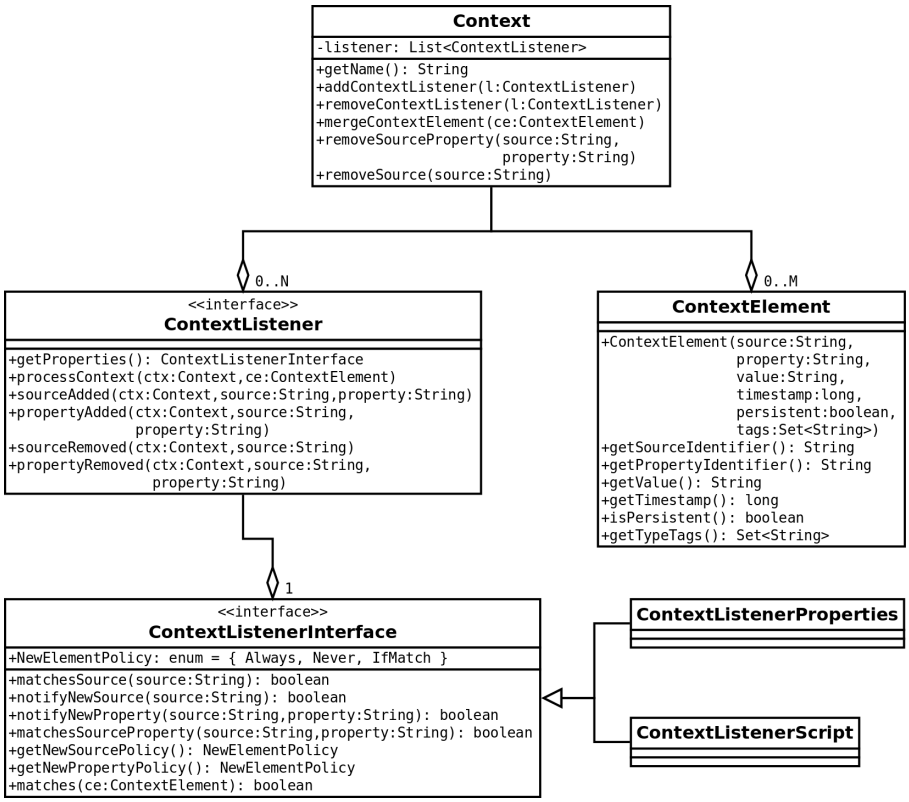

Figure 5.7: context data model

a reference to the originating *Context* class. When a change to the values of a property occurs, the *processContext* method of the listener is called. In addition to the context reference a *ContextElement* object is passed to the method that identifies the source and property and contains the context information.

On the server side, these classes are used to model client subscriptions to context information. A *ContextListenerInterface* implementation, provided by the *ContextListener* implementation, defines the events that are of interest to a client. Only when a matching event occurs (e.g. a subscription has been made) the server will send a message to that client. There exist currently two implementations of the *ContextListenerInterface*. The *ContextListenerProperties* class defines matching events by a set of given properties, such as valid source names, valid tags, etc. The *ContextListenerScript* class can be used by clients

to define more complex rules by submitting a script to be run on the server side. On each event an appropriate function of the script will be evaluated to see if an event should be transmitted to the client. Using a script can for example perform text comparisons for decisions or use an internal state that influences the outcome of a check. For a detailed overview on the class structure on the server side see figure B.3 in appendix B.

On the client side a simple class structure provides methods to communicate with the server side and convenience implementation for dealing with context information. At its core it implements the used protocol and provides a callback functionality to react to context information sent from the server. The diagram in figure 5.8 shows a simplified overview on the involved classes. See figure B.4 for a detailed version.

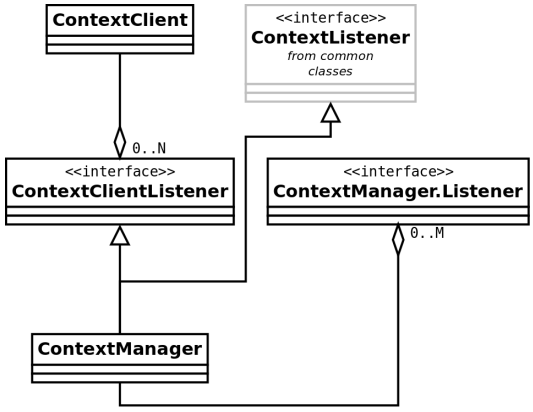

Figure 5.8: simplified client model

The *ContextClient* class implements the network protocol for the client side. It is used to connect to a server and can automatically handle connection loss and reconnect to a server without intervention from the application. It provides rudimentary methods for sending commands to the server, such such subscription to context information. It provides a callback mechanism to notify the application of incoming messages. The *ContextListener* interface is used on the client side to provide a needed abstraction from the raw protocol. The *ContextClient* class will create appropriate *Context* instances an communicate

context information with *ContextElement* objects. For more high level handling of context information the *ContextManager* class provides more functionality. The class itself implements the *ContextClient* interface and uses the provided low level information to update an internal *PassiveEnvironment* model. It can make use of the unique identifiers in requests to and updates from the server reducing the needed bandwidth (in contrast to using names). See section 5.4.5 for more information on this feature.

The *ContextManager* is meant to be used by context aware applications. It provides convenience methods to request and set properties and performs other housekeeping tasks that allow the developer to focus on the application logic instead of dealing with the framework.

5.4.3 Differences between TCF and CTK

The representation of context information in TCF is very different from the system used in CTK. While TCF demands a *context-source-property* structure for information a context widget in CTK can freely choose a structured abstract data type with named fields to transmit information. On the one hand, this enables the widget designer to model physical or logical structures of observed entities. On the other hand, this introduces a overhead when modelling simple properties, such as sensors that only provide one reading (e.g. a temperature sensor or a simple switch). Depending on the application domain the simple system of TCF can either ease the design of an appropriate information representation or could introduce the need for inconvenient workarounds.

Another conceptual difference between the approach here and the CTK is the use of a centralized server for receiving and distributing context information. The main reason for this decision is the difficulty of discovering devices in a peer-to-peer network as used in CTK. If an application using CTK wants to access a source of context information it *broadcasts* a message on the network asking for matching sources. The sources or a discoverer answer this message automatically which allows to deploy new sensors simply by connecting them to the network. While this is a very elegant solution it does not work in most mobile and wearable settings since *broadcasting* only works inside a single network (e.g. as found in home automation) and is not supported by cellular networks

in general. Even in an industrial setting this technique will most likely not work as wireless- and wired LANs are often technically different networks. A possible solution for this scenario is the use of a centralized server where each component registers itself and can query this server for information on how to contact other components to engage in a direct communication. This kind of approach is for example present in instant messengers where each participant is logged in at a central server and can be found by others.

In addition to the discovery problem there is another issue again related to network technology. If each component engages in peer-to-peer communication they need to be servers per definition and therefore require permissions in the firewall protecting the network. This potentially puts a huge burden on the IT workforce or requires tricks on the side of the components with help from a centralized server such as hole punching[FSK05] which may or may not be acceptable for a network depending on IT policies. With a centralized server approach only one device needs special treatment regarding the network configuration but can also be controlled more easily in terms of network security. Further benefits of a centralized system are additional services that can be provided such as storing sensor data for later use, data filtering and the generation of various statistics on the processed data.

5.4.4 Code Metrics

The framework follows a very simple design principle that allows developers to quickly create applications without the need to implement a specific model for the context information they intend to use. Since the framework is separated into a server and a client side the implementation follows the same division by creating packages that are specific for each side. A third part of the framework emerges from the need of equal functionality on both sides. These components are placed in a common package. This separation of code into packages and a set of common functionality leads to a development approach that limits the instability of the system as a whole. Changes to the server side do not require changes in the clients as long as no modification is needed in the set of common components.

To analyze the components in a more objective way a set of metrics has been generated similar to the findings in section 4.1.6. When analyzing the components common to both sides only a very flat structure is found. It consists of 32 classes in two packages that define the behaviour of the communication protocol and the needed classes and interfaces to model context information. This structure can be seen in figure 5.9.

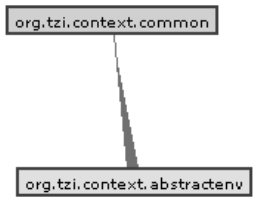

Figure 5.9: dependency graph of common package

Computing the instability metrics for the common classes results in a low value of 0.062 (± 0.062). The communication related functionality has a value of 0 from an efferent coupling of 0 and an afferent coupling of 32. For the context information model a value of 0.125 results from a coupling of 1 (efferent) and 7 (afferent). The value of 0 for the first set of classes shows a desired property of a common set of functionality as it has no external dependencies. Looking at the complexity metrics shows an average value of 2.642 (± 2.642) with a maximum value of 18. The highest valued method is found in a class dealing with the definition of listeners for context information that is mostly used on the server side. Other high values methods are related to the implementation of the network protocol and the representation of abstract data types in the communication.

The server part of the framework consists of 47 classes in two packages that are roughly split into classes for the available commands and classes that deal with the management of context data. This structure can be seen in figure 5.10 that also shows the dependencies on the common set of functionality. Looking at this graphical representation a circular dependency between the server and the commands is visible. This results from the nature of the provided commands

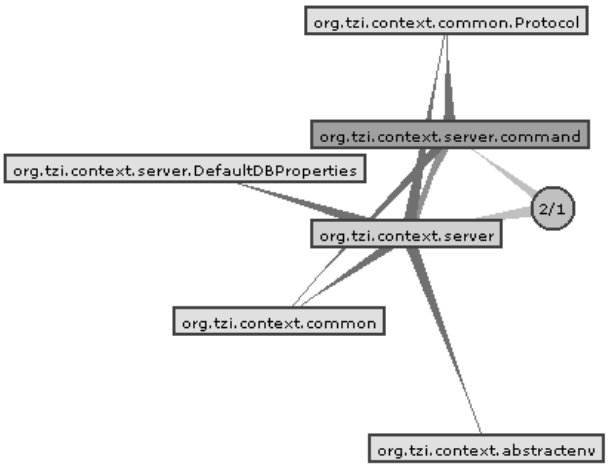

Figure 5.10: dependency graph of server package

that directly manipulate the state of the server and are in turn called by the server process.

While application developers will typically focus on the creation of applications that make use of the client side looking at the instability metrics can provide some insight on the design of the server side. A value of 0.696 ($\pm$0.251) has been computed for the server as a whole were the command package has an individual instability of 0.947 (afferent coupling of 1, efferent coupling of 18) and the control classes score 0.444 (afferent coupling 10, efferent coupling 8). This reflects the circular dependency of command and control structures that have to change simultaneously.

Looking at the complexity metrics reveals an average value of 4.313 ($\pm$ 7.638) with a maximum value of 75. This very high value is found for the command that deals with handling subscription requests from a client and is also the complexity of the main server loop. Other high values are also found for other commands and for functionality that deals with storing and retrieving contextual information from database systems. In general, code from the server side of the framework tends to be more complex but on the other hand this is of no

concern to the application developer.

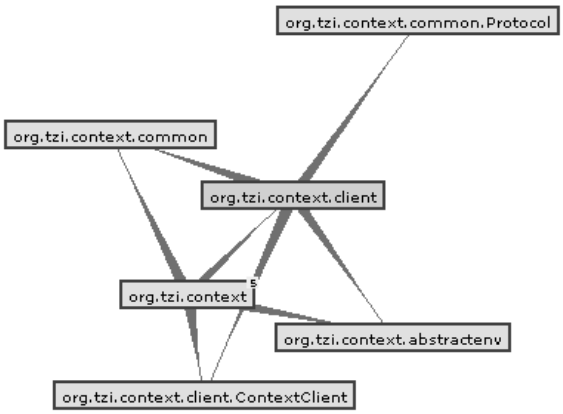

Figure 5.11: dependency graph of client package

The client part of the framework consists of separate classes for the client implementation and for demonstration classes. There are 17 classes for the client code that provide functionality to communicate with the server part. The demonstration code has an additional 32 classes that can be used as examples for implementation. The structure of dependencies, including dependencies on the common classes, can be seen in figure 5.11. The package *org.tzi.context* in this figure contains the demonstration code while the *org.tzi.context.client* package represents the implementation of the actual client side of the framework. No cyclic dependencies are present but one can clearly see that the *context.client* package and base *context* package form two clusters in the graph with similar dependencies. This reflects separation of the demonstration code.

The dependency graph can be shifted to the demonstration package that is more close to what an actual developer would experience with the framework. This rearrangement can be seen in figure 5.12. When looking at the resulting structure a desirable configuration can be seen where the demonstration code has clear dependencies to other packages on the client side and these packages have only a few other dependencies. This is an indicator for a suitable separation of concerns.

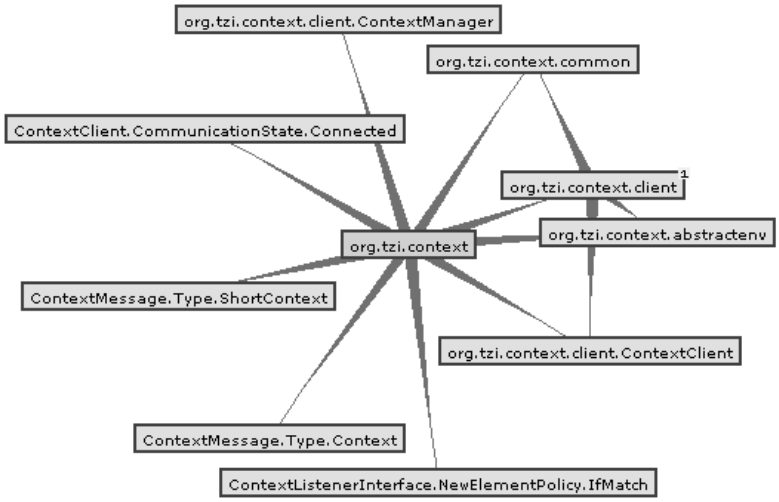

Figure 5.12: dependency graph of client package, context view

Instability metrics for the client side have been computed as an average of 0.583 (± 0.417). This includes the demonstration code. When computed separately the demonstration code shows an instability of 1 (afferent coupling 0, efferent coupling 5). This result is only natural as the demonstration code does not contribute to the framework and only uses its functionality. The instability for the actual client side is 0.167 from an afferent coupling of 15 and an efferent coupling of 3. This low instability reflects the graphical representation that shows very independent packages of functionality.

The complexity metrics for the client side could only be computed including the demonstration code. A average value of 3.489 (± 7.976) with a maximum of 70 is found. The highest value results from the main loop of the communication thread inside the client class. This method is not used by an application developer but handles receiving and sending data from and to the server. Other high values are found in demonstration code or other internal methods that handle different aspects of communication.

5.4.5 Communication Protocol

The following sections will provide an overview on the most important aspects of the implemented protocol in the context distribution framework. A complete protocol reference can be found in appendix A. The protocol used in communication between server and client is based on transmitting messages as lines of text. The end of a line indicates a complete message that is then processed by the corresponding communication partner. However, the protocol defines a maximum length of each line that serves as a protection against a malfunctioning client (or attack).

In general, clients send a command to the server and wait for a reply. When a client subscribes to context information, as an exception from this, the server will automatically send this information to clients as a special message. To help making messages platform independent their content is always represented as UTF-8[UTF03] encoded text and numbers use a decimal representation. Individual parts of the message are URL-encoded[URL05] before transmission. This encoding step allows the use of spaces in items without making the parsing more complicate (as spaces are replaced by other tokens) and also wraps all characters into a sequence of pure ASCII characters for a very robust transmission process. In many cases, where the message content only contains valid ASCII characters and no spaces the encoding step has no effect on the size of a message when sent over a network.

All messages have a similar structure as depicted in figure 5.13.

[#Prefix]	Command/Reply	Argument1	Argument2	...	ArgumentN

Figure 5.13: message structure

The *prefix* of a message is an optional part and can be used by clients to organize asynchronous communication. If a client starts a message with a sequence of a hash symbol (#) followed by one or more characters the server will use the same prefix for the reply to the command. By using this protocol feature clients can associate replies to their commands without keeping track of the order of

replies. The prefix feature is available for all messages and does not affect their meaning in any way. Each message contains at least one item that identifies the command to be executed or the type of reply. After this identifier any amount of arguments can follow.

While many different commands exist in the protocol the efficient transmission of context information is of highest importance. A sensor can potentially produce a lot of information in short time that needs to be transferred first to the server and then to the client. While the generation of context information from such a sensor is handled by a specific command message the resulting value updates sent to subscribers have a different format. As portrayed in section 5.4.2 the *ContextElement* abstract data type is used on the client side to encapsulate context information. The representation used in the protocol follows roughly this data type and results in the structure of figure 5.14.

Source	Property	NT	Tag [Tag[...]]	Value	TS	[P]

Figure 5.14: context data structure (transmission)

The *Source*, *Property* and *Value* items in this structure directly correspond to the fields in the abstract data type. The element denoted by *TS* contains the time-stamp as an integer number (in millisecond resolution). The *NT* element is an integer number that counts the number of *tags* for this element. It is directly followed by the specified amount of tags. A value of zero means that no tags are present. The last element in this representation is an optional indicator of the type of information. If the character *P* is present, the information has a persistent character. If not, it is treated as transient, which is the default case.

This representation is missing an identification of the associated context in the same way it is missing in the *ContextElement* model. For an actual transmission over the network, this structure is embedded in a *ContextMessage* class (shown in figure B.2). This class is not used in the internal model but serves as a translation aid from the network protocol into the internal model and reverse. When a context message is constructed the correct context and a *ContextElement* instance is used to form a message containing all needed information. This class

can also create messages that inform interested clients of structural changes to the environment such as added sources or properties. There exist also special modes of operation that allow compacting a context message further by only referencing the numeric id of the property in question. See section 5.4.6 for details. A message for transferring context information in the protocol follows the structure seen in figure 5.15.

Figure 5.15: context message

The command *CTX* identifies the message type and tells the client how to process the arguments. The *Id* field contains a unique number that identifies the subscription made by the client. This enables a client to quickly sort incoming messages from different subscriptions if needed. The *Context* field contains the name of the context where the information originated. It can be optionally followed by the numeric identifier of the context on the server side. Clients can check if the context name contains a semicolon and split this field accordingly.[2] After the identification of the subscription and the context, the actual information as shown in figure 5.14 is appended.

These context messages are sent asynchronously to clients by the server and can be easily read out to form a *ContextElement* structure with an associated context. The elements in the message are structured in a way that makes writing a parser for the format very simple and resource friendly. All items in a message are guaranteed to contain no white-space[3] and can therefore be extracted by splitting up a message whenever a white-space character occurs. Alternatively, clients can choose to simply read a word until white-space is found. After the command or reply type is identified the appropriate further processing option can be chosen, e.g. a context message can be processed according to its expected structure. Reading a context element follows the same simple parsing

[2]This additional information was retrofitted in the framework at a later state in the project and needed to be attached in a way that did not change the used protocol. In the future, this kind of information would require its own field.

[3]White-space refers to a set of invisible characters that contribute to horizontal space such as spaces and tabulators

approach where the names of the source and property are the first two items. The third item is the count of contained tags. Knowing this value in advance allows a client to allocate needed structures before going on in the parsing process making later adjustments unneeded. This very simple approach is used in all commands and allows constructing parsers without the need for looking ahead. The benefits of such an approach are extensively explained in [GJ90].

A client communicates with the server over a single TCP/IP connection. The default port for communication is 2009 (decimal) but can be defined on startup of the server. The communication starts with a short message from the server that can be used by the client to verify talking to the correct server and also contains the version of the protocol used by the server.

There are three different forms of communication between server and client. A client can send a query or property change to the server to which a response is generated indicating the success or failure of the operation and providing the requested information (if any). In case of a context event to which the client has made a subscription, the server will send the context message as outlined previously. As a third form of communication, either server or client can be engaged in the transfer of a message that was too long to transmit in a single operation. These large transfers are explained in section 5.4.7. After a transfer is complete the merged message is processed as if received in one piece. This last form of communication is transparent to the logical protocol and transfer handling is automatically facilitated by the implementation. Similar to the asynchronous context messages, the server will also perform periodic checks to see if a client is still participating in communication. It will send a **PING** message to clients if they have not send any events in a defined period of time. Non responding clients will be removed from the server to free up resources. This ensures long time usability of the server.

Figure 5.16 shows a flowchart of the main server loop. As mentioned before, the communication between the server and its client is stateful to allow a compact protocol. Before the actual communication protocol is used clients have to participate in a simple login procedure where the server transmit the used version of the protocol and the unique id of the client. In this initial communication

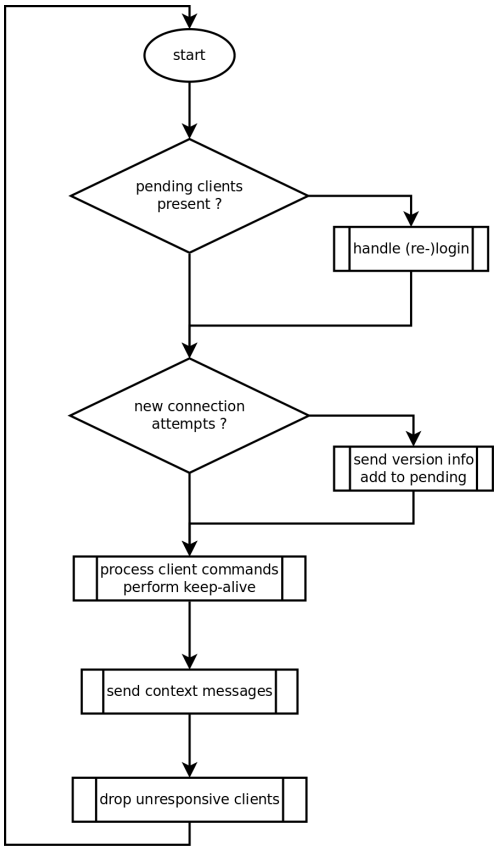

Figure 5.16: main server loop

the client can also provide a name for itself or re-claim a connection that was lost before.

The server will send a **DROP** message if the login procedure is erroneous or if the client does not respond any more. If a **DROP** event is received by the client the server will have closed the connection.

After successfully establishing a login each unknown or erroneous message sent by the client will cause a **FAIL** response from the server but will not cause the termination of the connection.

Receiving Context Information

In order to receive context information from the server a client needs to make subscribe to the information it is interested in. These subscriptions can be based on the names of the elements in the environment and/or on the existence of specific tags. There also exists a more complex type of subscription via server-side scripts that is discussed in section 5.4.10.

A standard subscription is made by sending a **SUBSCRIBE** message to the server that defines a number of matching items. The general format of this command is seen in figure 5.17 and allows a client to subscribe to different items in one call. The field *NM* is the number of *MatchDefinition*s that are present. *MatchDefinition*s declare matches inside a context by providing a number of

SUBSCRIBE	NM	MatchDefinition	...

Figure 5.17: subscribe command

*SourceMatchDefinition*s as shown in figure 5.18 where the field *NS* represents the number of these sub-definitions. Following this pattern, the *SourceMatchDefinition* entries outlined in figure 5.19 define a number of matching properties with *PropertyMatchDefinition*s (with the *NP* field counting their number). These

Context	NS	SourceMatchDefinition	...

Figure 5.18: subscribe command - context definition

innermost elements define matching properties by their name and potentially a number of tags that need to be present. Figure 5.20 shows this structure where the field *NT* is the number of tags. By convention, the absence of tags is interpreted as matching any tag.

There are two options when referencing the elements in the environment that

Source	NP	PropertyMatchDefinition	...

Figure 5.19: subscribe command - source definition

are used in the protocol. The first option is to use the name of an item. This is

Property	NT	Tag	...

Figure 5.20: subscribe command - property definition

indicated by prepending the name with an @ symbol in a message. Named items are not necessarily unique, e.g. there might be several sources with the same name in different contexts. When named items are used, the server will use the given information to look up the corresponding item in its internal state. The second option for referencing items is using their assigned numeric identifier. This is done by using its decimal representation in the respective field. Making use of identifiers allows the server to perform a very quick lookup as it keeps a global table containing a mapping from numeric identifiers to the corresponding object.

If a client only wants to subscribe to one specific property with one matching tag a message similar to figure 5.21 would be sent.

SUBSCRIBE	1	@cname	1	@sname	1	@pname	1	tname

Figure 5.21: subscribe command - example

If a client knows the numeric identifier of a property in question, the message for subscription can be shortened. If the context field contains only the letter 'P', the client indicates that it will not specify a *SourceDefinition* but skip directly to defining a single *PropertyDefinition* where the numeric identifier is used. The server will then use its lookup table to identify the associated context and source items. Such a message is depicted in figure 5.22. A similar shortcut exists for sources that can be referenced by numeric identifier to avoid sending the name of the context.

SUBSCRIBE	1	P	1234	1	tname

Figure 5.22: subscribe command - numeric id example

Clients might not always know the actual name or identifier of a source of context but might rather use specific sets of tags for identification. For this case, special *wildcard* names exists that instruct the server to match any name. The special

constants *<AllContexts>*, *<AllSources>*, *<AllProperties>* and *<AllTags>* can be used in the place of the corresponding identifier in a subscribe message. These special identifiers can be freely mixed with others to create the needed behaviour of context messages. A client could for example instruct the server to send all properties with the name *prop* from all available sources in a given context. Such a message is shown in figure 5.23.

SUBSCRIBE	1	@cname	1	<AllSources>	1	@prop	0

Figure 5.23: subscribe command - wildcard example

When the server has successfully processed the subscribe message it will send a reply message containing the unique identifier of this subscription. This identifier will be send along with any matching context message.

5.4.6 Short Context Mode

While the protocol in its simplicity can be easily read and even written by human observers the constant transmission of the same identifying names for contexts, sources and properties increases the volume of data and therefore limits the speed of transmission over any medium.

As described in the previous sections, the server assigns unique numeric identifiers to each known item and can transmit these values to the client. These identifiers can then be used in place of names or allow shortening any request by leaving out redundant information. While shortening of requests can be useful on its own the greatest amount of messages in a typical system will be context messages sent from the server to the client. Each of these messages normally contains information on the context, source and property origins of the event. When a client already knows all these details for a given property it can change the message structure to exclude them. This mode of operation is the *Short Context Mode* that can be enabled for each subscription individually. When activated the server will not send normal CTX messages but SCTX messages to make the different data formats easily distinguishable for the client. The SCTX messages contain the same information for reconstructing a *ContextElement* object but leave out the names of the source and property. This arrangement is

shown in figure 5.24 where the field *PropertyId* represents the numeric identifier and is used instead of two fields storing the names of source and property (in comparison to figure 5.14). The resulting structure of a short context message is shown in figure 5.25

PropertyId	NT	Tag [Tag[...]]	Value	TS	[P]

Figure 5.24: short context data structure

SCTX	Context[;Info]	Short-ContextElement-Representation

Figure 5.25: short context message

A client can even enable the short mode before knowing the used identifiers. Upon receiving a *SCTX* message the client can ask the server for the missing information on the property.

5.4.7 Transfer of large datasets

While typical sensors only produce small amounts of data per reading (e.g. numerical readings, boolean states, etc.), support for larger entities such as images, audio data or even complete files containing binary data must be supported to cover potential application needs. However, a problem arises from the architecture of the server. Since messages are processed in a single thread and not with individual threads per client a large and therefore time consuming transfer would block other clients while in progress. A solution to this is splitting a transfer into smaller blocks of data that are transferred sequentially and can therefore be mixed with other commands. While this approach introduces some overhead in data transfer it ensures that large data transfers do not cause the system as a whole to become non-responding. It also allows for a more easy control of data transfer since they can be stopped after each small block if needed. Finally, this concept is also a benefit for the client as it can sent unrelated commands to the server while the transfer is in progress and even transmit multiple large data sets simultaneously if needed.

Conceptually, a large transfer can originate from the server or from the client.

When a client updates a property and the size of the generated message exceeds the maximum message size defined by the protocol the message needs to be send in smaller blocks. To create an easy to parse representation the original message is URL encoded to replace all white-space and special symbols before transmitting. After this message has been transfered to the server, potential listeners will receive the new value in form of a context message that will most likely also be too large to transmit as a normal message. The server will therefore transfer the generated context message in smaller blocks.

A data transfer is performed by sending a **TX** message. This message is used for starting a transfer and also for sending the individual blocks. To initialize a new transfer, the first message contains a numeric id for the transfer, a block number of 0, the total size of the data and the first part of the actual data. The receiving end responds to each **TX** message by sending a **TXACK** message with the transfer id and the number of the block. In case of a missing block, the receiving side can also send a **TXRESEND** message with the id and the number of the missing block.
A transfer can also be cancelled by sending a **TXCANCEL** message with the corresponding transfer id.
After a complete transfer, the received data is URL decoded and processed as if received as a normal message.

As a special case, a context message is transmitted with a different message. If the message that initializes the transfer uses **TXCTX** as the command (instead **TX**) a client knows in advance that the message will contain context information. This feature is used to ease message parsing for clients but has no further benefit for the protocol.

Since the transfer of data blocks is marked with a transfer identifier and a block number, several transfers can take place in parallel and other messages can be sent in between. While the framework technically supports transfers of data using the described mechanisms it is not very efficient. If large datasets often need to be transferred in a specific setup it might be a more feasible solution to increase the maximum allowed size for messages.

5.4.8 History Queries

For various reasons a client may be interested in the retrieval of older context information. Examples include reconstructing the state of a model upon connection (trends, travel, etc.), recovery after unplanned disconnection from the server or a punctual need for data at the client side.

The TZI Context Framework supports this kind of history access through an abstraction layer that makes the actual type of storage transparent to the server and the client. The history function was initially engineered to store context information into a SQL database (making use of Javas JDBC concept). It uses a flexible SQL adapter that can support different SQL dialects such as MySQL, MS-SQL, SQLite and DerbyDB. There exists also an implementation of a volatile history that is only stored in the memory of the server. This history is lost when the server is stopped but depending on the task can provide an alternative functionality without further infrastructure.

The history functionality is an optional part of the framework and the actual class performing the storage and retrieval operations can be specified by the user. By extending the basic abstraction class additional mechanism for history functionality can be implemented by third parties. The need for extending a base class instead of providing an interface to be implemented for history functionality is a result of technical dependencies by the server. The class provides methods for handling properties that serve as an abstraction layer between the server and the backend data storage. Changes to most of the methods are unlikely (but possible) and therefore the extension from the class allows a quick implementation of new storage methods. Figure 5.26 shows a simplified overview of the class hierarchy where only selected attributes are shown. The abstract base class maintains a basic environment for context item handling that can be used by subclasses and defines methods to be implemented for storing data. The *HistoryDB* class performs the actual queries using a SQL connection while the *MemoryHistory* class stores a (limited) set of old values inside the working memory.

Clients can access the previous state of properties through the **HISTORY** command. This command is used by the client to form a query that specifies

the context, source and property in question and also a range of time and a limit for returned values. Optionally, a list of tags can be specified so that only property values that contain at least one of these tags are returned. Matching values should be returned in the same order as they where received by the server but may be sorted differently by the used history implementation. In any case, each returned value carries its time-stamp. The result of a history query is a list of short context formatted context elements but also leaving out the identifier (see section 5.4.6). In this list, each individual item has been URL encoded. The list is prefixed with a decimal number equal to the number of elements. The empty list is therefore returned as the string *0*.

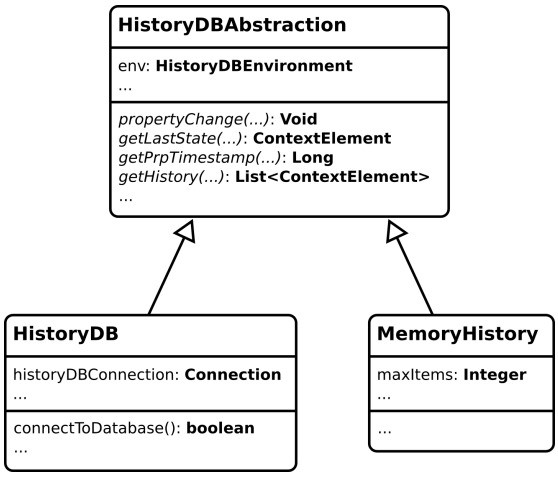

Figure 5.26: simplified history class relations

The *ContextManager* (see section 5.4.2) class provides convenience methods for requesting history information and makes the results available. When the server component does not support history functionality the history requests will fail. In contrast to the approach in CTK where each individual widget is responsible for storing previous values the centralized approach in this framework shifts this responsibility to the server component. This makes the maintenance of this information easier as only one connection to a suitable back-end has to be specified.

5.4.9 Storing Context Histories in Relational Databases

As mentioned in the previous section, an implementation for storing context histories into a database was created. A naive approach for storage would consists of simply inserting every event into a suitable database table as they are received at the server. This would however result in a lot of repetitive information if the names of the context, the sources and the properties are stored for every event.

A more sophisticated strategy is used here that exploits features present in existing database systems like MySQL and others. Similar to the hierarchical structure formed by contexts, sources and properties, a table is created in the database that stores the name of each context and assigns a unique numerical identifier to it. For each context a table of sources is created that contains the names and identifiers of all associated sources. Finally, for each source, a corresponding table of properties with their unique identifiers is created.
Property changes are stored in a table for each context where the source and property are referenced by the numeric identifier. While this approach minimizes the amount of data used to identify sources and properties it also speeds up searching for specific history elements as the database can use numeric comparison instead of matching names.

One important part of history storage is the handling of removed items. Properties, sources and contexts can be removed from the server to signal that the associated information is no longer available. To keep track of this information items need to be marked as active or inactive inside the history. Making this information available to the server is important as it will be used when the server restarts to reconstruct the state of all properties but only for those that are still present in the contexts. The current implementation stores this information inside the tables that assign unique ids to all items as a boolean value.

In figure 5.27 an example for the table structure is shown. The table *contexts* is used to look up existing contexts while all other table names are inferred by the unique ids assigned by the database.

Currently, tag values are stored as a comma separated string in the database

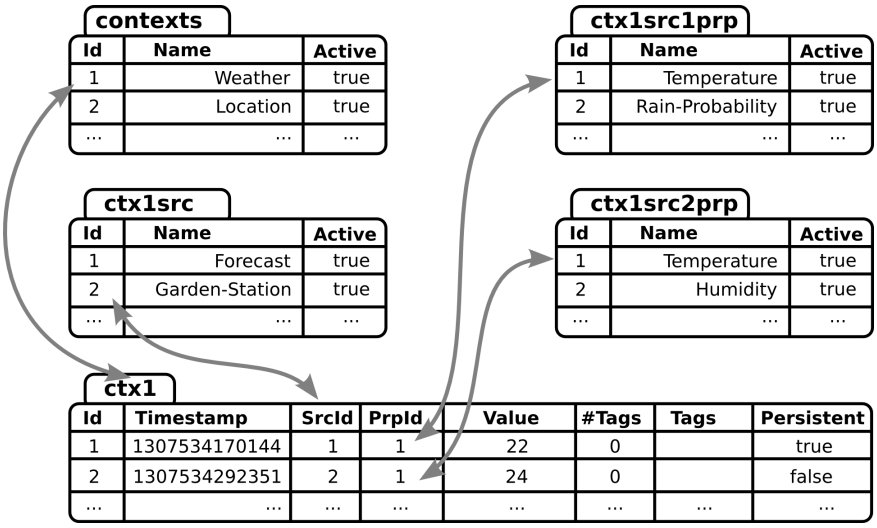

Figure 5.27: history database tables

tables. Depending on the specific use, creating a separate table that assigns unique ids to individual tags and then only saving a list of these numbers could improve storage performance and also speed up queries for matching tags by applying table joining techniques.

Database Queries

The history implementation keeps track of all unique identifiers assigned by the database and uses them for value lookups. This allows for efficient value queries and also serves as a bridge between the client queries that either specify the property by name or by the unique id from the server (which is different from the database id). If for example the context in question has the database id 2, the source has id 1 and the property has id 3, a query may be formed like this:

SELECT (value) FROM ctx2 WHERE source=1 AND property=3

Of course further SQL statements can be used to restrict the range of timestamps in returned results to match the specified query for the history database.

Storing Large Values in a DataBase

While the normal use of context information is supposed to transfer only small values that can be easily stored inside the database larger values are processed differently by the implementation. This is similar to the handling of large transfers described in section 5.4.7

Most SQL-like databases support the storage of large binary objects (so called BLOBs) using a special syntax. These objects use a special datatype in the table and need a more elaborate handling process for storage and retrieval. Since the usage of large values is expected to be an exception the table normally used for storing context events does not contain a field for a BLOB. Instead, a global table is used to store BLOBs that assigns a unique numeric identifier to them. This identifier is stored in a recognizable form instead of the value in the table. Some care has to be taken to modify values that would match an identifier before saving. The current implementation simply saves a large object identifier as a hash-symbol ('#') directly followed by a decimal representation of the numeric id. If a normal value starts with the hash symbol, a second hash-symbol is prepended to the value. When a value is to be retrieved from the database, it is therefore easy to determine if the value actually refers to a large object that can then be retrieved from the global table. See figure 5.28 for an illustration.

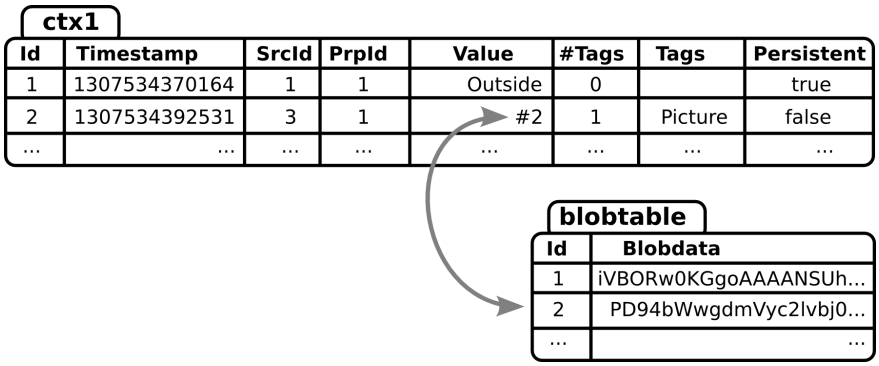

Figure 5.28: history database BLOB access

5.4.10 Scripted Subscription

The method of subscribing to context information shown in section 5.4.5 allows an application to receive information updates from entities that are known by name or by specific tag information. For more specific control over the matching process a client can deploy small scripts on the server that contain algorithms to determine if a context event is of interest to the client.

The **SUBSCRIPT** command is used instead of the **SUBSCRIBE** command to upload a script written in the ECMAScript[ECM14] language (commonly known as *JavaScript*) to the server. This script provides predefined functions that are evaluated on the server to decide if a context information event should be transferred to the client or not. Also the notifications on new context, source and property entities can be controlled by this.

The **SUBSCRIPT** message has two arguments, the context for the script (that can also be the special value *<AllContexts>*) and the script itself. As with all messages, the actual script data is transmitted as an URL encoded argument. The script is then evaluated on the server and any errors in the script will lead to a failure reply from the server.

On each context event, an appropriate method from the script is called. For context update events, the method *matches* is called with the list of tags, the name of the source and property, the value, time-stamp and persistence information. Only if the method returns *true*, the event is sent to the client. In addition to these methods, there are also methods for the notification of structural changes. The method *notifyNewSource* is called when new sources are added (with the name of the source as parameter) and *notifyNewProperty* when a new property is defined (with source and property as arguments). Similar for the removal of sources and properties the methods *matchesSource* and *matchesSourceProperty* are evaluated. Structural notifications are only sent when the methods return *true*. There are also methods that define if the client has any interest in structural changes at all. The methods *getNewSourcePolicy* and *getNewPropertyPolicy* can return a string from the set *always*, *never* or *match* to indicate the desired behaviour. Only when matching is selected, the evaluation methods for structural changes are used.

Using scripts is currently an experimental feature that has not been tested extensively. The core idea is to allow clients to formulate their requests in terms of an algorithm in order to reduce the amount of information that is transmitted to them. The server component requires a suitable scripting engine to process the scripts. Most Java environments provide a suitable engine or can be setup to integrate an external engine such as the Mozilla Rhino[Rhi14] project.

Chapter 6

Evaluation of the TZI Context Framework

6.1 Comparing CTK to TCF

The main difference between CTK and TCF is the choice of a centralized approach in favor of decentralized communication. A direct comparison needs to focus on the effects on a potential developer and also on the amount of data transmitted between components of an application. While CTK was designed to transfer knowledge from the domain of traditional UI development to the domain of context aware applications the main focus of TCF is providing a resource-efficient way of context distribution.

This section provides a technical comparison in terms of code metrics, implementation differences and transmission behaviour. A simple application is implemented using both frameworks and the resulting code is analyzed for its complexity. To measure data transmission characteristics, two additional applications are created that allow performing transmission tests with varying parameters.

6.1.1 Application Complexity

A simple application was constructed to compare both frameworks. It consists of a context aware application and a corresponding information provider. Temperature monitoring was chosen as a simple to understand example for this task where the context aware application simply monitors a temperature sensor. No further action is performed other than showing the value. In this case the provider of information is a piece of software that interfaces the actual sensor and makes the reading available to the monitoring application. In order to reduce the complexity of the example the temperature sensor is simulated by a UI element where the user can simply adjust the temperature value. Figure 6.1 shows a simplified UML diagram of this exemplary application where the *TempProvider* class represents either the *sensor* for CTK or TCF and *TempReader* implements the monitoring application.

The design of this application is modular to limit the coupling between the application and the frameworks. While the *Provider* and *Reader* classes are created specially for each framework the *TempUI* class that controls the slider element and the *TempReadUI* class that displays the temperature are identical. A complete listing of the code can be found in appendix C, section C.3 and C.4.

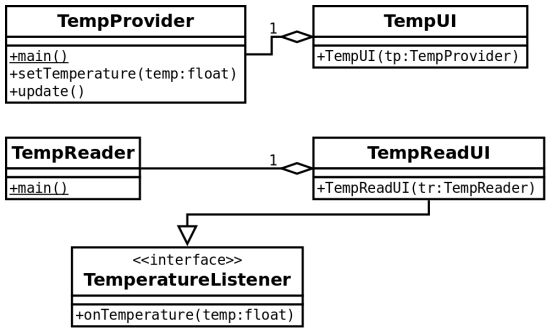

Figure 6.1: architecture of the temperature demo

In both cases, an interface as shown in figure 6.2 is presented to the user and a change to the slider is immediately reflected in the display. While the user interface is the same for each framework there is a huge technical difference that is not visible to the application developer. In case of CTK, two applications are

started that use the framework to register themselves. The monitoring application will query the framework for a temperature provider and then register at the providing application for updates. For the TCF approach, three applications are running. In addition to the two applications that provide and monitor information the central server is running as a third component. Both applications connect to the server and the temperature provider transmits updates when the slider changes. The monitoring application does not query for a providing component but simply subscribes at the server for matching information. The server will then transmit temperature values on each change to the monitoring application.

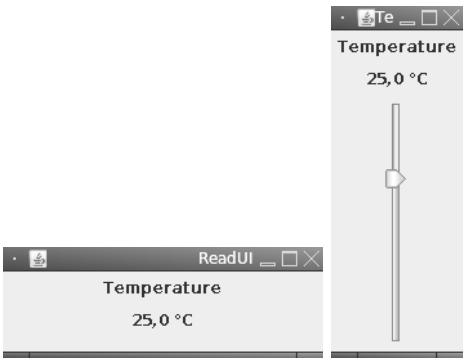

Figure 6.2: temperature application UI

Even though both applications are very simple they can be used to get an initial overview of the complexity of the frameworks. A visual representation of package dependencies and standard code metrics were generated with the *Metrics2*[Met14] plugin for the *Eclipse IDE*[Ecl14]. While the visual representation of dependencies can give a quick view on the complexity of an application a complex graph does not necessarily indicate bad design. It reflects however how many aspect of a framework need to be understood by the application developer. On the other side, a simple graph can also be an indicator of bad software design where a beneficial separation of concerns has not been done. In combination with other code metrics a more formal comparison is possible. A common expression for code maintainability is the *cyclomatic complexity* as defined by [McC76]. This value represents the number of possible control paths in a method as defined by the used control structures. A high value is an indicator

for a complex method as it requires more testing to ensure correct behaviour. However, there exist examples where methods with high cyclomatic complexity are actually trivial to understand for developers. The idea behind defining complexity measures is to set self defined limits on code. If a method becomes too complex it is split up into smaller, less complex methods. This prevents the existence of large methods that are hard to test and more prone to errors. Also, a reduced complexity generally results in a behaviour easier to document and therefore to understand. The cyclomatic complexity measurement can not be used to directly compare two given methods since it does not reflect the reasons for complexity. It can however be used to identify methods with a relatively high complexity among related methods. If these methods are to be used by other developers they need to be documented well otherwise leading to confusion.

Analysis of the CTK approach

Using CTK the temperature provider is implemented as a *Widget* that provides a *temperature* attribute and a callback function that allows other components to be notified of changes to this value. When the user changes the temperature using the slider the temperature is set using the defined *setTemperature* method (see figure 6.1) and then *update* is called. This in turn creates an *Attributes* object containing the temperature value that is transmitted to all subscribers of this object using methods from the *Widget* class.

The temperature monitor is implemented as a *BaseObject* instance that also implements the *Handler* interface. It inherits therefore the communication functionality from the *BaseObject* class but does not provide any information by itself. The *Handler* interface provides methods to receive updates from other components. Upon starting the application, the monitor sends a description of the required context information to the discovery service of the framework. This is done by creating an *AbstractQueryItem* that searches for an attribute with the name of *temperature* and a numeric type. It uses the first matching component that contains such an attribute and requests updates on the value by using a *ClientSideSubscriber* object. This object encapsulates information on the *Widget* that provides that data and the callback method to use. A method from the *BaseObject* class is used to set up receiving updates via the *Handler* interface. The result of this setup is a subscription identifier that can be used to

distinguish several senders. When an update arrives, the defined *handle* method is called with the subscription identifier and an abstract *DataObject*. The aforementioned identifier can be used to process data from several sources. The *DataObject* has to be inspected by the application to extract the transmitted information as it can store complex data types.

Generating a dependency graph for the package containing both applications (*ctktest*) results in figure 6.3. The central placement of the application package and the star pattern of dependencies indicates that all used packages are independent of each other. This is an indicator for a good application design in the framework. However, many packages are in use even for a very simple application which indicates that a lot of functionality is very common for all types of applications.

Looking at the cyclomatic complexity values for the classes and methods of the application, the methods of the monitor have a relatively high value. While the overall complexity is 2.09±2.99, the method for extracting the temperature value from the *DataObject* has a value of 15. The most complex method of the provider has a complexity value of 3 and deals with sending a new value to the subscribers. This reflects the experience in the implementation phase. While it was very easy to create the *Widget* for providing temperature information, creating the receiving side was far more complex.

Analysis of the TCF approach

Using TCF the temperature provider is a simple abstract data type that encapsulates the temperature information and an instance of the *ContextManager* and the *ContextClient* class. When the application starts the client is instructed to connect to the central server. When the user changes the temperature with the slider a new temperature is set using the defined *setTemperature* method (see figure 6.1) and then *update* is called. The update method transmits the new temperature value to the server using methods from the *ContextManager*. In addition to the temperature value, each update is *tagged* with the annotation *temperature* to allow clients to subscribe by this information.

The temperature monitor is constructed similar to the provider as it also en-

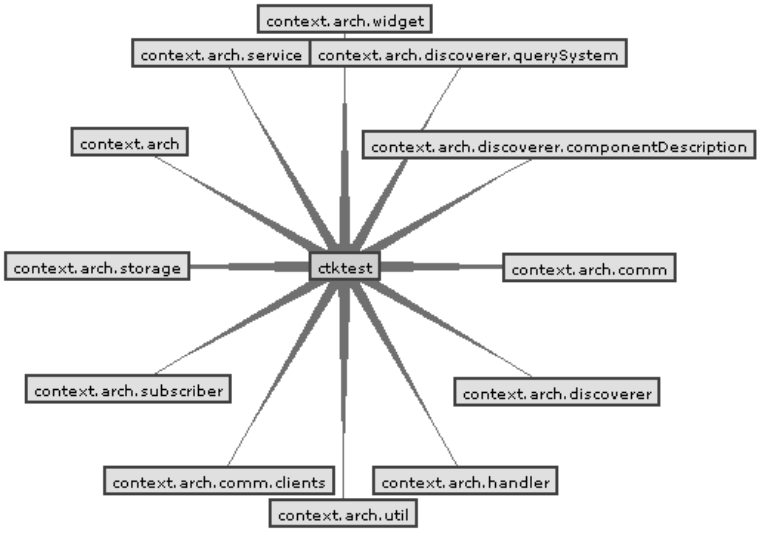

Figure 6.3: package dependencies CTK (temperature)

capsulates an instance of the *ContextManager* and the *ContextClient* class. Additionally it implements the *ContextListener* interface to process contextual information.
Upon startup the monitor subscribes to all context events that are *tagged* with the *temperature* annotation, matching the behaviour of the provider. When context information arrives the *processContext* method (from the *ContextClient* interface) is called with the name of the context of origin and a *ContextElement* object. The application then tries to interpret the value component of the *ContextElement* as a numeric value and updates the temperature display.

The dependency graph for the package of both applications is very simple as can be seen in figure 6.4. As with the CTK approach, the application package is central and other dependencies form a star pattern around it, showing no inter-package dependencies. In contrast to the CTK dependencies, only three external packages are referenced by the application (technically, only two packages are referenced as the third reference is a class from a referenced package). While a small dependency graph is not an indicator of good package design it shows that functionality for a simple client can be accessed by very few refer-

ences.

Looking at the cyclomatic complexity values for the classes and methods of the application, the methods of the monitor have the highest values. While the overall complexity is 1.09±0.29, the method for extracting the temperature value from the *ContextElememt* has a value of 2. The most complex method of the provider cannot be defined as all methods have the minimal value of 1. The only other method that has complexity value higher than 1 is the notification of the user interface after the temperature update. This method has a complexity value of 2. The very low complexity on both parts of the application is an indicator for a simple to use system. By design, the almost identical structure of context providers and receivers allow a quick implementation of both sides.

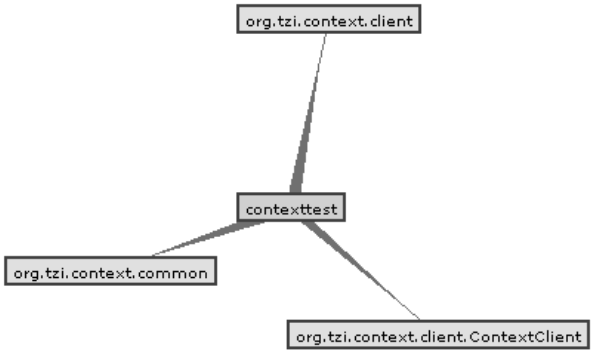

Figure 6.4: package dependencies TCF (temperature)

6.1.2 Transmission Performance and Efficiency

The Context ToolKit was benchmarked against the TZI Context Framework to get a better understanding of the advantages and disadvantages of each system in terms of transmission performance and bandwidth requirements. The general outcome of the experiments was that the Context Framework is most suitable for scenarios were only very few values are transmitted from each sender. For larger groups of values the data structures used by CTK are more efficient and result in less volume of transferred data. The overhead with CTKs protocol is however very large for small value sets where the actual data is only a small

fraction of the transmitted data.

All tests were performed on a local computer system where the bandwidth for transmitting data was virtually limitless. In this scenario CTK is clearly the fasted framework for transmitting data with almost no decrease in transmission time regardless of the amount of data per value update. When estimating the time needed to transfer the data over a limited connection (e.g. UMTS/3G) the transfer time using TCF instead of CTK can however be significantly lower for small value sets.

Test-Concept

Both frameworks have concepts to group a set of attributes. In CTK, each context widget defines its set of values that are transmitted. In TCF, a set of properties is associated with a source. In real world applications the number of attributes per entity will vary but can be assumed to be a small number. Therefore it is interesting to compare the performance of each system for varying number of attributes.

The test used here defines N attributes named $data$, $data_2$, ..., $data_N$ (where N is the number of attributes) and updates these values 10000 times. The receiving end just counts the received updates to assert complete transmissions. The resulting network traffic is recorded using the network analysis software WireShark. Each test condition is repeated three times to compensate for small variations in the network traffic. For unknown reasons the transmission speed for CTK had to be limited artificially as many values were lost during the tests when no delays were used. A delay of 2-3ms between triggering two transmissions was found to be suitable to prevent data loss. This cause of the problem could not be determined.

In addition to these systematic tests also a more practical test was performed where pre-recorded data from a tri-axial accelerometer was transmitted. Here the three axes were grouped and 10242 samples were sent. Each logical sample here consisted of three floating point values with three decimal places (acceleration on each axis). This test was also repeated three times.

The times and transmission sizes are specific to the given problem and the used hardware setup and can therefore not be used to compare performance between this and another setup. These values are only used to compare the relative performance between the two frameworks under the same circumstances.

Test-Setup

To test both frameworks under comparable conditions two small applications were written that use the appropriate mechanisms from each framework to perform the defined test. All software is running on the same machine using the local loop network device. Each framework requires a central part (the CTK discoverer or the TCF server) that is restarted prior to each test to avoid caching effects. Recording network traffic starts after this component is running before the sending and receiving applications are started.

Traffic recoding stops when the receiving part signals a complete transmission and the sending application has terminated. The information on the needed time for transmission is computed by the receiving application. It simply uses the difference between the last and the first message containing relevant values and does not take other parts of the protocol explicitly into account (e.g., initial connection, keep-alive messages). The size of the recorded traffic however takes all exchanged information into account regardless of their meaning to the problem as this is reflects the real transfer time needed on bandwidth limited connections.

Results

After a few initial test-runs it was apparent that the size and time development behaves linearly for both systems. A few large value-counts were chosen at regular intervals and the behaviour for one to five values was additionally measured as a small number of values is a regular case in many applications. The results are shown in table 6.1.

The most striking result here is that the CTK framework has only a very little increase in transfer time over for a large number of values per update. In contrast the transfer time for TCF grows linearly with the number of values per update. The size of the transmission is proportional to the transmitted data

	time (s)		size (MiB)		time 3G (s)		BW MBit/s		size
N	TCF	CTK	T	C	T	C	T	C	C/T
30	373,3	18,8	57,6	53,0	1258,6	1157,6	1,29	23,66	0,92
25	325,6	16,5	48,6	46,3	1061,4	1012,4	1,25	23,59	0,95
20	248,7	15,8	38,5	39,8	841,1	870,0	1,30	21,18	1,03
15	188,0	14,9	29,0	33,2	634,2	725,1	1,30	18,67	1,14
10	129,5	13,5	19,6	26,8	427,8	584,6	1,27	16,67	1,37
5	63,6	13,1	9,9	20,2	216,3	440,4	1,31	12,91	2,04
4	51,3	13,0	8,0	19,1	174,7	416,6	1,31	12,31	2,38
3	39,1	13,2	6,1	17,6	133,5	383,7	1,31	11,17	2,88
2	26,8	12,8	4,2	16,3	91,9	356,7	1,32	10,71	3,88
1	14,4	12,8	2,3	15,0	50,8	327,8	1,36	9,87	6,46

Table 6.1: results (10000 updates)

and while the growth rate is lower for CTK it has a very high offset compared to TCF. For this particular experiment the data volume per update produced by TCF is lower until the set contains more than 20 values. After this point the protocol used by CTK becomes more efficient.

Estimating the time for data transmission at 384kbit/s (maximum standard UMTS rate) shows that both systems perform faster than this type of transmission would allow. The reduction in used data volume would enable TCF to perform faster than CTK up to the point where the volume of data becomes larger than its CTK equivalent.

Transmitting the pre-recorded acceleration samples draws a similar picture as seen in table 6.2.

	time (s)		size (MiB)		time 3G (s)		BW MBit/s		size
N	TCF	CTK	T	C	T	C	T	C	C/T
3	40,4	34,3	5,7	18,9	125,5	412,7	1,19	4,63	3,29

Table 6.2: results (10242 acceleration tuples)

In this more practical scenario the TCF data uses slightly less volume for the

three acceleration values than in the synthetic test (for $N = 3$). The size ratio between TCF and CTK is also even better here for TCF. This shows that representation of data is also of importance and different for both systems. As in the previous test the amount of saved data is very high and shows a potential advantage of the TCF protocol over CTK in similar cases. It is worth noting that the transmission times for both systems are far below the time that was used to record the values from the physical device. Both systems would therefore be able to transmit these values in real time.

A minor observation in the tests was that data in CTK does not arrive sequentially. From a quick technical analysis it seems that a context widget creates multiple connections for sending data and these are scheduled by the operating system resulting in a more or less random sequence. Since TCF only uses one connection for data transmission the sequence of data is always the same (FIFO principle).

6.1.3 Conclusion

In terms of transmission performance CTK is much better than TCF. This comes however at the cost of data volume and the need for direct communication between the sensor and the receiver. Also the data does not necessarily arrive in the same order as it was generated due to the use of multiple connections for transmission. This has to be taken into account when designing a system using information provided by CTK. In terms of data volume TCF produces far less data for a small number of values. Its protocol however becomes less efficient for higher number of values. While TCF sends a message for every single value CTK groups all values into a single message. Only the overhead introduced by the message format makes its use for small number of values less desirable. A conceptual difference between TCF and CTK is found when the state of an entity needs to be transmitted atomically. While CTK transmits all values at once and therefore satisfies atomic transfers in TCF the user has to rely on the time-stamps of each property. This allows atomic transfers if there is more than one millisecond time between two updates. Still the receiving end has to wait for all properties that belong to an entity before an atomic update can be finished.

When a suitable network for CTK exists and the volume of data is not of interest, as it is the case with smart indoor environments, CTK should be used for data transmission. For mobile networks the centralized approach of TCF might be more suitable as the client only needs to connect to a known server. For a small set of properties, e.g. a small set of sensors on a device, the reduced data amount allows a fast transmission even over connections with small bandwidth. For reading sensors the sequence of readings might be important. If time-stamping and sorting data on the receiver is not desired, the guaranteed sequence order of TCF can be a benefit.

A technical consideration is the nature of the connection in both frameworks. CTK uses HTTP connections between all entities and by this needs to open and close a network socket for each message. In TCF each client maintains a persistent connection to the context server to transfer all messages. For some mobile networks the HTTP approach of CTK may not be suitable if the receiver is not visible from the network or if the carrier simply does not forward incoming connections. In this case the centralised approach used in TCF will still work as the clients only need to reach the server that can in most cases be deployed in a way that makes it accessible to mobile clients. It is worth noting, that both frameworks deal with the creation of smart applications in their respective environments. There are also other approaches that deal with creating a smarter environment such as shown in [LMPMP+07] with totally different requirements. However, the TCF aims at providing a practical approach for experimentation and rapid prototyping that can be deployed in many kinds of networks. It does not provide special services for creating smart environments or adaptive sensor networks.

6.2 Creating High-Level Context

In the previous chapter the aquisition and transmission of low-level context information was presented. While this kind of information can reflect properties of the real world and trigger context related actions a higher level of abstraction can be useful.

6.2.1 Context Aggregation

Following the same approach as in CRN[BKLA06] high-level context is generated by aggregating low-level context information. In this aggregation step information is collected and interpreted. The resulting high-level context is then transmitted in the same way as low-level information. As the information is virtually indistinguishable from other context information other means of separation have to be established if needed. The TCF can provide several mechanisms to create such a division. For a complete separation a new context entity can be created where only high-level context is provided. This would also allow further abstractions of context where a hierarchical model of context information abstractions would emerge. This approach might however be too complex for simple applications. Another method of separation is the introduction of special tags to mark aggregated information. This would allow other entities to just observe one context while subscribing to the low- and high-level events that they are interested in. An even simpler approach that might be suitable for prototyping is the introduction of special sources that 'generate' high-level context. However, this approach requires a distinction based on names that are know a-priory (or dynamically negotiated). Depending on the expected complexity and lifetime of the application each of these approaches or combinations thereof may be suitable.

6.2.2 High-Level Example

To provide a better understanding of how the context distribution system can be used to form high-level context information an easy to follow but very simplified example can be used.

Assuming there exist two sensors in the environment that measure temperature and relative humidity a high-level context information can be aggregated that describes the quality of the working conditions. These sensors are in some way connected to a computer system that can transmit their information to a computer running the context server (note that this is only a logical separation and may take place on the same machine).

For this example the temperature sensor measures a room temperature of 22°C

while the humidity sensors measures 45% relative humidity. This can be aggregated to a high-level description of the working conditions (which are *optimal* in this case). A *virtual sensor* that acts as context client receives both readings and uses some adequate form of computation to provide a high-level abstraction of the working conditions that is usable by other clients. This high-level information is then transmitted to the context server to be distributed to all interested clients. See figure 6.5 for a graphical representation.

Regardless of the chosen way of distributing high-level information to the clients the technical aspects do not change.

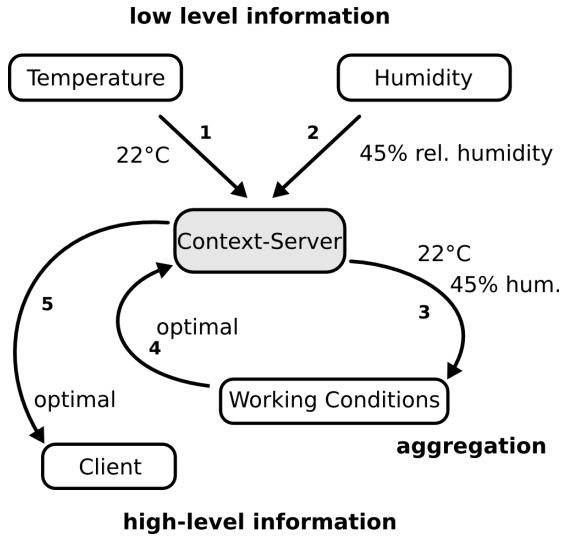

Figure 6.5: high-level context aggregation[1][2]

6.2.3 Limitations

While the creation of high-level abstractions is a straightforward process there are some limitations to be kept in mind. All communication between clients is relayed over the central context server. This design allows for a quick setup and discovery but has a drawback when building a long chain for context aggregation.

[1] numbers at arrows indicate sequence of transmission
[2] technically, step three would occur as two independent transmissions

In the given example for high-level context generation two sensors were inputting their readings into the server. Theses readings were then sent to a third component that computes an abstraction and sends it back to the server for distribution. If one imagines building more complex abstractions the need for reusing previous abstractions arises. For every abstraction level three sequential transmission blocks have to take place. First, the lower level information is sent to the server. In a second step this information is transmitted to all interested parties including the aggregator. Once the aggregator has computed the abstraction it is sent back to the server in a third step. While each transmission may be quick when seen alone this could become a problem if a timely response to a very high abstraction is needed. When comparing this to the approach used by the Context ToolKit it is obvious that sending the information directly from each client to other clients is an advantage there.

However, this requires that free communication between the clients is possible which may or may not be a trivial requirement for a given network. In many cell phone networks direct communication is not possible[1] and would need to be relayed over a central point. This would then technically amount to the same overhead that is always present in the protocol used by the context distribution system.

6.3 Framework usage in the SiWear Project

The TCF framework has been used in the SiWear project to transmit context information to clients. There were two stages of implementation with slightly different requirements to the distribution of context that were investigated. One of the goals of the SiWear project was improving the process of order picking with wearable computing technology. Optimizations in picking are of interest in many industrial areas as these operations account for 55%[BH09] to 65%[CBL02] of the total operational costs of a warehouse. Also, successfull wearable computing applications[SFH+98, Sta02] exist in this sector providing a reference for potential optimisations.

[1] internet connected cell phones are often not provided with an externally reachable address by the service provider

In a first step the potential benefit of a context aware wearable system for the picking process was evaluated in laboratory settings. The results from these experiments were used to refine the requirements for a system that could be used for a study in a live production environment. The context framework matured during the laboratory phase where an initial Wizard of OZ[Kel83] setting was gradually replaced by a computer controlled system. The performed studies are documented in [IBRK09].
In the next phase of the project the setup for the picking process, while still in a laboratory setting became more elaborate. While the first studies evaluated the use of a wearable computing application in general the second study compared different methods of conveying information about the picking tasks to the users of the system. Also ambient displays were used to mirror information on the tasks for the experimenters to be easily recorded on video. This scenerio made use of the distribution aspect of the TCF. This study is documented in [WBS+10].
While the previous studies simulated the use of environmental sensors and used the framework for general application control further studies incorporated real sensors to provide contextual information. To identify picking tasks a barcode reader was used to trigger loading relevant data into the system while laser range finder sensors where used to automatically detect picking of items. In preparation of the study documented in [BSI+11] both sensors were first simulated in software to define a suitable structure of the information within the framework. While processing information from a barcode reader is very straightforward the output of laser range finders is not directly useable. Adapters were developed to extract the position of hands as a high level information from the sensor data using heuristic methods. This approach is described in [Ibe12]. The simulated sensors could later be replaced by the real hardware without changes to the picking application as they simply used the same structure for providing contextual information.

6.3.1 Context for Picking

Throughout the performed studies processing information on performed picks is the primary driver of the wearable computing application. This information can either come from a Wizard of OZ setup or a sensor in the environment. In both cases the picking event needs to identify the location where the pick

occurred to enable the application to detect errors in the picking process. In industrial picking warehouses items are organized in hierarchical structures where normally an item has a defined location inside a shelf. In some cases, especially for large parts, a shelving unit can contain only a single location for that shelf but still keep the logical grouping. In any case item placements will follow a homogeneous pattern that allows assigning identifiers to their location.

First studies examined simple setups containing a small number of shelves with a few boxes for the items. For pick detection an event is needed that identifies the box. Apart from pick detection general control signals are needed that allow controlling the application, e.g. switching to the next task by an event from the barcode reader. The framework does not assign a meaning to the information it transmits and therefore relies on common knowledge among the components. To have a basic association with the task at hand contextual information for the picking process is transmitted in a context with the name *picking* (see section 5.4 for a reference on the internal model in the framework). While not used here, other applications or processes can listen to this specific context to monitor the flow of all information regarding picking.

All entities in the environment that provide information for picking are modelled as sources under this context. For each shelf a source with a matching identifier (to match the shelf in the used location definition) is created. Pick events are modelled as properties of the shelf with a name matching the identifier of the shelf. The value of this property indicates if a pick is currently taking place in the location, e.g. on the start of a pick, the value changes the *entered* and changed back to *left* when the worker retracts the hand again. Since these events are only valid for a short time and can change quickly they are modelled as transient properties in the context framework (as discussed in section 5.4). Since registering the interest in picking events for all shelfs and all boxes would be inefficient for an application, pick events make use of the tagging mechanism the framework provides. They are tagged with the *pick* tag and applications simply register interest in receiving events that contain this tag (inside the picking context).

6.3.2 Test-Scenario

Several studies in SiWear were carried out in a laboratory environment were the working conditions encountered at an industrial picking workplace were reconstructed. A laser range finder (LRF) from Hokuyo[2] was used to detect picks from a shelf or other picking location and a barcode reader from Metrologic/Honeywell[3] was used by participants to setup the task for picking. Both devices were monitored by clients for the server to transmit detected events to the wearable client.

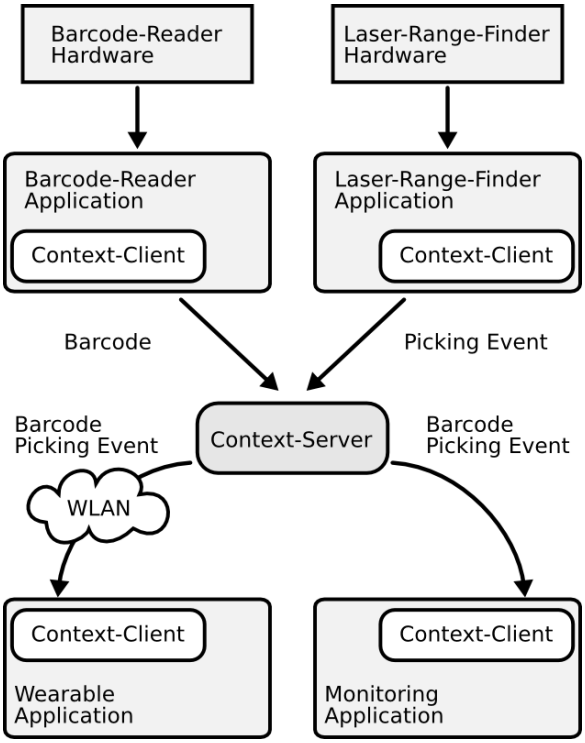

Figure 6.6: logical setup for tests

The client processing the LRF did not transmit the sensor data directly but used a mapping of known pick locations to transmit events for these areas. These events were the entering and leaving of the area. The Barcode-Reader infor-

[2] http://www.hokuyo-aut.jp
[3] http://www.honeywell.com

mation was transmitted as received by the sensor and only a small check was performed to filter wrong readings occurring on a few occasions.

On the wearable device a single context client connected to the central context server subscribing to events from the laser range finder and the barcode reader. The application was driven by these events and if needed by additional manual input. Figure 6.6 shows the logical setup of the system.

The greatest benefit of using the context framework in this scenario was the easy way of adding a second application to the setup that passively monitored all messages sent to the client. This allowed the experiment conductors to deploy a quick visual representation in into the system to keep track of what the user is seeing. This was not limited to presenting an exact copy of the users view but also provided additional data.

Context-Design

Context information for this scenario comes from processes triggered by the worker via direct interaction using the barcode scanner and implicit interaction when picks from the shelfs are detected by the LRF. To use this information a simple logical structure for context was designed.

The barcode reader and the laser range finder operate in the same logical context *'picking'*. The physical barcode reader is represented by a source named *'USB-Barcode'* (since the reader was connected via USB) and has a single property named *'barcode'* that gets updated with the value received from the reader. Additionally a tag *'Barcode'* is introduced to allow listeners to either register for events from the specific source or to all events using the tag. This can be useful to allow sending synthetic barcodes via software for corrections.

The *'picking'* context has no representation of the LRF but provides a source for each available shelf in the environment. The picking locations inside the shelves are represented by properties of the shelves using the location identifiers for names. Picks are modelled by setting these properties to 'enter' and 'leave' when the picker reaches into the location respectively when the pick is no longer detected by the LRF.

A typical tuple for context, source and property in this scenario has a structure similar to *('picking', 'B112', '23')* (where *'B112'* is the identifier of the shelf and *'23'* is the identifier of the location in the shelf').

6.3.3 Evaluation-Scenario

The framework was used in experiments on the picking process where workers were picking parts for a given task. The physical picking was monitored by a system of Laser Range Finders (LRF) from Leuze[4] that emitted detected picks to control a *pick-by-light* setup. These events were monitored by an application on a computer system to generate context events to be used on the wearable client. In addition to the pick information, the information on the current task was also extracted from a database and sent to the client as context information.

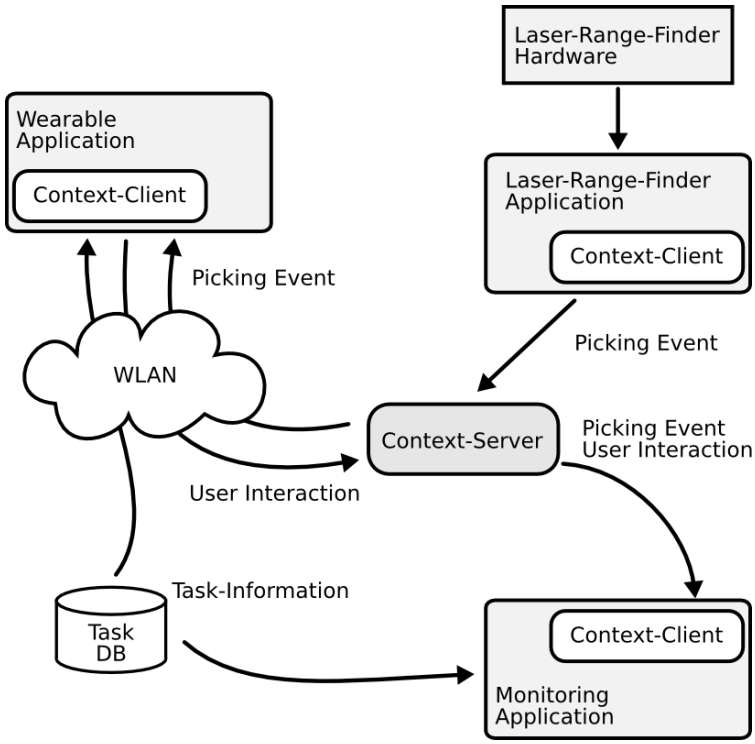

Figure 6.7: logical setup for evaluation

[4] http://www.leuze-electronic.de

Similar to the testing scenario the information from the laser range finder system was not sent directly. Because of different naming schemes in both applications a simple mapping layer was added to translate the received area identifiers to the named picking areas. The information on the items to pick were obtained from a snapshot of the production database. A query on the database resulted in sets of items to pick which were in turn analyzed and converted to an internal representation to be used by the picking application on the wearable client.
Again similar to the testing scenario a monitoring application was setup to provide the experiment conductors with real time information on the state of the wearable application. However the form of user interaction was different in this scenario as new tasks were not triggered by an external barcode reader but by user interaction on the wearable. Therefore all user interaction was replayed on the monitoring application by sending all relevant events as context information. Both applications were accessing the same database to get information on the current items to pick. Figure 6.7 shows the logical setup used in this scenario.

This scenario shows the flexibility of the framework. While the wearable client receives information from the LRF to aid in the picking process the user interface independently uses the context distribution mechanism to send information on its state. If this had been done with the CTK, the LRF had been modelled as a provider of information while the application would need to implement an additional context widget to distribute the state of the user interface. While this logical separation can be justified it implies using more network connections and makes service discovery more complex.

Chapter 7

Conclusion and Outlook

The creation of the TZI Context Framework (TCF) was motivated by the lack of a suitable framework for distribution of context information for wearable computing applications in the scope of industrial settings. The first chapter gave an introduction into the field and motivated the need for providing a framework for supporting wearable computing applications in general. In the second chapter functional aspects that a suitable framework would need to provide were defined and discussed. The third chapter of this work evaluated previous attemps of building wearable computing applications and collected the resulting requirements for handling information from the environment. Shared problems from the field of pervasive and ambient computing led to the evaluation of an existing framework for distributing context information for creating context aware applications. Chapter four evaluated the Context ToolKit (CTK) for its fitness in the field of wearable computing. Various technical and design aspects were found sub-optimal in the light of the identified requirements. This realization provoked the design of a specialized framework. In chapter five the design and technical implementation of the TCF is outlined. A technical comparison to CTK is shown in chapter 6 were synthetic tests show how the behaviour of the framework provides a benefit for wearable computing applications. The framework is then evaluated in the different stages of a wearable computing project showing its practical usabilty.

The TCF framework has shown to be an efficient part of a wearable computing

architecture that supports developers by providing a simple to use development approach that encourages experimentation. Its design also limits the use of resources on wearable clients and has only small requirements on bandwidth for data transmissions. While only an indirect conjecture, both factors have a positive effect on the overall energy consumption of the devices that are either part of the wearable application or the environment. This statement is based on the fact that a higher amount of transfered data and a higher amount of needed computing resources will require a greater amount of energy.

In an attempt to make the framework available to a larger community it has been published on the GitHub software collaboration platform and can be found at:
`https://github.com/wearlab-uni-bremen`
While there are no guarantees on how long this platform will be available it allows collaboration with other developers without any formal connection to a specific working group. The motivation behind this choice of platform is to encourage using the framework and potentially improving it based on upcoming needs.

While the framework has been developed with tasks from the industrial field in mind it is currently being used for prototyping applications in a different domain. In the Rehab@Home[1][PBLG13] project different environmental and on-body sensors are evaluated to create *serious games* for rehabilitation where exercises are motivated by playing games. The framework is used to create an abstraction layer between the actual games and the sensor devices. While these applications are not meant to be context aware, the features of the framework to transmit sensor values had a positive effect on development time. Similar to the SiWear project, it was again beneficial to be able to simulate sensors before creating a real implementation.

The framework can potentially be applied to many application domains that have the need to communicate events to distributed components. Its resource efficient design separates it from other available communication schemes and its simple data model allows developers to quickly start using it. In contrast

[1] `http://www.rehabathome-project.eu`

to web-service based approaches the stateful nature of the framework simplifies many aspects of communication. This comes however at the cost of introducing a new protocol. It is up to individual developers to carefully evaluate which approach is the best for their situation.

The TCF is not in all cases the ideal solution but its properties make it an interesting candidate for integrating sensor information into an application. While it was meant to help developers in a rapid-prototyping approach for wearable computing applications it has also potential to be used in other areas. In the spirit of *'the street finds its own uses for things'*[Gib86] other successful uses of the framework may emerge in the future.

References

[BGH+07] Christian Bürgy, Ulrich Glotzbach, Axel Hildebrand, Motoki Tonn, and Thomas Ziegert. Sichere Wearable Systeme zur Kommissionierung industrieller Güter sowie für Diagnose, Wartung und Reparatur. In Thilo Paul-Stueve, editor, *Mensch & Computer Workshopband*, pages 117–120. Verlag der Bauhaus-Universität Weimar, 2007.

[BH09] J. Bartholdi and S. Hackmann. Warehouse and distribution science release 0.89. Technical report, Georgia Institute of Technology, January 2009.

[BKLA06] David Bannach, Kai S. Kunze, Paul Lukowicz, and Oliver Amft. Distributed modular toolbox for multi-modal context recognition. In *ARCS*, pages 99–113, 2006.

[BSI+11] Hannes Baumann, Thad Starner, Hendrik Iben, Anna Lewandowski, and Patrick Zschaler. Evaluation of graphical user-interfaces for order picking using head-mounted displays. In *Proceedings of the 13th International Conference on Multimodal Interfaces*, ICMI '11, pages 377–384, New York, NY, USA, 2011. ACM.

[CBL02] J. Coyle, E. Bardi, and C. Langley. *The Management of Business Logistics: A Supply Chain Perspective*. South-Western College, Cincinnati, OH, 2002.

[DBu14] The D-Bus message bus system. `http://dbus.freedesktop.org`, 2014. accessed 07.02.2014.

[Dey00] Anind Kumar Dey. *Providing architectural support for building context-aware applications*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2000. AAI9994400.

[DSGP03] Richard W. DeVaul, Michael Sung, Jonathan Gips, and Alex Pentland. MIThril 2003: Applications and architecture. In *ISWC*, pages 4–11. IEEE Computer Society, 2003.

[Ecl14] Eclipse IDE. `http://www.eclipse.org`, 2014. accessed 27.03.2014.

[ECM14] ECMAScript - the language of the web. `http://www.ecmascript.org/`, 2014. accessed 09.04.2014.

[Eud98] The Eudaemons' shoe. `http://wearcam.org/historical/node3.html`, 1998. accessed 20.11.2013.

[FSK05] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gib86] William Gibson. Burning Chrome. In *Burning Chrome*. Arbor House, New York, NY, USA, 1986.

[GJ90] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, Upper Saddle River, NJ, USA, 1990.

[HTT99] HTTP protocol. `http://tools.ietf.org/html/rfc2616`, 1999. accessed 27.03.2014.

[Ibe12] Hendrik Iben. A heuristic approach to robust laser range finder based pick detection. In *Proceedings of the 16th International*

Symposium on Wearable Computers, ISWC '12, pages 108 –111, 2012.

[IBRK09] Hendrik Iben, Hannes Baumann, Carmen Ruthenbeck, and Tobias Klug. Visual based picking supported by context awareness: Comparing picking performance using paper-based lists versus lists presented on a head mounted display with contextual support. In *Proceedings of the 2009 International Conference on Multimodal Interfaces*, ICMI-MLMI '09, pages 281–288, New York, NY, USA, 2009. ACM.

[IRC93] Internet Relay Chat (IRC) protocol. `http://tools.ietf.org/html/rfc1459`, 1993. accessed 07.02.2014.

[IRC94] Client-To-Client-Protocol (CTCP). `http://www.irchelp.org/irchelp/rfc/ctcpspec.html`, 1994. accessed 14.03.2014.

[KA82] Ralph Katz and Thomas J. Allen. Investigating the not invented here (NIH) syndrome: A look at the performance, tenure, and communication patterns of 50 R&D project groups. *R&D Management*, 12(1):7–20, 1982.

[Kel83] J. F. Kelley. An empirical methodology for writing user-friendly natural language computer applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '83, pages 193–196, New York, NY, USA, 1983. ACM.

[KWN05] Holger Kenn, Hendrik Witt, and Tom Nicolai. Towards a formal description of context. In *2nd International Forum on Applied Wearable Computing (IFAWC)*, 2005.

[LHBK11] Michael Lawo, Otthein Herzog, Michael Boronowsky, and Peter Knackfuss. The Open Wearable Computing Group. *IEEE Pervasive Computing*, 10(2):78–81, 2011.

[LHW07] Michael Lawo, Otthein Herzog, and Hendrik Witt. An industrial case study on wearable computing applications. In *Proceedings of the 9th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '07, pages 448–451, New York, NY, USA, 2007. ACM.

[LMPMP+07] Clemens Lombriser, Mihai Marin-Perianu, Raluca Marin-Perianu, Daniel Roggen, Paul Havinga, and Gerhard Tröster. Organizing context information processing in dynamic wireless sensor networks. In *3rd International Conference on Intelligent Sensors, Sensor Networks, and Information Processing (ISSNIP)*, pages 67–72, 0 2007.

[Lov05] Robert Love. Get on the D-Bus. *Linux Journal*, (130), 2005. `http://www.linuxjournal.com/article/7744`.

[LTGLH07] Paul Lukowicz, Andreas Timm-Giel, Michael Lawo, and Otthein Herzog. WearIT@work: Toward real-world industrial wearable computing. *IEEE Pervasive Computing*, 6(4):8–13, October 2007.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[McC76] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[Met14] Metrics plugin for the Eclipse IDE. `http://metrics2.sourceforge.net`, 2014. accessed 27.03.2014.

[MHP00] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present and future of user interface software tools. *ACM TRANSACTIONS ON COMPUTER-HUMAN INTERACTION*, 7:3–28, 2000.

[PBLG13] Lucia Pannese, Giancarlo Bo, Michael Lawo, and Silvia Gabrielli. The Rehab@Home project: Engaging game-based home rehabilitation for improved quality of life. In *Proceedings of the SEGAMED Conference*, 2013.

[Rhi14] Mozilla Rhino. `www.mozilla.org/rhino`, 2014. accessed 09.04.2014.

[RLRT11] Daniel Roggen, Clemens Lombriser, Mirco Rossi, and Gerhard Tröster. Titan: An enabling framework for activity-aware "pervasive apps" in opportunistic personal area networks. *EURASIP J. Wirel. Commun. Netw.*, 2011:1:1–1:22, January 2011.

[Sch00] Albrecht Schmidt. Implicit human computer interaction through context. *Personal Technologies*, 4(2-3):191–199, June 2000.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The Context Toolkit: Aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 434–441, New York, NY, USA, 1999. ACM.

[SFH+98] R. Stein, S. Ferrero, M. Hetfield, A. Quinn, and M. Krichever. Development of a commercially successful wearable data collection system. In *IEEE Intl. Symp. on Wearable Computers*. IEEE Computer Society, 1998.

[Spr13] Spring framework. `http://spring.io`, 2013. accessed 16.12.2013.

[SS02] Asim Smailagic and Daniel Siewiorek. Application design for wearable and context-aware computers. *IEEE Pervasive Computing*, 1(4):20–29, October 2002.

[ST94] Bill Schilit and Marvin Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8:22–32, 1994.

[Sta02] Thad E. Starner. Wearable computers: No longer science fiction. *IEEE Pervasive Computing*, 1(1):86–88, 2002.

[Sti08] Thomas Stiefmeier. *Real-Time Spotting of Human Activities in Industrial Environments*. PhD thesis, ETH, Zürich, Germany, 2008.

[Tho98] Edward O. Thorp. The invention of the first wearable computer. In *Proceedings of the 2Nd IEEE International Symposium on Wearable Computers*, ISWC '98, pages 4–, Washington, DC, USA, 1998. IEEE Computer Society.

[URL05] Uniform Resource Identifier (URI): Generic syntax. `http://tools.ietf.org/html/rfc3986`, 2005. accessed 08.04.2014.

[UTF03] UTF-8, a transformation format of ISO 10646. `http://tools.ietf.org/html/rfc3629`, 2003. accessed 07.04.2014.

[WBS+10] Kimberly A. Weaver, Hannes Baumann, Thad Starner, Hendrick Iben, and Michael Lawo. An empirical task analysis of warehouse order picking using head-mounted displays. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1695–1704, New York, NY, USA, 2010. ACM.

[Web04] Web Services architecture. `http://www.w3.org/TR/ws-arch/`, 2004. accessed 27.03.2014.

[Web11] WebSocket protocol. `http://tools.ietf.org/html/rfc6455`, 2011. accessed 27.03.2014.

[Wer96] C. Werry. Linguistic and Interactional Features of Internet Relay Chat. In S. Herring, editor, *Computer-Mediated Communication: Linguistic, Social and Cross-Cultural Perspectives*, pages 47–63. John Benjamins, Amsterdam, 1996.

[Wit08] Hendrik Witt. *User interfaces for wearable computers: development and evaluation.* PhD thesis, University of Bremen, 2008. http://d-nb.info/987814362.

[WNK07] Hendrik Witt, Tom Nicolai, and Holger Kenn. The WUI-toolkit: A model-driven UI development framework for wearable user interfaces. In *ICDCS Workshops*, page 43. IEEE Computer Society, 2007.

List of Figures

4.1 component model in CTK (from [Dey00]) 25
4.2 CTK architecture (from [Dey00]) 26
4.3 dependency graph of the Context ToolKit 30
4.4 dependency graph of the Context ToolKit core 32

5.1 room context example 45
5.2 context hierarchy 45
5.3 property details 46
5.4 client server diagram 47
5.5 context filtering for a second server 48
5.6 simplified environment model 49
5.7 context data model 51
5.8 simplified client model 52
5.9 dependency graph of common package 55
5.10 dependency graph of server package 56
5.11 dependency graph of client package 57
5.12 dependency graph of client package, context view 58
5.13 message structure 59
5.14 context data structure (transmission) 60
5.15 context message 61
5.16 main server loop 63
5.17 subscribe command 64
5.18 subscribe command - context definition 64
5.19 subscribe command - source definition 64
5.20 subscribe command - property definition 65
5.21 subscribe command - example 65

5.22 subscribe command - numeric id example 65
5.23 subscribe command - wildcard example 66
5.24 short context data structure 67
5.25 short context message 67
5.26 simplified history class relations 70
5.27 history database tables 72
5.28 history database BLOB access 73

6.1 architecture of the temperature demo 78
6.2 temperature application UI 79
6.3 package dependencies CTK (temperature) 82
6.4 package dependencies TCF (temperature) 83
6.5 high-level context aggregation 90
6.6 logical setup for tests 94
6.7 logical setup for evaluation 96

A.1 example LISTSRC reply 115
A.2 example LISTPRP reply 116

B.1 Environment Model 122
B.2 Common Data Structures 123
B.3 Server Data Structures 124
B.4 Client Data Structures 125

List of Abbreviations

BLOB	Binary Large Object
CPU	Central Processing Unit
CRN	Context Recognition Network (Toolbox)
CTK	Context ToolKit
DI	Dependency Injection
FIFO	First In, First Out
GPRS	General Packet Radio Service
GPS	Global Positioning System
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol
IDE	Integrated Development Environment
IPC	Inter Process Communication
JDBC	Java Database Connectivity
LAN	Local Area Network
LRF	Laser Range Finder
MIT	Massachusetts Institute of Technology
OOP	Object Oriented Programming
SQL	Structured Query Language

TCF TZI Context Framework

UI User Interface

UML Unified Modelling Language

URL Universal Resource Locator

UTF-8 8-bit UCS Transformation Format

WLAN Wireless Local Area Network

WUI-TK Wearable UI Toolkit

XML eXtensible Markup Language

Appendix A

Server Commands

A.1 Connection Maintenance Commands

LOGIN *<Name>*
Connect to the server with a given name
Reply: **ACCEPT** *<Name>* *<Id>* - *Name* is your name and *Id* is a positive integer that identifies this connection (unique)

RELOGIN *<Name>* *<Id>*
Reclaim connection after connection loss using old *Id*
Reply: **ACCEPT** *<Name>* *<Id>* - same as for **LOGIN**
Reply: **FAIL** - if the *Id* is not known (anymore)

When a client successfully reconnects to the server, all previous subscriptions still apply. If any context events arrived while the client was not connected, they will be transmitted now. However, some events might still be lost depending on the technical circumstances causing the earlier involuntary disconnect.
If the client set up *short context mode* the server will continue using this mode.

LOGOUT
Terminate connection to the server; invalidates your *Id*
Reply: **OK**

After logging out, the server will free any resources associated with the particular client. This includes all subscriptions and settings.

PING[1]
Request a *PONG*-reply from the server.
Reply: **PONG**

A client can use this command as a means of checking the connection. The server will reply as fast as possible.
While no other communication takes place, the server will also periodically send a *PING* message to check the connection. The client is expected to either reply with a *PONG* message or any other valid message.
If the client does not reply to server generated *PING* messages the server will assume a communication problem and disconnect. For a defined time, the client will be able to recover the connection using the **RELOGIN** command.

PONG[1]
Answer a *PING*-message from the server.
Reply: Nothing

While this command has no further meaning it can be used by a client to proactively mark itself as being active. A very simple client could choose to ignore *PING* messages from the server and instead periodically send *PONG* messages while no other commands need to be executed.

A.2 General Queries

STARTTIME
Request startup time from server
Reply: **REPLY** <*time*> - time of server start (epoch, in ms)

A client can use the start-time value to determine if a server was restarted since the last communication.

[1] can be send at any time when logged in

2 123 3 345=source1 456=source2 567*=source3 234 1 678=source4
Sources for two contexts (123 and 234). Context one has three sources (345, 456 and the unset source 567). The second context has one source (678).

Figure A.1: example LISTSRC reply

LISTIDS
List all known numeric ids
Reply: **REPLY** *<#Ids> <Id_1> ... <Id_n>*
number of ids followed by all ids

LISTCLT
List all connected clients
Reply: **REPLY** *<#Clients> <$ClientId_1$>=<$ClientName_1$>, ...*
<$ClientId_n$>=<$ClientName_n$>
number of clients followed by all pairs of client ids and associated names.

LISTCTX
List all defined contexts
Reply: **REPLY** *<#Contexts> <$ContextId_1$>=<$ContextName_1$>, ...*
<$ContextId_n$>=<$ContextName_n$>
number of contexts followed by all pairs of context ids and associated names.

LISTSRC *[<Context>*]*
List all sources for given contexts (or all contexts, if none given)
Reply: **REPLY** *<#Contexts> <$ContextId_1$> <#Sources>*
<$SourceId_1$>=<$SourceName_1$> ... <$SourceId_n$>=<$SourceName_n$> ... <$ContextId_m$>
...
number of contexts in answer and that many times a *ContextId* followed by the number of sources in that context. This is followed by pairs of source ids and associated names. Source ids may have an additional postfix '*' that signals that this source does not yet contain anything. See figure A.1 for an example on the output format.

LISTPRP *<Context> <#Sources> <$Source_1$> ... <$Source_n$>* *
List all properties for given contexts and sources (or all sources on a context if 0

2 123 2 345 2 912=prop1 192*=prop2 456 1 139=prop3
234 1 678 1 492=prop4

Properties for two contexts (123 and 234). Two sources at context one; source 345 has 2 properties (912 and the unset property 192). Source 456 has one property (139). There ist one source for the second context (678) with one property(492).

Figure A.2: example LISTPRP reply

is given for number of sources). You can specify multiple contexts and sources to list.
Reply: **REPLY** *<#Contexts> <ContextId$_1$> <#Sources> <SourceId$_1$> #Properties <PropertyId$_1$>=<PropertyName$_1$> ...*
<PropertyId$_n$>=<PropertyName$_n$> ... <ContextId$_m$> ...
number of contexts in answer and that many times a *ContextId* followed by the number of sources listed for that context. Each sources id is given, followed by the number of properties for that source. This number indicates the number of property id and name pairs which follow. Additionally, each property id can have a postfix '*' indicating that this property has not been set yet. See figure A.2 for an example on the output format.

GETCTXID *<ContextName>*
Request the numerical id for a given context
Reply: **REPLY** *<id>* - returns the id of the context or **FAIL** if no such context is known.

GETSRCID *<Context> <SourceName>*
Request the numerical id for a given source
Reply: **REPLY** *<id>* - returns the id of the source or **FAIL** if no such source (or context) is known.

GETPRPID (*<Context> <Source>* | *<SourceId>*) *<PropertyName>*
Request the numerical id for a given property
Reply: **REPLY** *<id>* - returns the id of the property or **FAIL** if no such property (or context/source) is known.

GETIDINFO *<Id>*
Request information on id
Reply: **REPLY IDINFO** *(C | S | P | U) <Id>* - returns *C* for contexts, *S* for sources and *P* for properties. Everything else (invalid ids, client ids, subscription ids, etc.) is returned as *U* (unknown).

A.3 Context Manupilation Commands

CREATECTX *<ContextName>*
Create a context
Reply: **REPLY** <Id> - Returns the id of the created context; if a context by the same name already exists, the existing id is returned

CREATESRC *<Context> <SourcetName>*
Create a source
Reply: **REPLY** <Id> - Returns the id of the created source; if a source of the same name exist in this context, the existing id is returned
FAIL is returned, if the context does not exist

CREATEPRP (*<Context> <Source>* | *<SourceId>*) *<PropertytName>*
Create a property
Reply: **REPLY** <Id> - Returns the id of the created property; if a source of the same name exist in this source, the existing id is returned
FAIL is returned, if either the context or the source do not exist

A.4 Subscription Management

SUBSCRIBE *#Def* (*<Context> #SrcDef* (*<Source> #PrpDef* (*<Property> #Tags Tag**)))
SUBSCRIBE *#Def* (S (*<SourceId> #PrpDef* (*<Property> #Tags Tag**)))
SUBSCRIBE *#Def* (P (*<PropertyId> #Tags Tag**))
Subscribe to context events.
Context, Source and Property items can be reference either by their name or

their numeric identifier. A name is prefixed with '@', otherwise the numeric identifier is required. Numeric ids for sources and properties imply information on their parent structures. Therefore two shorthand versions of the subscribe command exists, that allow skipping parts of the definitions. A 'P' in place of the context identifier means that a property will be referenced by identifier. The letter 'S' indicates that a numeric source identifier will be used. The arguments that are marked with a '#' in the command definitions define the number of times the following structure is repeated. In case of the short-cut versions, the previous number serves this purpose, e.g. a '#Def' of 2 with a 'P' short-cut selects two property definitions.
Reply: **REPLY** <Id> - Returns the id of the created subscription
See section 5.4.5 for more details on subscribing.

SUBSCRIPT <*Context*> <*Script*> - Send a script to handle events
Reply: **REPLY** <Id> - Returns the id of the created subscription
See section 5.4.10 for more details on the scripting mechanism.

LISTSUB
List current subscriptions
Reply: **REPLY** *#Subscriptions* $SubscriptionId_1$ ... $SubscriptionId_n$

SHORTSUB (**true**|**false**) (**all** | <$SubscriptionId_1$> ... <$SubscriptionId_n$>)
Enable or disable short mode for subscriptions. You can either specify *all* to affect all subscription or list subscriptions individually.

Reply: **OK**; on invalid subscription id, **FAIL** is returned and only some of the given subscriptions may have been processed.

CANCELSUB (**all** | <$SubscriptionId_1$> ... <$SubscriptionId_n$>)
Cancel subscriptions. You can either specify *all* to affect all subscription or list subscriptions individually.
Reply: **OK**; on invalid subscription id, **FAIL** is returned and only some of the given subscriptions may have been processed.

A.5 Context Setting

SETPRP (<*Context*> <*Source*> <*Property*> | <*SourceId*> <*Property*> | <*PropertyId*>) = <*Value*> <*TimeStamp*> #*Tags* Tag_1 ... Tag_n [**P**]
Set a property value
A time-stamp can be specified in ms since epoch or can be set to -1. In case of -1 the server will set to timestamp to the time of arrival.
The number of tags may be 0 or must be followed by that number of tags. Specifying an optional *P* at the end marks the property as being *persistent* (default is non-persistent).
Reply: **OK**
FAIL is returned when the specified property does not exist.

This command represents the central function of the server from a information source point of view. A sensor representation uses this command to update the corresponding information on the server. The server will in turn check, if any client is interested in receiving a notification about the change and send a *CTX* or *SCTX* message if needed.

A.6 Context History

HISTORY (<*Context*> <*Source*> <*Property*> | <*SourceId*> <*Property*> | <*PropertyId*>) ((**GET** [**-**]<From> [+|**r**]<To> <Limit>) | latest | earliest) -
Retrieve history values for given property
Reply: #*Elements* $ContextElement_1$ ... $ContextElement_n$
The returned context elements are returned in the same way as for normal context events but without the identifying context or subscription id.

All query commands can fail if an unknown context, source or property is specified.

A.7 Large Transfers

Transfer commands have a *transfer identifier* as their first argument. This id is chosen by the side that initiates a transfer.

TX | CTXTX <*TID*> (0 <SIZE> | <*PNUM*> <DATA>)
Start a transfer of send data. A transfer starts with a packet numer of 0 and the total size of the data to be transfered. Following messages have a packet number greater 0 and contain data. The difference between TX and CTXTX is that the latter is used to indicate the transfer of a context event.

TXACK <*TID*> <*PNUM*>
Packet acknowledgement. When this message is received the next packet should be sent.

TXRESEND <*TID*> <*PNUM*>
Packet resend request. When this message is received the packet with the given number should be resend.

TXCANCEL <*TID*>
Transfer cancel. When this message is received the corresponding transfer is cancelled. This command is only valid when sent from the side that initialized the transfer.

Appendix B

UML Diagrams

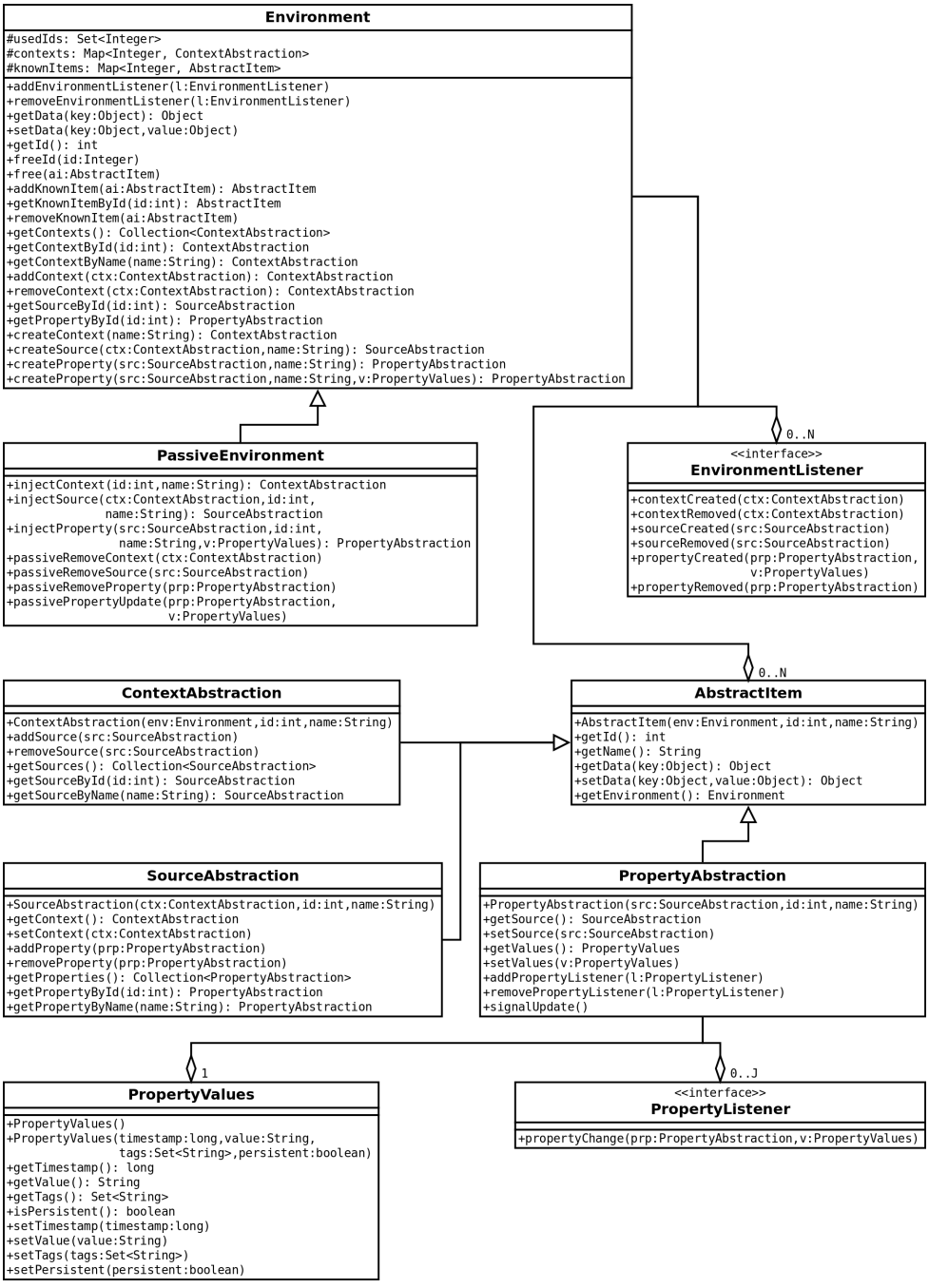

Figure B.1: Environment Model

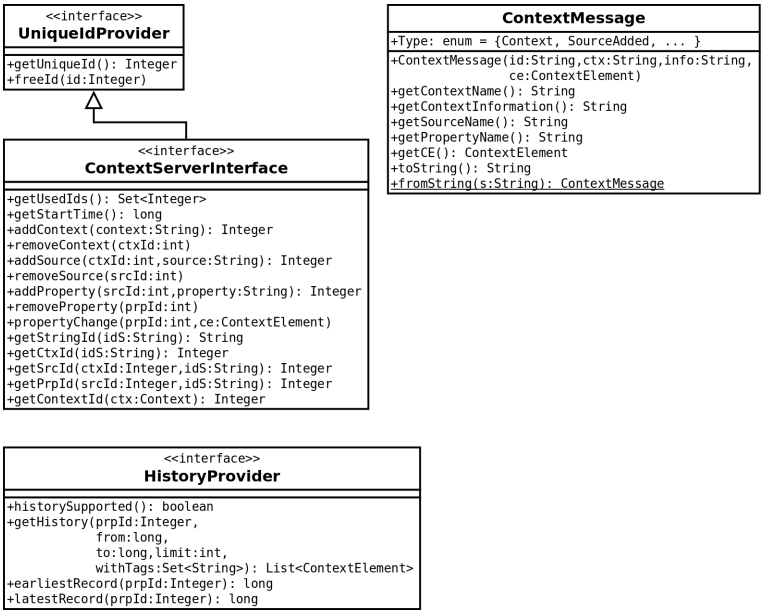

Figure B.2: Common Data Structures

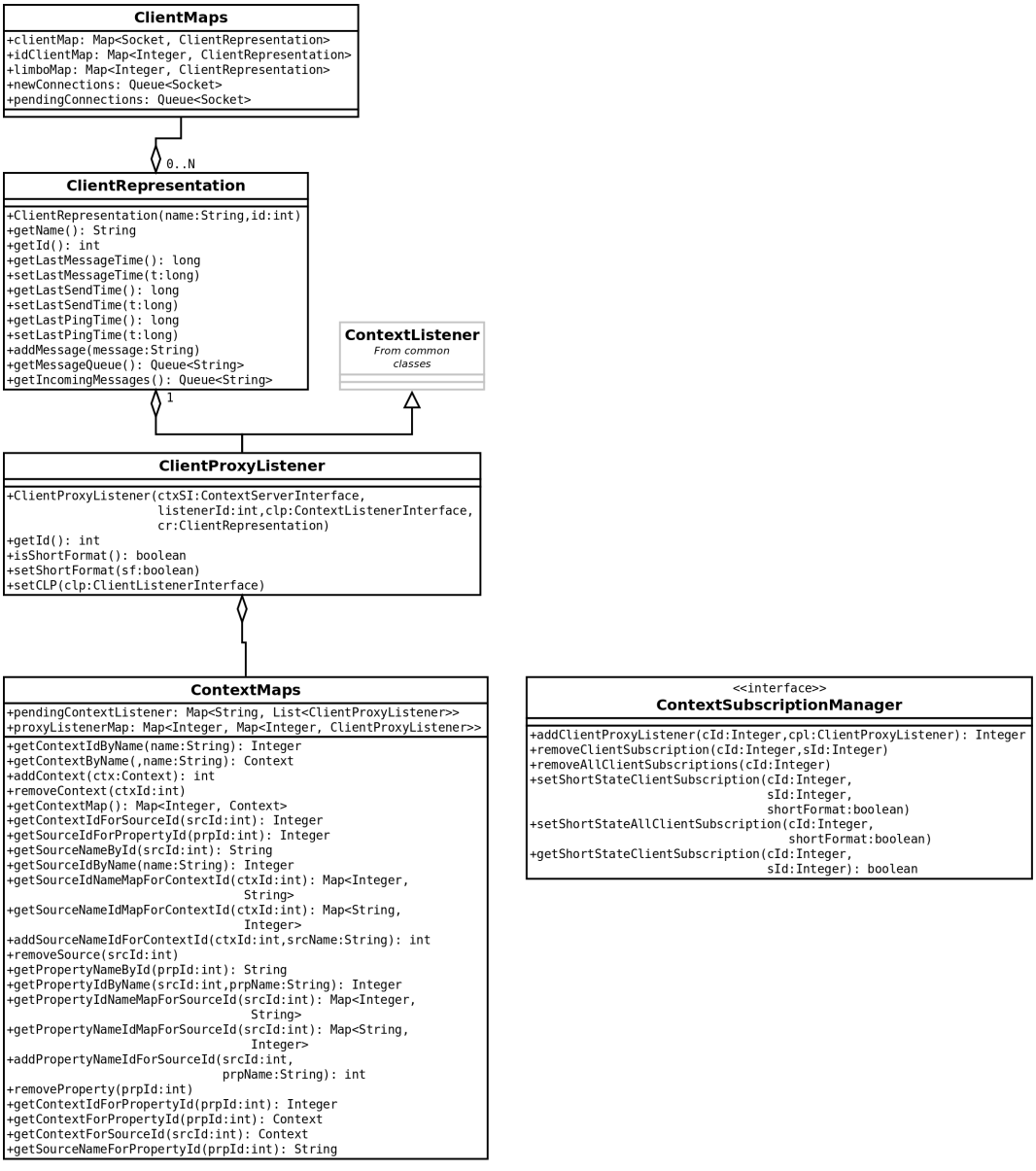

Figure B.3: Server Data Structures

Figure B.4: Client Data Structures

Appendix C

Code Examples

The following code examples show small fragments of key functionality and are not by themselves complete programs.

C.1 Context Subscription and Processing

An application can make use of the framework by simply subscribing to known sources of information and process incoming events by registering an instance of a listener interface.

```
//[...] initialisation code
//      hostname and port refer to server
//      name is an arbitrary identifier
  ContextClient cc = new ContextClient();
  cc.init(hostname, port);
  cc.login(name);
  cc.setAutoReconnect(true);
  ContextManager cm = new ContextManager(cc);
  cm.addContextListener(this);

  cm.subscribe("sensors", "temperature", "office");

//[...] implementation of Listener interface
@Override
```

```
public void processContext(Context ctx, ContextElement ce) {
  // data processing
}
```

C.2 Context Generation

The general pattern for context generation is to perform an initial setup and then updating information as it comes in. A software interface for a temperature sensor would read the sensor at set intervals and update the context information after each result.

```
// [...] instance field
ContextManager cm;

//[...] initialisation code
//       hostname and port refer to server
//       name is an arbitrary identifier
  ContextClient cc = new ContextClient();
  cc.init(hostname, port);
  cc.login(name);
  cc.setAutoReconnect(true);
  ContextManager cm = new ContextManager(cc);

//[...] application specific update event, e.g. sensor reading
  cm.setProperty("sensors", "temperature", "office",
    Integer.toString(reading));
```

When generating context the actual transmission of values is transparent to the application and performed in a background task. It is therefore possible to perform context updates inside methods that need to finish quickly such as GUI event listeners.

C.3 Temperature Demo CTK

This code was used in the application complexity tests in section 6.1.1 for the CTK values.

C.3.1 CTKTempReader.java

```
package ctktest;

import java.awt.EventQueue;
import java.util.LinkedList;
import java.util.List;
import java.util.Vector;

import context.arch.BaseObject;
import context.arch.InvalidMethodException;
import context.arch.MethodException;
import context.arch.comm.DataObject;
import context.arch.comm.clients.IndependentCommunication
    ;
import context.arch.discoverer.ComponentDescription;
import context.arch.discoverer.Discoverer;
import context.arch.discoverer.componentDescription.
    NonConstantAttributeElement;
import context.arch.discoverer.querySystem.
    AbstractQueryItem;
import context.arch.discoverer.querySystem.QueryItem;
import context.arch.handler.Handler;
import context.arch.storage.Attribute;
import context.arch.subscriber.ClientSideSubscriber;
import context.arch.subscriber.DiscovererSubscriber;
import context.arch.util.Error;
import context.arch.widget.Widget;

public class CTKTempReader extends BaseObject implements
    Handler {
  private String widgetId = null;
  private String discoId = null;

  public interface TemperatureListener {
```

```
    public void onTemperature(float temp);
  }

  private List<TemperatureListener> listener = new
      LinkedList<TemperatureListener>();

  private void notifyListener(float temp) {
    List<TemperatureListener> tmplist = null;
    synchronized (listener) {
      tmplist = new LinkedList<TemperatureListener>(
          listener);
    }
    for(TemperatureListener l : tmplist) {
      l.onTemperature(temp);
    }
  }

  public void addListener(TemperatureListener l) {
    synchronized (listener) {
      listener.add(l);
    }
  }
  public void removeListener(TemperatureListener l) {
    synchronized (listener) {
      listener.remove(l);
    }
  }

  public CTKTempReader(int localServerPort) {
    super(localServerPort);
    setId(Widget.getId("TempReader", localServerPort));
    findDiscoverer();
    discovererRegistration();

    boolean success = false;
```

```
AbstractQueryItem aqi = new QueryItem(new
    NonConstantAttributeElement("temperature", null,
    Attribute.FLOAT));

Vector<?> res = discovererQuery(aqi);

if(res != null && res.size()>0) {
  ComponentDescription cd = (ComponentDescription)res
      .firstElement();
  ClientSideSubscriber cs = new ClientSideSubscriber(
      getId(), getHostName(), getPort(), "update",
      null, null);
  Error er = subscribeTo(this, cd.id, cd.hostname, cd
      .port, cs);


  if(er.getError().equals(Error.NO_ERROR)) {
    widgetId = cs.getSubscriptionId();
    System.out.format("WidgetID: %s\n", widgetId);

    success = true;
    System.out.format("Successfully subscribed...\n")
        ;
  } else {
    System.err.format("Error while subscribing... %s\
        n", er.toString());
  }
} else {
  System.err.format("Did not discover the temperature
      widget...\n");
}

if(!success) {
```

```
      System.out.println("Subscribing to discoverer...\n"
          );

      DiscovererSubscriber ds = new DiscovererSubscriber(
          getId(), getHostName(), getPort(), Discoverer.
          NEW_COMPONENT, aqi);
      discovererSubscribe(this, ds);

      discoId = ds.getSubscriptionId();
    }
  }

  private Object getValue(DataObject data, String name) {
    DataObject nonconstant = data.getDataObject("NCANVS")
        ;
    if(nonconstant==null) {
      System.err.println("No non-constant attributes...")
          ;
      return null;
    }

    DataObject attributes = nonconstant.getDataObject("
        attributes");

    if(attributes==null) {
      System.err.println("No attributes...");
      return null;
    }

    for(Object o : attributes.getChildren()) {
      if(o instanceof DataObject) {
        DataObject dochild = (DataObject)o;
        if(!dochild.getName().equals("attributeNameValue"
            ))
          continue;
```

```
        Vector<?> namedAttrs = dochild.getValue();
        Object value = null;
        boolean hasName = false;
        boolean hasValue = false;
        for(Object no : namedAttrs) {
          if(no instanceof DataObject) {
            DataObject ndo = (DataObject)no;
            Vector<?> values = ndo.getValue();

            if(values == null || values.size() < 1)
              continue;

            if(ndo.getName().equals("attributeName") &&
                name.equals(values.get(0))) {
              hasName = true;
            }
            if(ndo.getName().equals("attributeValue")) {
              hasValue = true;
              value = values.get(0);
            }
          }
          if(hasName && hasValue)
            return value;
        }
      }
    }
    System.err.println("No matching data...\n");
    return null;
  }

  @Override
  public DataObject handle(String subscriptionId,
      DataObject data)
  throws InvalidMethodException, MethodException {
```

```
System.out.format("Message from %s\n", subscriptionId
    );
System.out.println(data);
System.out.println("HT: " + data.getAttributes());
if(subscriptionId.equals(widgetId)) {
  Object dotemp = getValue(data, "temperature");
  if(dotemp instanceof String) {
    System.out.println("Temp: " + dotemp);
    try {
      notifyListener(Float.parseFloat((String)dotemp)
          );
    } catch(NumberFormatException nfe) {
      System.err.println("Invalid temperature: " +
          dotemp);
    }
  } else {
    System.err.println("No Temp! " + dotemp + (dotemp
        == null ? "<null>" : dotemp.getClass().
       getName()));
  }
    return (new Error(Error.NO_ERROR)).toDataObject
        ();
}
if(subscriptionId.equalsIgnoreCase(discoId)) {
  DataObject w = data.getDataObject (Discoverer.
     DISCOVERER_QUERY_REPLY_CONTENT);
      ComponentDescription widget =
          ComponentDescription.
          dataObjectToComponentDescription (w);
      ClientSideSubscriber cs = new
          ClientSideSubscriber(getId(), getHostName(),
           getPort(), "update", null, null);

      subscribeTo(this, widget.id, widget.hostname,
          widget.port, cs);
```

```
            widgetId = cs.getSubscriptionId();
            System.out.format("WidgetID: %s\n", widgetId);
            return (new Error(Error.NO_ERROR)).toDataObject
                ();
        }

        return null;
    }

    @Override
    public void handleIndependentReply(
        IndependentCommunication independentCommunication)
            {
    }

    public static void main(String... args) {
        final CTKTempReader tr = new CTKTempReader(1235);
        EventQueue.invokeLater(new Runnable() {

            @Override
            public void run() {
              new TempReadUI(tr);
            }
        });
    }
}
```

C.3.2 TempReadUI.java

```
package ctktest;

import java.awt.BorderLayout;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
```

```
import javax.swing.BoxLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class TempReadUI implements CTKTempReader.
    TemperatureListener {
  private JLabel tempLabel;


  private JFrame frame;

  public TempReadUI(CTKTempReader tr) {
    tr.addListener(this);

    frame = new JFrame("ReadUI");
    frame.setDefaultCloseOperation(JFrame.
        DISPOSE_ON_CLOSE);
    frame.setLocationByPlatform(true);

    JPanel tempPanel = new JPanel();
    tempPanel.setLayout(new BoxLayout(tempPanel,
        BoxLayout.PAGE_AXIS));


    JPanel p;
    p = new JPanel();
    p.add(new JLabel("Temperature"));
    tempPanel.add(p);

    p = new JPanel();
    p.add(tempLabel = new JLabel("?␣°C"));
    tempPanel.add(p);
```

```
    frame.add(tempPanel, BorderLayout.NORTH);

    frame.pack();
    frame.setVisible(true);

    frame.addWindowListener(new WindowListener() {

      @Override
      public void windowOpened(WindowEvent e) {  }

      @Override
      public void windowIconified(WindowEvent e) {  }

      @Override
      public void windowDeiconified(WindowEvent e) {  }

      @Override
      public void windowDeactivated(WindowEvent e) {  }

      @Override
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }

      @Override
      public void windowClosed(WindowEvent e) {  }

      @Override
      public void windowActivated(WindowEvent e) {  }
    });
  }

  @Override
  public void onTemperature(float temp) {
    tempLabel.setText(String.format("%.1f °C", temp));
```

```
  }
}
```

C.3.3 CTKTempProvider.java

```
package ctktest;

import java.awt.EventQueue;

import context.arch.service.Services;

import context.arch.storage.Attribute;
import context.arch.storage.Attributes;
import context.arch.subscriber.Callbacks;
import context.arch.widget.Widget;

public class CTKTempProvider extends Widget {

  private String location;
  private float temperature = 0.0f;

  public CTKTempProvider(String location, int port,
      String id, boolean storageFlag) {
    super(port, id, storageFlag);
    this.location = location;
    findDiscoverer(true);
  }

  public CTKTempProvider(String location, int port,
      String id) {
    this(location, port, id, false);
  }

  @Override
  protected Attributes initAttributes() {
    Attributes atts = new Attributes();
```

```
    atts.addAttribute("temperature");
    return atts;
  }

  @Override
  protected Callbacks initCallbacks() {
    Callbacks ca = new Callbacks();
    Attributes atts = new Attributes();
    atts.addAttribute("temperature", Attribute.FLOAT);
    ca.addCallback(Widget.UPDATE, atts);
    return ca;
  }

  @Override
  protected Attributes initConstantAttributes() {
    Attributes atts = new Attributes();
    atts.addAttributeNameValue("location", location);
    return atts;
  }

  @Override
  protected Services initServices() {
    return new Services();
  }

  @Override
  protected Attributes queryGenerator() {
    Attributes atts = new Attributes();
    atts.addAttributeNameValue("temperature", Float.
        valueOf(temperature), Attribute.FLOAT);
    return atts;
  }

  public void setTemperature(float temp) {
    this.temperature = temp;
```

```
  }

  public float getTemperature() {
    return temperature;
  }

  @Override
  public void notify(String event, Object data) {
    Attributes atts = queryGenerator();

    if(atts!=null) {
      setNonConstantAttributes(atts);
      if(subscribers.size()>0) {
        sendToSubscribers(event);
      }

      store(atts);
    }
  }

  /**
   * @param args
   */
  public static void main(String[] args) {
    int port = 1234;
    final CTKTempProvider tp = new CTKTempProvider("
        fridge", port, Widget.getId("TempProvider", port))
        ;
    EventQueue.invokeLater(new Runnable() {

      @Override
      public void run() {
        new TempUI(tp);
      }
    });
```

```
  }
}
```

C.3.4 TempUI.java

```
package ctktest;

import java.awt.BorderLayout;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.BoxLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class TempUI {
  private CTKTempProvider tp;
  private JLabel tempLabel;
  private JFrame frame;
  private JSlider tempSlider;

  public TempUI(CTKTempProvider tp) {
    this.tp = tp;

    frame = new JFrame("TempUI");
    frame.setDefaultCloseOperation(JFrame.
        DISPOSE_ON_CLOSE);
    frame.setLocationByPlatform(true);

    JPanel tempPanel = new JPanel();
    tempPanel.setLayout(new BoxLayout(tempPanel,
        BoxLayout.PAGE_AXIS));
```

```
JPanel p;
p = new JPanel();
p.add(new JLabel("Temperature"));
tempPanel.add(p);

p = new JPanel();
p.add(tempLabel = new JLabel("? °C"));
tempPanel.add(p);

frame.add(tempPanel, BorderLayout.NORTH);

tempSlider = new JSlider(JSlider.VERTICAL, -80, 400,
    (int)(10 * tp.getTemperature()));
tempSlider.addChangeListener(new ChangeListener() {

  @Override
  public void stateChanged(ChangeEvent e) {
    int v = tempSlider.getValue();
    float tmp = (float)v / 10.0f;
    tempLabel.setText(String.format("%.1f °C", tmp));
    TempUI.this.tp.setTemperature(tmp);
    TempUI.this.tp.notify("update", null);
  }
});

frame.add(tempSlider, BorderLayout.CENTER);
frame.pack();
frame.setVisible(true);

frame.addWindowListener(new WindowListener() {

  @Override
  public void windowOpened(WindowEvent e) {  }
```

```
        @Override
        public void windowIconified(WindowEvent e) {   }

        @Override
        public void windowDeiconified(WindowEvent e) {   }

        @Override
        public void windowDeactivated(WindowEvent e) {   }

        @Override
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }

        @Override
        public void windowClosed(WindowEvent e) {   }

        @Override
        public void windowActivated(WindowEvent e) {   }
      });
    }
}
```

C.4 Temperature Demo TCF

This code was used in the application complexity tests in section 6.1.1 for the TCF values.

C.4.1 CTXTempReader.java

```
package contexttest;

import java.awt.EventQueue;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
```

```
import org.tzi.context.client.ContextClient;
import org.tzi.context.client.ContextClientListener;
import org.tzi.context.client.ContextManager;
import org.tzi.context.client.ContextClient.
    CommunicationState;
import org.tzi.context.common.Context;
import org.tzi.context.common.ContextElement;
import org.tzi.context.common.ContextListener;
import org.tzi.context.common.ContextListenerInterface;

public class CTXTempReader implements ContextListener {
  private ContextClient cc;
  private List<String> tags = Arrays.asList(new String []
      { "temperature" });

  public interface TemperatureListener {
    public void onTemperature(float temp);
  }

  private List<TemperatureListener> listener = new
      LinkedList<TemperatureListener>();

  private void notifyListener(float temp) {
    List<TemperatureListener> tmplist = null;
    synchronized (listener) {
      tmplist = new LinkedList<TemperatureListener>(
          listener);
    }

    for(TemperatureListener l : tmplist) {
      l.onTemperature(temp);
    }
  }
```

```
public void quit() {
  cc.logoutIfConnected();
  cc.terminate();
}

public void addListener(TemperatureListener l) {
  synchronized (listener) {
    listener.add(l);
  }
}
public void removeListener(TemperatureListener l) {
  synchronized (listener) {
  listener.remove(l);
  }
}

private CTXTempReader(ContextClient cc, ContextManager
    cm) {
  this.cc = cc;
  cm.subscribe(Context.ALL_CONTEXTS, Context.
      ALL_SOURCES, Context.ALL_PROPERTIES, tags);
  cm.addContextListener(this);
}

public static void main(String... args) {
  String server = "localhost";
  int port = 2009;
  String user = "tempReader";

  ContextClient cc = new ContextClient();
  cc.init(server, port);
  cc.login(user);

  ContextManager cm = new ContextManager(cc);
```

```
    final CTXTempReader tr = new CTXTempReader(cc, cm);

    EventQueue.invokeLater(new Runnable() {
      @Override
      public void run() {
        new TempReadUI(tr);
      }
    });
  }

  @Override
    public ContextListenerInterface getProperties() {
    return null;
  }

  @Override
  public void processContext(Context ctx, ContextElement
      ce) {
    try {
      float temp = Float.parseFloat(ce.getValue());
      notifyListener(temp);
    } catch(Exception e) {
      System.err.println("Invalid temperature: " + ce.
          getValue());
    }
  }

  @Override
  public void propertyAdded(Context ctx, String source,
      String property) {  }

  @Override
  public void propertyRemoved(Context ctx, String source,
      String property) {  }
```

```
  @Override
  public void sourceAdded(Context ctx, String source,
      String property) {  }

  @Override
  public void sourceRemoved(Context ctx, String source) {
      }
}
```

C.4.2 TempReadUI.java

```
package contexttest;

import java.awt.BorderLayout;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.BoxLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class TempReadUI implements CTXTempReader.
    TemperatureListener {
  private JLabel tempLabel;
  private JFrame frame;

  public TempReadUI(CTXTempReader tr) {
    tr.addListener(this);

    frame = new JFrame("ReadUI");
    frame.setDefaultCloseOperation(JFrame.
        DISPOSE_ON_CLOSE);
    frame.setLocationByPlatform(true);
```

```
JPanel tempPanel = new JPanel();
tempPanel.setLayout(new BoxLayout(tempPanel,
    BoxLayout.PAGE_AXIS));

JPanel p;
p = new JPanel();
p.add(new JLabel("Temperature"));
tempPanel.add(p);

p = new JPanel();
p.add(tempLabel = new JLabel("? °C"));
tempPanel.add(p);

frame.add(tempPanel, BorderLayout.NORTH);

frame.pack();
frame.setVisible(true);

frame.addWindowListener(new WindowListener() {
  @Override
  public void windowOpened(WindowEvent e) { }

  @Override
  public void windowIconified(WindowEvent e) { }

  @Override
  public void windowDeiconified(WindowEvent e) {  }

  @Override
  public void windowDeactivated(WindowEvent e) {  }

  @Override
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
```

```
        @Override
        public void windowClosed(WindowEvent e) { }

        @Override
        public void windowActivated(WindowEvent e) { }
      });
    }

    @Override
    public void onTemperature(float temp) {
      tempLabel.setText(String.format("%.1f °C", temp));
    }
}
```

C.4.3 CTXTempProvider.java

```
package contexttest;

import java.awt.EventQueue;
import java.util.Arrays;
import java.util.List;
import java.util.Locale;

import org.tzi.context.client.ContextClient;
import org.tzi.context.client.ContextManager;

public class CTXTempProvider {
  private float temperature = 0.0f;

  private ContextClient cc;
  private ContextManager cm;
  private String context;
  private String source;
  private String property = "temperature";
```

```
  private List<String> tags = Arrays.asList(new String []
      { "temperature" });

  public CTXTempProvider(ContextClient cc, ContextManager
      cm, String context, String source) {
    this.cc = cc;
    this.cm = cm;
    this.context = context;
    this.source = source;
  }

  public void setTemperature(float temp) {
    temperature = temp;
  }

  public float getTemperature() {
    return temperature;
  }

  public void update() {
    cm.setProperty(context, source, property, String.
        format(Locale.US, "%.1f", temperature), tags, -1);
  }

  public void quit() {
    cc.logoutIfConnected();
    cc.terminate();
  }

  public static void main(String... args) {
    String server = "localhost";
    int port = 2009;
    String user = "tempSensor";

    ContextClient cc = new ContextClient();
```

```
    cc.init(server, port);
    cc.login(user);

    ContextManager cm = new ContextManager(cc);

    final CTXTempProvider tp = new CTXTempProvider(cc, cm
        , "Storage", "Fridge");

    EventQueue.invokeLater(new Runnable() {
      @Override
      public void run() {
        new TempUI(tp);
      }
    });
  }
}
```

C.4.4 TempUI.java

```
package contexttest;

import java.awt.BorderLayout;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.BoxLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class TempUI {
  private CTXTempProvider tp;
  private JLabel tempLabel;
```

```
  private JFrame frame;
  private JSlider tempSlider;

  public TempUI(CTXTempProvider tp) {
    this.tp = tp;

    frame = new JFrame("TempUI");
    frame.setDefaultCloseOperation(JFrame.
        DISPOSE_ON_CLOSE);
    frame.setLocationByPlatform(true);

    JPanel tempPanel = new JPanel();
    tempPanel.setLayout(new BoxLayout(tempPanel,
        BoxLayout.PAGE_AXIS));

    JPanel p;
    p = new JPanel();
    p.add(new JLabel("Temperature"));
    tempPanel.add(p);

    p = new JPanel();
    p.add(tempLabel = new JLabel("?␣°C"));
    tempPanel.add(p);
    frame.add(tempPanel, BorderLayout.NORTH);

    tempSlider = new JSlider(JSlider.VERTICAL, -80, 400,
        (int)(10 * tp.getTemperature()));
    tempSlider.addChangeListener(new ChangeListener() {

      @Override
      public void stateChanged(ChangeEvent e) {
        int v = tempSlider.getValue();
        float tmp = (float)v / 10.0f;
        tempLabel.setText(String.format("%.1f␣°C", tmp));
```

```
        TempUI.this.tp.setTemperature(tmp);
        TempUI.this.tp.update();
      }
    });

    frame.add(tempSlider, BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
    frame.addWindowListener(new WindowListener() {

      @Override
      public void windowOpened(WindowEvent e) {  }

      @Override
      public void windowIconified(WindowEvent e) {  }

      @Override
      public void windowDeiconified(WindowEvent e) {  }

      @Override
      public void windowDeactivated(WindowEvent e) {  }

      @Override
      public void windowClosing(WindowEvent e) {
        TempUI.this.tp.quit();
      }

      @Override
      public void windowClosed(WindowEvent e) {  }

      @Override
      public void windowActivated(WindowEvent e) {  }
    });
  }
}
```