Composite Device
Using a Distributed MVC Architecture

Author: Søren Stiig Hansen (19771130)
Instructor: Henrik Bærbak Christensen

Center for Pervasive Computing
University of Aarhus

3 May 2005
1 Contents

1 Contents ........................................................................................................................................ 2
2 Abstract ....................................................................................................................................... 4
3 Introduction .............................................................................................................................. 5
3.1 Motivation ............................................................................................................................. 5
3.2 Reader’s Guide ....................................................................................................................... 6
3.3 Problem Statement and Scope .............................................................................................. 8
3.4 Scenarios ............................................................................................................................... 9
3.4.1 Scenario 1: Zooming in Pictures ...................................................................................... 10
3.4.2 Scenario 2: Using a PC as a Keyboard for a PDA ........................................................... 10
3.4.3 Scenario 3: Publish a New Service for a PDA on a PC ................................................... 11
4 Analysis ................................................................................................................................... 12
4.1 Distributed MVC Architecture Requirements .................................................................... 13
4.2 Availability ........................................................................................................................... 15
4.3 Discovery Services ................................................................................................................. 16
4.4 Adaptability .......................................................................................................................... 16
4.5 Access Transparency ............................................................................................................ 16
4.6 Location Transparency .......................................................................................................... 16
4.7 Location and Context Awareness .......................................................................................... 17
4.8 Quality of Service ................................................................................................................ 17
4.8.1 Error and Fallout Recovery ............................................................................................ 17
4.8.2 Security ........................................................................................................................... 17
4.8.3 Performance .................................................................................................................... 18
4.9 Framework Qualities ........................................................................................................... 19
5 Related Work .......................................................................................................................... 20
5.1 Distributed MVC Architectural Requirements ................................................................ 20
5.2 Composite Device Environment ......................................................................................... 20
5.3 Service Oriented Architecture ............................................................................................. 21
5.4 Discovery Service ................................................................................................................ 22
5.4.1 CORBA .......................................................................................................................... 25
5.4.2 Jini .................................................................................................................................. 25
5.4.3 Universal Plug and Play ................................................................................................. 25
5.4.4 J2EE Architecture .......................................................................................................... 26
6 Architectural Design ............................................................................................................... 27
6.1 Architectural Vision .............................................................................................................. 27
6.2 Architecture Decisions .......................................................................................................... 28
6.3 Architecture Description ....................................................................................................... 29
6.3.1 Logical View .................................................................................................................. 29
6.3.2 Process View .................................................................................................................. 31
6.3.3 Development View ......................................................................................................... 32
6.3.4 Physical View ................................................................................................................ 34
6.4 Discovery Services ............................................................................................................... 35
6.5 Middleware ........................................................................................................................... 36
6.5.1 Connection .................................................................................................................... 36
6.6 Quality of Service ................................................................................................................. 37
6.6.1 Failure Model .................................................................................................................. 37
Content

Composite Device Using a Distributed MVC Architecture

6.6.2 Security Model ................................................................. 37
6.6.3 Performance ................................................................. 38
7 Component Design ............................................................... 39
7.1 Basic MVC Pattern ............................................................. 39
7.1.1 Coupling ................................................................. 41
7.2 Distributed MVC ............................................................... 41
7.2.1 MVC Structure ............................................................. 42
7.2.2 MVC Component Factory .................................................. 44
7.2.3 MVC Structure Handling .................................................. 45
7.2.4 MVC Builder ............................................................. 46
7.3 Session Manager ............................................................... 47
7.3.1 Session Manager Control View ........................................... 47
7.4 Session Manager Proxy ........................................................ 47
7.4.1 PDA Controller View ....................................................... 48
7.5 Design of Model ............................................................... 49
7.6 Design of View ............................................................... 49
7.7 Design of Controller ........................................................ 51
8 Implementation ................................................................. 55
8.1 Runtime Environment ....................................................... 55
8.2 Selection of Platform ........................................................ 55
8.2.1 Key Functionality of the Prototype ..................................... 56
8.2.2 Limitations of the Prototype ............................................... 56
8.2.3 Using the Framework ........................................................ 57
9 Discussion and Conclusion .................................................. 58
9.1 Discussion ................................................................. 58
9.1.1 Evaluation Against Problem Statement ................................ 58
9.1.2 Evaluation Against Fundamental Requirements ....................... 60
9.2 Future Work ................................................................. 60
9.2.1 Discovery Services .......................................................... 60
9.2.2 Context Awareness ......................................................... 61
9.2.3 Security ................................................................. 61
9.3 Conclusion ................................................................. 61
10 Definitions ................................................................. 63
11 Abbreviations ................................................................. 64
12 References ................................................................. 65
13 Links ................................................................. 67
14 Appendices ................................................................. 68
14.1 Source Code Documentation ................................................ 68
14.2 Used Frameworks ........................................................... 68
14.3 Source Code Organization ................................................ 68
14.4 How to Install the Prototype ............................................. 69
14.4.1 Configuration of the WLAN ............................................. 69
14.4.2 Installation on a PC ....................................................... 70
14.4.3 Installation on a PDA ..................................................... 70
14.5 How to Use the Prototype .................................................. 70

Author: Søren Stiig Hansen
Revision: 3.92  Date: 24-05-2005
Page 3 of 72
2 Abstract

This thesis evaluates whether a distributed MVC architecture is a feasible way to incorporate available computing resources in the surrounding area into a composite device for solving specific tasks.

When a PDA uses a service from a nearby PC, e.g. using the screen of the PC, they together form a composite device. The goal of this is primarily to overcome the physical limitations of the PDA. The infrastructure for managing composite devices will be described and analyzed.

To put the architecture into context, a number of usage scenarios are described, tested and used for validation.

To support the architecture, a proof of concept prototype is developed. The developed prototype implements a distributed MVC architecture with the following key functionalities implemented:

- Ability to dynamically switch between one service and another internally in the PC or in a composite device composed of a PC and a PDA.
- Ability to dynamically switch between service types (between MVC trinities)
- Registration of available services
- Discovery of available services
- Infrastructure to handle creation of a composite device in the form of a MVC structure composed of a mixture of internal and external MVC components

The view and controller services in the MVC architecture are abstractions of corresponding visual objects, e.g. PC screen and PC keyboard, making it easy for the user to understand.

It is concluded that the distributed MVC Architecture represents an improvement of the user’s possibilities and an improvement over standard service architecture, due to the support for building composite devices and administrating the relations between the components in the MVC triad.

**Keywords:** Pervasive computing, composite device, MVC, middleware, distributed MVC architecture
3 Introduction

3.1 Motivation

In the last couple of years the world we are living in has become more and more pervasive with IT in for example: cars, PCs, PDAs, washing machines, high resolution monitors, TV sets, mobile phones etc. The wireless connectivity of computing devices has become a fast growing and successful technology where the use of mobile phones, PDAs, and portable PCs with integrated wireless networking are becoming the majority of computing devices. The sale of portable PCs has in 2004 in Denmark overtaken the sale of stationary PCs and this trend is expected to continue [L8]. Global shipments of mobile devices as a whole (wireless and non-wireless) grew by 51 percent in last quarter of 2004 [L8].

PDAs have their strengths in mobility and size, but have a number of weaknesses: limited screen and interaction abilities (no real keyboard), slow network connection to surrounding computing devices, limited resources, easily misplaced, and access to network varies depending on its physical location [R4].

The growth in use of small portable devices like PDAs together with their improvement in wireless capabilities makes it timely to consider development of their interaction and co-operation with other computing devices. Until recently, computational devices had been stand-alone applications running on a single computer or being based on the client-server paradigm.

In different scenarios within the intelligent home, home entertainment, healthcare and many more, there is a real need for PDAs and PCs or computers embedded within devices to communicate with each other in order to solve specific tasks. When PDAs and PCs together solve specific tasks by cooperating, they are forming a composite device. A composite device is a composition of available hardware resources that surround the user’s current location. One of the primary purposes of the co-operation is to overcome the physical limitations of PDA. The PDA screen is in its nature limited. When the screen of a nearby PC is more suitable, then why not enable lending that resource out to the PDA? This is, in other words, a kind of resource sharing. The constellation of participating devices depends on the user’s current position and situation.

The key focus point in this thesis is developing an architecture that enables tasks, which are not well suited for the PDA to be outsourced to more appropriate devices. This thesis suggests this to be handled by implementing an architecture enabling the PDA to discover, request and use services from the computing devices in its surroundings. The architecture must contain a multi-tier communication infrastructure allowing a PDA to communicate with the surrounding computing devices. Furthermore, the architecture must make it easy to publish and subscribe to services like displaying on a screen nearby. For users of the system, it is essential that the computer technology is almost invisible and seamlessly supports tasks and work flows.

As described in the Book of Visions 2000 [L9]:

*The value of communications technologies is often said to grow proportional to the square of the number of the connected devices. Therefore the universal wireless interconnectivity emerging from today’s mobile Internet core networks will be one...*
crucial task. To offer the right level of support for the various specialized radio interfaces and terminals will be a key requirement.

The real challenge, as described above, is making all those devices into composite devices that interact in an intelligent way and implement a middleware, which allows this to be possible.

As described in [R20]:

The way mobile computing devices and applications are developed, deployed and used today does not meet the expectations of the user community and falls far short of the potential for pervasive computing. A vision of pervasive computing is described, along with attributes for a new application model.

The above article has its focus on how to make computing devices able to take advantage of the services provided by computing devices in the surrounding area. This must be implemented so that resources are shared between client devices and servers. The problems identified in the article are quite similar to those analyzed in this thesis.

The research and consulting company Gartner has made an investigation for the Danish Ministry of Science and Technological Development, which describes Government requirements for IT, architecture [R25]. The background for these requirements is existing requirements developed by members of the EU, the USA and other countries. The key recommendations and arguments in this survey are:

- An IT architecture should support the use of PDAs and Mobile computing devices.
- For development of new systems, Service Oriented Architecture (SOA) is recommended. The reason for this is that SOA is excellent for building composite applications by re-using existing services instead of building a completely new system. The Government’s main goal is not to reduce the price but to reduce risk and improve quality of service.

Composite devices are a very interesting topic in their own right, but there is also a need for the technology [R25].

In the section Service Oriented Architecture it is argued that Distributed MVC Architecture is build on top of SOA, which makes a good combination.

### 3.2 Reader’s Guide

There are a number of different terms commonly used in this field e.g. Composite Device Computing Environment (CDCE), pervasive computing (“IT in everything”), ubiquitous computing etc. The CDCE environment is discussed in [R16]. I have chosen the term CDCE to be used in this thesis because this thesis is primarily about making a composite device infrastructure or environment. Other terms used in this thesis are defined in section entitled Definitions.

In terms of architecture, it is simple to have “IT in everything”, but the big challenge is to enable cooperation between computing devices, thus making them composite. The composite devices in this context have the primary purpose of doing tasks not well suited for a PDA alone.
In this document, the term computing device will be used, meaning a single computer or a PDA or anything with a computer embedded.

I have used the following kinds of references:
- [Qx] Questions this thesis will try to answer.
- [Rx] References to literature as articles or books. See section entitled References
- [Lx] Links used on the WEB. See section entitled Links.

All design will be documented using UML 2.0. For more details on UML 2.0 notation please refer to [R18].

The thesis is organized into a number of chapters:

**Introduction**
This chapter contains the motivation for the thesis, the problem statement and definition of a number of key scenarios. The scenarios are used to put the thesis in a context and further to validate whether the suggested architecture is adequate. Furthermore, it describes the workflow and structure of this thesis.

**Analysis**
This chapter contains an analysis of the problem statement with focus on identifying which fundamental requirements or architectural qualities are most central when developing a distributed MVC architecture. The architecture will thereafter be evaluated against these architectural qualities.

**Related Work**
This chapter compares the identified architectural fundamental requirements to those identified in related work, and further discusses related projects or frameworks in general.

**Architectural Design**
This chapter describes the architectural vision and the design needed to satisfy the identified fundamental requirements and software qualities.

**Component Design**
This chapter describes the component design needed to satisfy the architectural design and vision.

**Implementation**
This Chapter describes the implemented prototype.

**Discussion and Conclusion**
This chapter summarizes the thesis and concludes whether the developed proof-of-concept prototype satisfies the identified requirements for a distributed MVC architecture being able to build composite devices.

**Definitions**
This chapter contains definitions of terms used within CDCE.
3.3 Problem Statement and Scope

The main problem this thesis tries to answer is: “Is it possible to create a framework for a distributed Model View Controller (MVC) architecture which enables incorporation of computing devices in the user’s near surroundings, when solving a specific task, and thereby to overcome the physical limitations of a PDA?”

One of the issues is how to reconfigure the roles in the architecture at runtime, maintaining a way for the devices to still solve specific tasks. This could either be switching components or the whole service. Switching components would be e.g. to switch between two views without changing the model and controller. One service type could be a picture service where the PDA is controller of a PC screen nearby or a keyboard service where the PDA uses the PC keyboard to enter text into itself. This is referred to as switching service type.

This main problem can be separated into a number of sub-issues or questions. The questions will be referred to as [Qx] where x is the number of the question. The questions will be analyzed and answered one by one:

[Q1] Is the distributed MVC architecture suggested by Bardram and Christensen in [R4] a feasible architecture for a CDCE and what are the quality attributes and trade-offs of this architecture?

[Q2] Is it possible, with a distributed MVC architecture, to enable a PDA to be a “remote controller” of a screen nearby or using a PC keyboard as input device?

[Q3] Is it possible at runtime to reconfigure the composition of the composite device and e.g. use another display or another input device? Or in more general terms, how to dynamically reconfigure the MVC roles?

[Q4] Is it possible to construct the distributed MVC architecture so that it enables the MVC roles to communicate in an efficient way?

[Q5] Is it possible to make the distributed MVC architecture so that it satisfies the expectations of a framework?

[Q6] Is it possible to give the distributed MVC architecture the ability to easily add and use new service types?

The main focus of this thesis is the architectural challenge. Several other relevant and essential challenges, such as security, location and context awareness will not be covered in terms of implementation. The choice of focus on architectural challenges has been done in a way which should not limit future work. The selection is primarily based on interest and available time. Within the area of mobile devices, the described prototype covers only testing with a PDA and not mobile phones etc.
The development of the thesis is as follows:

1. Problem definition
2. Analysis of important aspects and identification of fundamental requirements for the distributed MVC architecture
3. Design of the necessary architecture and infrastructure
4. Component design
5. Implementation of a proof-of-concept prototype
6. An evaluation of the developed prototype against other related works
7. Finally a conclusion on the problem under investigation

### 3.4 Scenarios

This section will describe a number of scenarios. Each scenario has been validated by the proof of concept prototype developed in connection with this work.

The scenarios are used to define the context for the use of CDCE and to make the discussions more concrete. Some screen shoots from the developed prototype are shown together with the scenarios.

In a pilot project, in one of the wards of Bispebjerg hospital in Copenhagen, the staff are investigating registration and analysis of the food intake of patients. This is done in order to optimize the nutritional state of the patients. A good nutritional state will secure that the patients are fit and properly prepared before an operation and best able to recover afterwards. Nurses use PDAs in order to collect information about food consumed, by type and amount. The intake is collected either by scanning a barcode on a food item or by keying it in. The plan is to use the PDA as remote control for a screen nearby for illustration to interest, the patient in evolution of their food intake. In this scenario it is a precondition that each nurse is equipped with a PDA affording wireless access to the hospital patient information system. The PDA can request composite elements in the room of the patient visited. A composite element is a screen nearby or a keyboard connection to a PC nearby. I will not cover further details about the hospital domain and their workflow.

The scenarios described in the following subsections are generalized not including any description of a potential domain. The first scenario e.g. is zooming in a picture where a picture could be a graph of a patient’s food intake or an X-ray, etc.

The domain for the scenarios could as well be “networking in a hotel” as described in [R1]. The zooming could be used by guests at the hotel looking through freshly taken holiday photographs on the TV screen in their hotel room. The TV would need to contain a computer. New TVs will contain computers and hard drives for recording instead of using a traditional video recorder. The composite device will be the TV and a PDA where the PDA will be the remote control. When the hotel guest checks in, he or she will be registered as a user of the system and added to the Access Control Lists (ACL) enabling this to happen. This could be quite realistic in the domain of home entertainment.

The scenes of the described scenarios are in closed environments at hospitals, hotels, and work places or in private homes. In those environments it is natural that the users in advance are registered in the system ACLs or similar. This means that the context is not spontaneous networking, as used in e.g. an airport etc.
The central point in the suggested architecture is the MVC and the following scenarios will consider the use of these roles.

### 3.4.1 Scenario 1: Zooming in Pictures

The scenario supports the display of a graph of a patient’s food intake, or an X-ray, or holiday pictures on the TV set in a hotel etc.

The model is a collection of pictures. It should be possible to show these pictures on a PC or TV screen and zoom in on these pictures by means of tapping on the PDA screen. The View is deployed on the PC, the controller is deployed on the PDA and the picture is contained in a model deployed on a PC server somewhere. The issue is to find a way to overcome the limitation of the small screen on the PDA. The Figure 3-1 below shows a picture on the PC screen where zooming is controlled by the PDA.

![Figure 3-1 PDA Controls Zooming in a Picture](image)

### 3.4.2 Scenario 2: Using a PC as a Keyboard for a PDA

This scenario supports the case where a nurse must key in the food intake for a patient at Bispebjerg hospital when the food is without barcode. This scenario enables the nurse to use the keyboard on a PC nearby instead of the limited facilities on the PDA.

The model is the text keyed in. The view is the visual display of the text. The PC keyboard is the controller.

The issue is to overcome the limitation of the PDA not having a real keyboard.
Figure 3-2 above shows a situation where the PC keyboard is used for keying text into the PDA.

### 3.4.3 Scenario 3: Publish a New Service for a PDA on a PC

The services contained in the developed prototype are either a keyboard service or a picture view service. This scenario must demonstrate how easy it is to add and publish a new service enabling the new service to be used from a PDA. The purpose of this is to analyze whether this prototype has the qualities expected of a framework.

The users of this scenario are developers of new services to be added.
4 Analysis

Before analyzing the problem domain relevant for the context of this thesis, there are a few terms need to be defined. Some terms are solely defined in the section entitled Definitions, and the most important terms are defined when used for the first time.

One of the important concepts within distributed architecture is the concept of a service. A service is an entity that can be used by a person via a program, or another service. A service is a self-contained, stateless function, which accepts a request through a well-defined interface and performs a unit of work. Two examples of services are zooming in a picture via a picture service, or feeding characters into a PDA via a PC keyboard service.

The terms service and resources are defined by [R1] as:

The term service is used for a distinct part of a computer system that manages a collection of related resources and presents their functionality to users and applications.
The term resource is a rather abstract one, but is best characterizes by the range of things that can usefully be shared in a networked computer system. It extends from hardware components such as disks and printers to software-defined entities such as files, databases and data objects of all kinds.

The term session is defined by [L11] as:

A session is the activity that a user with a unique IP address spends on a Web site during a specified period of time. The period of time a user interfaces with an application. The user session begins when the user accesses the application and ends when the user quits the application.

The term session is often used in the literature but rarely defined. In the context of this thesis a session starts when a user, by means of a discovery lookup service, starts a session by using one of the offered services and continues until the use is ended. One fundamental issue concerning handling a session, is whether it includes handling a state of the session or not. If the service can be implemented as stateless, it is easier to combine with other services. For each of the terms service, resources and session, the above definitions will be used.

The architecture of a system is defined by its components and the structure of the components. As described in [R1]:

An architectural model of a distributed system is concerned with the placement and the relationships between them.
Fundamental models are concerned with a more formal description of the properties that are common in all of the architectural models.
Three models address these issues:
- An interaction model
- A failure model
- A security model

The interaction model classifies processes as server processes, client processes and peer processes, the placement of components across the network and the interrelationship between...
components. The interaction model also defines a number of system attributes e.g. whether the system is synchronous or asynchronous. The interaction model will be defined by the fundamental requirements identified.

In a distributed system, both processes and communication channels may fail. The failure model defines the ways in which failure may occur and is handled. The failure model will be defined in terms of quality of service.

The security model describes how security can be achieved. Defining a security model includes defining a threat model in order to identify which types of attacks the system might be exposed to and evaluation of the risks involved from these attacks. The security model will be identified in terms of quality of service.

### 4.1 Distributed MVC Architecture Requirements

Besides the standard characteristics of a normal MVC pattern, a distributed MVC pattern must also enable the MVC roles to be deployed on different nodes. This requires a middleware supporting interaction between the components playing the MVC roles.

The term distributed system is defined by [R1] as:

> The definition of a distributed system is a system in which components located at network computers communicate and coordinate their actions only by passing messages. The sharing of resources is a main motivation for constructing distributed systems.

Distributed systems exist in two variants: synchronous distributed systems or asynchronous distributed systems. In a synchronous system, there is defined bound on: time to execute each step, the transmission time between nodes and the drift of each node’s local clock. Synchronous systems can be build but this is difficult and rarely done. Synchronous systems are meet within the embedded world. The distributed system described in this thesis is an asynchronous system with no bounds on execution speed, message transmission delay and clock drift rates. This exactly models the Internet.

In the section entitled Problem Statement and Scope the key questions [Q1] to [Q6] were introduced briefly. In the following the questions will be discussed in more detail and it will be defined which requirements or quality of service the prototype must implement to solve the problems raised by the questions.

The section entitled Motivation states the need for a way to separate a computing device into a number of logical services supported by the architecture, where the services can be used by the individual computing devices in the user’s surroundings. As described in [R19] a computing device is organized with one or more CPUs, memory and a number of I/O devices. The I/O devices can be separated into input and output devices. Input devices are keyboard, mouse, scanner, PDA stylus etc. Output devices are screen, persistent store (primarily hard drive), printer etc. The architecture must support splitting a computing device into a number of logical services. The model is a part of the executing application implementing the “Model Service”. The executing application is executed by the CPU and loaded into memory from the persistent store. A given input device represents the controller and an output device represents the view. The way the architecture is distributed is as described in the section entitled Motivation. A
PDA and a PC can combine to make a composite device. The nearby PC represents the view, the PDA represents the controller and the model can be anywhere and avoid where it is most convenient for the usage.

The MVC pattern contains three roles: a model, one or more views and corresponding controllers. This pattern when used in this context is special in the sense that there is no obvious need for more than one view subscribing from the model. This is due to the fact that it is all about making a composite device where the nearby PC is the view and the PDA is the controller. There is no limitation in the design or prototype concerning this. Support for more views could easily have been added but this is not the focus. One view can be exchanged with another view deployed on another node, but there will only be only one active view at a time, corresponding to one model and one controller.

In order to fully support the scenarios described in the Scenarios section, and besides the six key questions, I have identified the following list of fundamental requirements or qualities the architecture must support. These fundamental requirements will now briefly be defined and with further details provided in the following sections.

**Availability** – the architecture must support the mobile computing devices being able to connect via a wireless network to the nearby PC for using the display for viewing. Obviously it only makes sense to control a view in the same room as the person controlling it. Further the architecture shall include a middleware with a high availability enabling distributed devices to communicate, regardless of whether or not they are mobile.

**Discovery Services** – the architecture must support means for computing devices to discover what services are available in the network to which they are connected and to investigate their properties. A discovery service must offer two interfaces: a registration service and a lookup service.

**Adaptability** – the architecture must be able to handle changes in system configuration and in number, or type, of resources available. This means managing service elements and being able to exchange elements dynamically. It must be possible to move a service from one device to another.

**Access Transparency** – the architecture must enable local and remote-computing devices to access services using identical operations.

**Location Transparency** – the architecture must enable computing devices to access services without knowledge of the location of the service in question.

**Location and Context Awareness** – the architecture must support the computing devices recognizing their location and being aware of the context in which they function.

**Quality of Service** – is generally defined as reliability, security and performance. The architecture shall support ability for reconnection in case of an error or fallout and shall clearly support security – especially within a healthcare environment where patient data ay be at risk. It is critical to achieve fast response times and in some cases also high
data transfer rates if e.g. the X-ray must be transferred to another node containing the view.

**Framework** – the architecture shall have the qualities of a framework. One must expect new devices and services to be added at high rate over the lifetime of the system so it must be prepared for frequent modification.

### 4.2 Availability

The Distributed MVC Architecture must support mobile devices. PDAs are by nature mobile and the architecture must enable the PDAs to connect to other devices participating in the MVC by means of a suitable network type. For the architecture to support the scenarios it is vital that it is easy to connect to the network. The nodes in the system are either connected with cables or by a wireless link.

Mobile Computing Devices like PDAs use a Wireless Local Area Network (WLAN) defined by IEEE standard 802.11. The 802.11 standard intends to support wireless communication between computers within about 150 meter at a speed up to 11 Mbps. The IEEE 802.11 network uses radio frequency signals in the 2.4 GHz band. Ethernets Carrier Sense Multiple Access/Collision Detection (CSMA/CD) mechanism is different when using wireless instead of wires. The carrier sensing and collision-detection are only effective when the strength of signals is approximately the same throughout the network. In general, failures in wireless networking are due to fading, hidden stations and collision masking. A slot reservation mechanism is added to the Media Access (MAC) protocol. The CSMA/CD scheme is thereby changed to a CSMA/CA scheme, which is called Carrier Sensing Multiple access with collision avoidance. A Request To Send frame and a Ready To Send frame do this. For more details please see [R11]. A wireless network that connects to a conventional LAN through an access point is known as an infrastructure network. Ad hoc networks do not include an access point or base station. Add hoc networks are built on-the-fly as a result of mutual detection by two or more mobile devices with wireless interface in the same vicinity.

Other Computing Devices like mobile telephones connect via Wireless Application Protocol (WAP). Again other devices use IR or Bluetooth.

A challenge in the area of wireless networks is the limited connectivity due to the user not always being connected when moving from one location to another, depending on the coverage of the wireless network.

The distributed MVC architecture shall enable the role of the MVC architecture to live in different nodes and to interact through a middleware. The Middleware includes mapping of request to services via the concept of a session.

The types of network that support a distributed system at a specific geographic location at a hospital, at a hotel or at a person’s home are mostly a combination of LANs and WLANs because of a more suitable access radius. The Middleware shall provide infrastructure to access computing devices located on a LAN from a WLAN.
4.3 Discovery Services

The architecture shall support the mobile computing devices in discovering which resources or services are available. It shall be possible for the PDA to use a new service without recompiling. This is on the condition that the PDA is enabled for a similar service which implements the same service interface. A discovery and lookup service is a requirement, regardless of how the architecture of an offered service in a distributed system is divided.

The architecture shall support mapping of requests to services. When a mobile computing device requests a service it must be executed if available. It must also be possible to change from one service to another service dynamically e.g. exchanging one view service with another view service of a different type or at a different location.

The architecture shall support ability for more than one service type by handling pools of services organized by the type of service e.g. picture services.

The need for a discovery service is also identified by: [R4][R1][R2].

4.4 Adaptability

This requirement means being able to manage a pool of services and being able to switch from one service to another service. This could e.g. be exchanging one view component with another at the given location. The elements in the composite devices are managed based on location-dependent information.

The number of computing devices surroundings us are expected to grow radical and thereby realizing the visions in [L9] and also expected by [L8]. A system, which supports a growing number of computing devices and services traditionally, implies a lot of configuration and administrative maintenance. The traditional solution seems very inefficient. There is a real need for dynamic adaptation. This can be achieved in a flexible and scalable way by implementing it as a discovery service.

A key attribute is that others are able to modify the framework for usage within the domain in question and as described by [R4]:

*Modifiability: We think that by employing the MVC pattern as cornerstone in the architecture, the task for writing viewers and controllers to new devices is a manageable task.*

4.5 Access Transparency

Access Transparency enables local and remote resources to be accessed using identical operations.

4.6 Location Transparency

Network resources and services have to be accessed via a service locator without knowledge of their location. This has to be transparent for the user of the service.
4.7 Location and Context Awareness

Context information is about places, people and things; where are they located and what are they doing. The selected service contains a model, which controls the workflow and the state of the model and thereby handles the Context Awareness. A specific service is chosen due to the physical location of the node offering the service. A PDA may need to use a PC nearby as view, but it makes no sense if the PC is located in another room. For implementing location awareness some location-tracking components are needed which can be either sensor based (e.g., GPS, RADAR, sound, IR, active badges [R22] or using specific location information available through the communication system (e.g., radio propagation measurements). Given this information simple location awareness services can be realized.

The location information will be of the form of a location identifier. The means for getting this information through a location information system is not covered by this thesis.

The MVC architecture should enable the use of composite devices based on location-dependent information. This functional requirement is in [R3] referred to as:

Location and context awareness - the architecture should support that a mobile device knows its location and its context.

4.8 Quality of Service

The main non-functional requirements, which affect the quality of a systems service, are reliability, security and performance. The quality of service covers the failure and the security model.

Reliability is critical in the design of most systems and strongly related to the failure model.

The performance aspect is defined in terms of ability to meet timeline guarantees. The service requested response must be within a specified time-frame.

4.8.1 Error and Fallout Recovery

The correct operation of a distributed system is threatened whenever a fault occurs in any computing device on which it runs or in the network that connects them. When the distributed system is asynchronous it allows no assumptions about the time intervals involved in any execution. In case of a failure, a relevant error messages will be presented for the user and the system must enable the user to try again. If the failure originates from broken hardware, and in the case of no redundancy, a replacement is necessary for completing the service operation.

4.8.2 Security

When considering security issues, there are three different topics which must be considered [R1] [R28]:

- Threat model – what are the threats to the system in question and which attacks on the system are possible?
- Security policy – a security policy is a result of the need to protect the integrity and privacy of information and other resources belonging to individuals and organizations. Based on the threats and e.g. users’ expectations of the system what do we as a provider decide to be our protection strategy?
- Security mechanisms – are the mechanisms, which implement the security strategy.
Security threats can be divided into a number of classes [R1]:
- Leakage - acquisition of information by unauthorized people.
- Tampering - unauthorized alteration of information
- Vandalism – interference with the proper operation of a system

A threat model for a wireless distributed system needs to handle the following threats [R27]:
- The privacy and integrity of communication is an obvious concern in wireless networks. It is obvious that if security is compromised an intruder can gain access to personal X-rays or to an intelligent house getting remote control over the functions of the house including the door locks.
- Any station that is within range and equipped with a receiver or transmitter might seek to join or interrupt the network. It might eavesdrop on transmissions between stations. IEEE 802.11 standard addresses some of these problems. IEEE 802.11 can be configured to require an authentication exchange for each station joining the network in which knowledge of a shared key is demonstrated. This is an instance of the stream cipher technique, which uses shared key authentication.
- Denial of service by flooding a channel with messages in order to deny access for others.
- Individuals within the system obtaining access to data for which they should not be allowed.

A security policy is specific for the environment in which the system is used and the policy is therefore different from company to company. A security policy will always have to obey the laws of the country defining how to handle and register personal data. I assume that the security policy defines that the identified threats must be handled.

Analyzing the threats I have identified the following security mechanisms needed:
- Authorization (access control) – assures that only authorized people get access to data
- Integrity (cryptographic checksums) - assures that received data has not been tampered with.
- Confidentiality (encryption) - assures that the transmission cannot be eavesdropped

Security is not the main focus in this thesis and will only be covered to the extent of assuring that it can be added later on.

4.8.3 Performance

The performance aspect of quality of service is defined in terms of responsiveness and computational throughput and is highly related to the architectural model of the system. The service types described in this thesis are both user interactive services, where fast response on, e.g. a zooming action, is needed. Traditionally, response time is defined as the interval from when a user initiates a request to the instant at which the first part of the response is received by the application.
4.9 Framework Qualities

First of all I will define the term framework, which originates from different authors: 

\textit{A framework is a set of cooperation classes that make up a reusable design for a specific class of software} [R8].

\textit{A framework is the skeleton of an application that can be customized by an application developer} [L10].

The domain in this case is a distributed MVC architecture supporting building composite devices. The user of the framework is an application programmer doing the customization.

The above framework definitions will be used to evaluate the prototype developed versus the scenario described in section Scenario 3: Publish a New Service for a PDA on a PC.
5 Related Work

In this chapter other related works are described and analyzed. These related works has served as inspiration for this investigation.

5.1 Distributed MVC Architectural Requirements

It can be difficult to have focus on a large number of fundamental requirements or qualities. Some requirements can be included as sub requirements in other requirements. As an example could I justify that availability includes discovery service. How can a service be available if another computing device is not able to locate the service? Fundamental requirements are composed differently in different literature and can therefore be difficult to compare.

In [R4] the following architectural fundamental requirements are identified: adaptability, availability, security, modifiability and performance. Taking the above fundamental requirements one by one and compare these to the fundamental requirements and qualities identified in section Distributed MVC Architecture there are all covered by this thesis.

The following list document how the architectural fundamental requirements from [R4] are mapped to what I have identified in section Distributed MVC Architecture:

- Adaptability – maps exactly to adaptability
- Availability – maps exactly to availability
- Security – maps to part of quality of services
- Modifiability – maps to framework qualities
- Performance – maps to part of quality of services

As described in the section Motivation and Problem Statement and Scope one of the main focus areas in this thesis is to validate the architecture suggested in [R4] by means of a prototype.

5.2 Composite Device Environment

The investigation focus at Siemens Corporate Research (SCR) described in [R2] and [R21] is as follows:

*The fundamental idea is to avoid having to use a single PDA to perform all tasks.*

Observations have led us to investigate approaches for overcoming the inherent display limitations of a PDA. In the Composite Device Computing Environment (CDCE) project the surrounding available computing resources are considered as another facet of situated computing. Having acknowledged the limitation of the PDA, the CDCE framework provides mechanisms for seamlessly exploiting and interacting with available surrounding computing resources (e.g. PCs, workstations, TVs telephones) to augment the PDA.

To achieve the above they introduce an intelligent gateway. The gateway manages a pool of services, establish composite devices based on location information, maps PDA requests to services and perform dynamic conversions needed to present the information on the selected output node. This is implemented by the means of the Distributed Component Object Model (DCOM). DCOM is selected to fulfil these requirements and the client side are purely a Web
solution. The motivation in Siemens investigation is similar to the motivation in this thesis. When comparing Siemens suggested solutions and this suggested solution they are primarily different in selection of programming model. This thesis suggests a solution based on an application framework instead of a WEB framework to be able to satisfy requirements as performance. This is done because a middleware-based application is much faster than a standard Internet base application. The reason is that the Internet Based protocols are text based with a lot of mapping overhead in both ends. The text-based protocols are not based on strongly typed interface and therefore lacks of compile and runtime error checking. Instead text based protocols have advantages of creating a loose coupled connections.

5.3 Service Oriented Architecture

A good architecture emphasizes the separation of responsibilities. For example in a presentation tier which manages the presentation components; a business logic tier managing the business logic component; and a data access tier managing the data access components. This organisation is showed in Figure 5-1.

![Figure 5-1 A typical 3-tier Application](image)

A 3-tier application traditionally has a lot of communication channels to different business objects, which makes the client code responsible for traversing complex objects models and understanding details about domain specific logic. The connection between applications or between application and business objects are tight coupled with CORBA, RMI or .NET Remoting etc. Objects are marshalled across process boundaries through the proxy/skeleton techniques.

![Figure 5-2 A Service Oriented Application Architecture (SOA)](image)
A Service Oriented Architecture (SOA) model introduces an extra layer of indirection by means of a “service layer”. The services in the service layer can avoid some of the pain associated with traversing complex object models and provide a black-box functionality. A SOA service is a self-contained, stateless function, which accepts a request and returns a response through a well-defined interface.

In a SOA clients consume services instead of invoking discreet method call directly on business objects. SOA is in this way an evolution compared to normal 3-tier architecture adding this facade hiding details of the business objects. This situation is shown on Figure 5-2.

The interface to the service in question is loosely coupled and relies on universally accepted standards such as XML. The openness of the technologies makes it open for implementation in different platforms.

A SOA is usually comprised of three primary parties: a producer (of services), a consumer (of services) and a directory (of services).

SOA is an enterprise architecture and begins with the need for the enterprise; an application server etc.

For more information about SOA please refer to [L16].

The distributed MVC architecture described in this thesis offers a Composite Device structure based matching a composite service structure. The described connections between the MVC components are loose coupled as in a SOA with a text based protocol though not XML structured. This is shown in Figure 5-3 above.

5.4 Discovery Service

Discovery service is a must for enabling device in a composite device structure to discover and thereby use each other’s services. In the following discovery service in general and a few of the most known discovery services frameworks are described.
Looking at discovery services at a high level show that they are relatively similar and include how needed services may be discovered on demand with minimal prior knowledge of the network. Typically clients can search for services by type or by descriptive attributes.

As described in [R20]:

*Applications and services live in the surrounding physical distributed environment. A discovery mechanism allows a mobile device to dynamically identify and enumerate the applications and services its local vicinity. The major challenge posed by dynamic discovery is the definition of a service adaptation layer.*

The above referred need is just one of many requiring this fundamental need for discovery services within this area of CDCE.

Discovery services are closely related to name services. Name services are used to hold the addresses and other details of users, computers, network domains, services and remote objects. A name service maps a name to attributes. As an example when using the Internet, the Internets Domain Name Service (DNS) is used. DNS maps domain names to the attributes of a host computer by means of its IP address.

Within the context of name services the term name space is used defining the collection of all valid names recognized by a particular service. This term is closely related to the term pervasive space described in [R24] as:

*The intelligence is embedded into the system around the user also known as “Pervasive Space”.*

The pervasive space is the environment in which a task executes e.g. view a picture and relies on service composition that creates customized services from existing services. The basic fundament for this is discovery of services and binding a service name to a specific service context (service type and location).

A name service stores collections of <name, attribute> pairs, and how the attributes are looked up from a name. A discovery service is defined in [R1] as:

*A discovery service is a directory service that registers services provided in a spontaneous environment.*

*A directory service stores collections of bindings between names and attributes and looks up entries the match attribute-based specifications.*

Recent developments within the area of discovery services includes the following discovery protocols:

- Service Location Protocol (SLP), The SLP is a decentralized, lightweight, scalable and extensible protocol for service discovery within a site developed by The Internet Engineering Task Force (IETF). See [L13]
- Jini, Sun's Java-based approach to service discovery. See [L5]
- Salutation, architecture for service discovery and service management. See [L14]
- Microsoft's Universal Plug and Play (UPnP)
- Bluetooth Service Discovery Protocol (SDP)
- IPv6, the "next generation Internet" protocol designed by the IETF and is expected to include a discovery service. See [L15]
The specification of IPv6 include the following two Request For Comments (RFC): RFC2165 and RFC2608. RFC2165 and RFC2608 define a service location protocol:

*The Service Location Protocol provides a scalable framework for the discovery and selection of network services. Using this protocol, computers using the Internet no longer need so much static configuration of network services for network-based applications. This is especially important as computers become more portable, and users less tolerant or able to fulfill the demands of network system administration.*

The intention is to include discovery service in IPv6 standard but is still an open issue being a request for comments.

All discovery protocols have similar system architectures but differ in their functionality, network transport protocol, and the possibility of code mobility. See [L5] and [L6] for further details. The research and development work on service discovery protocols is not completed yet.

Besides discovery protocols there are a number of middleware services enabling communication in a distributed system. Examples of middleware services not implementing discovery but just being middleware are:

- Common Object Request Broker Architecture (CORBA) - Object Management Group (OMG)
- Java 2 Enterprise Edition (J2EE) – from Sun Microsystems
- Java 2 Standard Edition (J2SE) containing Remote Message Invocation (RMI) – Sun Microsystems
- .NET Remoting - from Microsoft
- Distributed Component Object Model (DCOM) - from Microsoft

The different middleware services solve similar problems but are implemented differently. Except for CORBA all the services are platform dependent in its basic form. When e.g. using J2EE it is possible to added in middleware like Common Network Services (CNS) to supplement Java Messaging Services (JMS) to make a bridge to the C programming language. The most important and ambitious middleware is CORBA. CORBA is a product developed by over 700+ companies in a consortium – called the Object Management Group (OMG).

All discovery services are faced with the problem of how do clients locate the lookup service? In some cases the lookup service is configured in advance or done by using multicast to a well-known IP multicast address. In the last case the multicast address has to be known by all instances of clients using published services. A potential client multicast the service name to be resolved and only the server that holds the named attribute responds. Resolving means translating the service name into the named resource or object and thereby the service is discovered.

In a distributed system security is very important and almost all mentioned discovery services have a number of security issues currently active.

In the following sections the most important available frameworks within discovery services will be described. There is a lot of common ground among the various choices in service discovery frameworks. All support the concepts of client and service. Clients perform
5.4.1 CORBA
CORBA defines what is called an open object bus that defines the components that live within it and how they interoperate. The object bus is based on an Object Request Broker (ORB). The ORB lets objects located locally or remotely make requests to and review response form other objects located at the bus. Before invoking the operations of a CORBA object, a caller must first obtain a reference to the object. The reference is then used to locate and contact the object. In CORBA, objects are located by their Interoperable Object References (IORs). The ORB implements the discovery service by providing an API to convert an object reference to a string and vice versa.

CORBA is platform independent by having an Interface Definition Language (IDL) defining the bindings between objects and their interfaces. CORBA makes tight coupling between applications and services. It is not possible to change a IDL without recompiling both the service and the user of the service.

5.4.2 Jini
The Jini architecture allows printers, storage systems, speakers and any other piece of hardware to be plugged directly into a network. Each pluggable device announces itself immediately to a network device registry. Jini is a discovery service designed for spontaneous networking. It is entirely Java-based and requires that a JVM run in all of the computers. The JVM allows the computers to communicate with one another by means of RMI. Jini provides services discovery, transactions, shared data spaces called JavaSpaces and events.

The discovery component in Jini is a LookUp Service (LUS). The lookup service allows services to register the services they offer and clients to request services that match their requests. Jini provides a flexible definition of a service. A Jini service can be hardware based such as printing or fax, or completely software based e.g. a service for rendering images. Upon initialization, a Jini service uploads a proxy object to one or more lookup services to announce its availability to other Jini entities. A Jini service contains a catalog of these proxy objects. Clients discover catalogs of services, provides the interfaces that defines the needed services and download matching proxy objects. The search for a Jini service is therefore primarily based on object class matching.

Jini is the only of the discovery technologies discussed in this thesis that relies on code mobility – moving pieces of executable code around the network. This code mobility is the reason for requiring a JVM running in all of the computes involved. Jini further depends on the use of RMI for communication between different computing devices. The way the service is invocated is by ordinary Java method calls on the service supplied on request. The service proxy supplied communicates with the real remote service.

5.4.3 Universal Plug and Play
Universal Plug and Play (UPnP) allows computers to discover and use network-based devices. As described in [R26]:

discovery in order to find services. Either the clients find the service themselves or they consult a service catalog administrated by a server. Even though the following discovery service frameworks are relatively similar at a high level they have incompatible design philosophies and are based on different technologies.
The Universal Plug and Play (UPnP) specification describes a set of protocols for enabling networked entities of all sorts to initialize autonomously and then discover and share one another’s services.

The UPnP specification only addresses protocols expressed entirely in XML and do not specify a particular API or programming language. The UPnP specification addresses six areas: addressing, discovery, description, control, event handling and presentation. The UPnP specification dictates that an UPnP device must have an IP address.

In the following I will shortly describe the discovery part. For discovery UPnP uses the Simple Service Discovery Protocol (SSDP). SSDP messages are transmitted using UDP multicast. UPnP is a capability that allows devices on a network to discover other devices and determine how to work with them. A computer can use UPnP to detect whether there are printers on the network and learn how to use them. When an UPnP-capable computer boots, there may already be devices on the network that it can use. To determine whether this is the case, the computer sends a broadcast request asking that any UPnP-capable device within earshot responds directly to it and provides information about using it. This announces its presence on the network and invites computers to make use of its services. When an UPnP device wishes to leave the network it sends one “bye bye” message.

Windows ME and XP include native UPnP support. Microsoft’s UPnP implementation still has a number of security issues active and is working to get those fixed. The issues concern how an attacker can gain complete control over an affected system or how to affect a system from providing useful service by a denial of service attack.

5.4.4 J2EE Architecture

Java’s platform: Java 2 Enterprise Edition (J2EE) is designed for developing component-based enterprise applications. The J2EE platform proves a complete set of services to those components. The J2EE platform is defined by a specification and a reference implementation. The J2EE platform includes an application server, a development environment and a large number of services including:

- Middleware based on RMI or CORBA but can be extended with network protocols from different vendors
- Naming service: Java Naming and Directory Interface (JNDI)

J2EE components normally use Java Naming and Directory Interface (JNDI). JNDI is the native naming service of the J2EE platform providing a standard interface for locating users, machines, networks, objects and services. JNDI is a bridge between naming and directory services and can be configured also to use other naming services e.g. as Lightweight Directory Access Protocol (LDAP). The benefit of JNDI is that you only need to learn a single API to access all sorts of directory services. The naming service will therefore use the same procedure to retrieve object references, regardless of whether those objects are hosted on the J2EE platform or CORBA platform.
6 Architectural Design

This chapter gives an overview of the distributed MVC architectural design. Details on component design are described in the chapter entitled Component Design. The design for a distributed MVC architecture must satisfy the fundamental requirements or qualities specified in the chapter Analysis.

The Focus is on the structure, relationship between the MVC components and how the MVC components interact.

Dividing software architecture into layers is one of the most important techniques to break a complicated software system into understandable units. Making this division in layers has a number of important benefits, which will be describe in session entitled Development View.

The failure model and security model will be covered in the section entitled Quality of Service.

6.1 Architectural Vision

In order to support the scenarios described in the Scenarios I envision the following architecture that are displayed in Figure 6-1 below.

![Figure 6-1 Architectural Vision](image-url)

The PDA get connected to the LAN via a wireless connection to the Broadband router. Services are found and resolved by a lookup service deployed on a background server. The lookup service maps directly to functionality provided e.g. a picture or keyboard service.

It is obvious that the background server is a single point of failure. Two or more servers operating in a cluster would solve this and further more could take care of load balancing. An example of a cluster system is the J2EE Enterprise framework from SUN Microsystems. This framework is described in section entitled J2EE Architecture.
A typical scenario could be as follows:

PDA gets connection with the background server via a wireless connection and request a service where e.g. the PDA controls the screen of the nearby PC for displaying and zooming in a picture. The Background Server contains the model, the PDA is the controller and the PC Screen is the view. The PC could also function as a TV proxy and send the picture signal to a large TV on the wall.

### 6.2 Architecture Decisions

Two completely different architectural software models exist: a centralized model and a fully distributed model each with its own pros and cons. In other words the architecture can be implemented by letting the processes have a client-server relationship or a per-2-per (P2P) relationship or a combination of the two.

The centralized model is characterized by having one or more centralized servers containing the services offered to the different clients. This architecture contains a single-point-of-failure and the central server is also usually a bottleneck. The single-point-of-failure can be omitted by letting a number of servers operate in a cluster environment. If configured with load balance abilities it also takes care of the bottleneck problem. A cluster solution fits the purpose in case of a hotel or healthcare environment and should be within the economically possibilities. In a home environment there is expected to be some single-point-of-failures due to most people only having one PDA and one network, which is usually acceptable. This means no redundancy.

The advantage with a central solution is that it is more simple to overlook and easier to debug. When everything gets distributed and each computing device’s available services has to be mirrored to all other computing devices the simple overview is lost. A clean P2P solution is difficult to make and it is almost impossible to give guaranties for throughput and prevent deadlocks and alike.

Architectural decision due to the argumentation in the present section is as follows:

- The ability to handle the discovery service is made distributed in the sense that all server components have the ability. When using the discovery service it is configured which discovery service to use and in case of a failure another discovery service at another server can be used. In case of an environment where a switchover cannot be tolerated like in a hospital in some situations the discovery service must exist in a cluster environment.

- The architecture is divided so all portable devices have a client relationship to the stationary devices which are servers. This way of dividing the roles in the architecture has the function of delegating responsibility for bootstrapping a session. The only real difference is that the computing device playing the server role has full functionality concerning controlling a session where the computing device playing the client role has a proxy delegating the session control to the server.
6.3 Architecture Description

The Distributed MVC Architecture implemented in the prototype is described by the 4+1 View Model of Software Architecture introduced by Philippe Kruchten. This use of multiple views on architecture allows addressing the various concerns separately e.g. the functional and non-functional requirements. For more details concerning the 4+1 View Model please refer to [R17].

In turn the distributed MVC architecture will be described from each of the following views:

- Logical View
- Development View
- Process View
- Physical View (also called deployment view)

For each view its purpose and concerns will be addressed. The “+1” view is the scenarios that bind the views together. The scenarios are described in section entitled Scenarios.

The Figure 6-2 below shows the 4 + 1 views and includes a short description on the usage of the views.

![Figure 6-2 The 4+1 View of the Architecture](image)

6.3.1 Logical View

The logical view primarily supports the functional requirements – what the system should provide in terms of services to its users.

The logical view shows classes, associations, inheritance and compositions. The following figure shows the logical view. The logical view is divided into a number of namespaces and each namespace has its own concern. The logical view contains two superior namespaces, which are structured in the same way: the PDA and the server. Typically the PDA will request a service from the server. Each of the two superior namespaces contains the following namespaces:

- **MVC namespace** – defines the interfaces of the triad MVC components in general for all offered services. This means that e.g. the general model interface covers what is equal for a picture model and a keyboard model etc. The MVC namespace contains an
abstract factory being able to create a factory specific for the service requested. The specific factory is contained together with the specific service components. To assure that the MVC components fit together construction of them is delegate to an MVCBuilder. Furthermore a MVC structure object that is able to initialize (enable) MVC components in the right order or disable.

- **Picture namespace** – contains the picture service components. This means different triads of PictureModel, PictureView and PictureController. Furthermore a PictureMVCFactory who can create all of the picture components.
- **Keyboard namespace** – contains the keyboard service components. Organized as the picture namespace.
- **Session namespace** – contains primarily the session manager and session request.
- **Discovery namespace** – contains the discovery service including a service locator and a location locator and defines a location.
- **Connection namespace** – handling the TCP/IP protocol layer. This layer includes a server and a client connection.

Dependencies between namespaces or classes or between a namespace and a class are shown with dotted lines. The arrow indicates who are depending of whom (Who knows who). A dotted line without arrows is equal to a dotted line with arrows in both ends. This indicates that the peers know each other.

The picture and the keyboard namespace are structured in the same way and therefore only the picture namespace is showed in details.

The logical view only shows the essential classes and namespaces. The organization of namespaces in the PDA is structured in the same way as the server and is therefore not shown in detail.
6.3.2 Process View

The process view shows processes, tasks, threads and message exchange. The process view addresses non-functional requirements such as performance, availability, concurrency, and distribution.

Due to the IO-bound operations both client and server side application are implemented as multithreaded applications. When the PDA has contacted the session manager for a specific service it will shortly afterwards receive the location of the service requested. Then the PDA can contact the computing device containing the requested service. This is the reason why the process view on Figure 6-4 shows two server processes. The only difference between the two separate server applications is that only one of them has the network services configured. The system can of course have more than two servers.
The SessionManager on the server side has a running thread waiting to accept external calls from clients. In the server there are three threads: a GUI thread, a thread handling the server connection and a thread handling the incoming requests. The ServerConnection spawns a thread for each incoming request.

The PDA application contains one main application with one GUI thread. When a service request is issued a ClientConnection thread will be spawned to handle the communication with the server. The ClientConnection thread is not depicted on the process view due to the dynamical nature. A new ClientConnection will be created each time a new service request is issued. The dynamics of how to make a service request within a session is described in a later sequence diagram in section entitled Distributed MVC.

**Figure 6-4 Process View**

### 6.3.3 Development View

The development view focuses on software module organization and on the software development environment. The development view has its focus on the layers the software is broken into.

Breaking a system into a number of layers has a number of important benefits when designed with the focus of minimizing dependencies between layers:

- A single layer can be understood without knowing the details of other layers
- A layer encapsulate the functionality of the layer
- A layer can be substituted with an alternative implementation
- A layer and the layers below can be changed and compiled without compiling the above layers. Circular dependencies are difficult to handle and must be avoided.

The only cons creating an extra layer is that it has a minor performance penalty due to doing an extra indirection.
The development view shows namespaces, subsystems, and compilation dependencies. The dotted lines with arrows define dependencies. All dependencies go from a top layer to a layer below. No dependencies go upwards.

Software architecture traditionally is divided into the following three layers:

- **Presentation or HMI layer**
- **Business layer containing the real point of the system**
- **Communication layer communicating with other systems that carry out tasks on behalf of the application**

As shown in Figure 6-5 below the business layer is divided into a session and a service layer and a bottom layer is added containing the used framework services.

![Figure 6-5 Development View](attachment:image.png)
6.3.4 Physical View

The physical view is also called the deployment view. This view takes primarily the non-functional requirements into account such as availability, reliability (fault tolerance), performance and scalability.

A PDA application will exist in all the PDAs in the system and a server application will exist in all the stationary computing devices where a component exists to be lent out for being part of a composite device. See figure 6.6 below. The figure shows one PDA and two servers but there is no limitation on these numbers in the system.

The physical view shows codes, communication mediums (LAN, WLAN), and physical subsystems. The server IP address has to be known by the PDA client.

The servers in the system wait for session request(s) from clients. The clients know the location of the session manager that contains the discovery service. The location of the clients is unknown to the session manager. All servers contain a session manager, which as minimum knows the services contained in the context of the session manager. Servers can be contacted directly from a client and they together can form a composite device. Either the client can have all available servers configured internally or only the central one. If it is the first option the PDA contacts the server needed directly or if it is the last option the PDA contacts the discovery service and gets the location of the server requested and afterwards contacts this server directly. The system developed can handle both situations.

![Figure 6-6 Physical View](image-url)
6.4 Discovery Services

One of the key fundamental requirements identified for making the distributed MVC Architecture work is the ability to discover which services are available and where. The Discovery service must therefore provide a registry of services available.

Within the discovery service there is a need for the following functionalities:

1. Knowledge of the present context in form of knowing this location specification
2. Ability to register services or delete service registrations in a service registry.
3. Knowledge of the location of a specific service by looking up the service in a service registry.

This can be separated in two concerns: a service registration concern and a service location lookup concern.

The Locator handles both the service registration and service lookup concern. The Locator contains a registry of all services available in the distributed architecture. The Locator has delegate the location concern to a LocationLocator who knows the location of all available services.

When a new service is created e.g. when a new view component is added to the distributed architecture, the MVCFactory in question registers the service by the Locator in the context.

When a PDA e.g. wants to show a picture on a PC it has to issue a session request to the PC in question. A session request is sent via the PDAs internal SessionManagerProxy who forwards the request to the SessionManger in the PC. A session request contains a service type e.g. the picture service and location specification of the requested components (location of model, view and controller). When a service is discovered the service can be addressed directly on a peer-to-peer basis. A session request is shown in Figure 6-7 below.

![Figure 6-7 Session Request](image-url)
This system has one central session manager who serve as the place to get discovery information. All PCs are designed as server side computing devices and have a session manager who potentially could work as the central session manager. It is the same code base but only the central session manager has all discovery information configured. All computing device including the PDAs have all their own services registered by the local locator. The session manager and the MVCBuilder are both using the Locator to discover which services they can offer.

The discovery service implemented has the ability to extract external service locations by means of an XML specification.

### 6.5 Middleware

Middleware is defined in [R1] as:

> A layer of software whose purpose is to mask heterogeneity and to provide a convenient programming model to application programmers. Middleware is represented by processes or objects in a set of computers that interact with each other to implement communication and resource sharing support for distributed applications.

The above definition will be used in this thesis.

#### 6.5.1 Connection

The .NET Framework offers a number of different technologies to invoke methods on remote objects:

1. DCOM enabled remote communication among COM objects
2. .NET Remoting
3. Web services. Web services use the Simple Object Access Protocol (SOAP) to invoke objects remotely. The SOAP protocol packages going back and forth between the client and the server are using XML. SOAP does not mandate a specific transport protocol, but the preferred one is HTTP 1.1. SOAP does not cover issues as security or transactions.
4. Sockets on top of TCP/IP.
5. Universal Plug and Play (UPnP) protocol suite where SOAP is part of the suite.

DCOM is an old technology and not recommended by Gartner Research for future developments. See [R25].

The natural choice in a distributed MVC architecture based on objects in a non-Web application is the .NET Remoting, but this is not supported by Microsoft compact framework (Windows CE and alike).

SOAP could be an attractive alternative but is not lightweight due to the need for XML generators and parses on client and server sides, and just sending one byte character of payload can use 1000 bytes of message. SOAP is of course a generally accepted standard.

Socket on top of TCP/IP is a possibility. With a text based lightweight protocol it can work efficiently and do the job. Further it would later be possible to wrap the text based protocol with SOAP as long as the protocol is kept structured.

UPnP is a suite of protocols with SOAP as one of them.
The main purpose of this thesis is to discover how the MVC architecture supports CDCE and as a first step I will go for socket on top of TCP/IP with a text based protocol leaving the possibility to convert to SOAP on a later stages if times allows it.

In case of a picture service the connection between the PDA, where the controller is deployed, and the PC, where the view is deployed, is based on a WLAN. This implies two execution environments: one with a view deployed on a PC and one with a controller deployed on a PDA.

### 6.6 Quality of Service

The failure model and security model is defined by the quality of service.

#### 6.6.1 Failure Model

As middleware sockets on top of TCP/IP is selected. See section entitled Connection.

The TCP layer include mechanisms for sequencing, flow control, retransmission, buffering and checksum. TCP guarantees message delivery. Omission failures are handled by retransmissions and ordering is handled by sequencing. The checksum is used to detect and reject corrupt packets.

TCP/IP is not a complete reliable communication service due to the dependency on the quality of the physical network. It does not guarantee to deliver messages in the face of all possible difficulties. It is possible to make a reliable communication based on TCP/IP where either a message is delivered or the failure is detected.

When a connection is broken the process using the connection will be notified when the process tries to write or read from the network stream. The process must then take the necessary actions and make the user able to reconnect, if possible. If the physical network is broken e.g. due to a broken router or cable the physical damages must be repaired and the user can reconnect. If it is essential not to have any broken connections the physical network and the system in general must be made redundant so no single point of failure will exist.

#### 6.6.2 Security Model

As described in section entitled Security in the chapter entitled Analysis the privacy and integrity of communication is an obvious concern for a wireless network and the security mechanisms to be addressed are: authorization, integrity, and confidentiality.

I have assumed when deploying the WLAN that is deployed as an extension to an existing wired LAN.

Authorization and access control are assumed handled by the system which this described system extends. The entry point to the network is through a computing device and doing login. Login is a process that identifies the authenticity of a user based on the credentials he or she provides for example username and password. The system defines which information and resources in the system the user is authorized to use. This means when a user request using a PC screen from a PDA the user will during this session be requested to login.
Concerning the security risks that the WLAN adds an attacker must be near enough to interact with the network infrastructure. An attacker in Aarhus cannot attach a system in Copenhagen. A user must be authenticated before being granted full access. The 802.1x authentication occurs when a client first joins a network. All other traffic is blocked at the access point. Furthermore an attacker must be able to break the cryptographic stream of data and have knowledge of the service protocol between the MVC components.

The 802.1x provides enhanced security for existing infrastructure, and confidentiality on the network. 802.1x is a fully extensible and robust authentication mechanism that allows infrastructures to authenticate users [R29].

As described in [R7] the WEP protocol is still considered insecure due to the improper use of initialization vectors defined by the WEP protocol. This is recently detected by a cryptanalyst at the University of California at Berkeley. IEEE is currently working on extensions to the WEP to strengthening the security mechanisms. Based on the above using WLAN must still be considered as adding some security risks to an exciting system.

Integrity and confidentiality is handled by configuration of the 802.1x standard part of the WLAN with the security risks described above.

### 6.6.3 Performance

The performance issue depends a lot on the placement of the individual MVC components and this is addressed in the section entitled Distributed MVC.
7 Component Design

7.1 Basic MVC Pattern

The MVC pattern involves three roles [R9].

- The model represents information about the domain and is a non-visual object containing all the data. The model component encapsulates core data and functionality. The model is independent of specific output representations and input behavior.
- The view represents the display of the model and is solely about display of information.
- The controller accepts input by its event-handlers, interprets the event, and activates a service method on the model. The model executes the requested service, which results in a change in its internal data.

There are two principal separations in the MVC pattern: separating the presentation from the model and separating the controller from the view. A key point in this separation is the direction of dependencies. The presentation depends on the model but the model should be entirely unaware of the presentation being used. This both simplifies the tasks and makes it easier to add new presentations later on.

The model notifies all observers registered about a change in the model by calling their update methods. If the user makes a change to the model from one presentation, all views need to be updated. To handle this without creating a dependency this is implemented as the observer pattern [R8]. The observers are the view(s) and the observable is the model. If the observer pattern is implemented with a push strategy so that the data changed are supplied in the update call. Alternatively the observers can get the changed data by calling a get data method on the observable. The MVC pattern is described in details in [R9] and briefly in [R8].

One of the benefits of using the MVC pattern described in [R9] is that:

*The separation of control aspects form views supporting the combination of different controllers with a view. Also called “pluggable controllers”.*
This exactly fits the purpose to plug in a PDA as controller for a view displayed at the PC nearby.
Each view has an associated controller component. Most GUI frameworks combine the view and the controller. This is very handy when the application including all the three roles is deployed in one computing device. In this case where the roles shall be separated it is difficult to use the GUI frameworks due to this feature.

**Figure 7-2 MVC Architecture Basic Sequence Diagram**

Figure 7-2 MVC Architecture Basic Sequence Diagram shows how the primary objects of the MVC architecture collaborate. The scenario works in the following steps:

1. The PDAApplication requests the SessionMgr to create a session including a model, a special PDAView and a PDAController. The session request could of course specify another combination of model, view and controller or a complete different service.
2. The SessionMgr responds to the request by creating a PDAView and the PDAView attaches itself to the model.
3. The SessionMgr create a PDAController and the PDAController attaches it selves to the model.
4. Then normal MVC operations can occur.
5. Then the PDASession wants to update the session and exchange the PDAView with a PCView.
6. The SessionMgr unloads the PDAView, which detaches itself from the model.
7. The SessionMgr creates a PCView and the PCView attaches itself to the model.

As described in the above initialization steps the controller is attached to the model and the view enables event processing and delegates these events to the controller. The view of the application specifies the behavior of the system in response to user actions. The underlying platform delivers every action from a user as an event. The controller receives and interprets these events using a dedicated procedure.

The basic MVC operations are shown as the last three steps in Figure 7-2 MVC Architecture Basic Sequence Diagram:
8. The user makes some action on the user interface which initiates the controller to call a service on the model.
9. The model changes its internal data structure accordingly and calls the Notify method on its selves, which activates call to all the views attached.

The term local in connection with a view or a controller will be used when the MVC architecture is running local inside one computing device. The term remote is used for the MVC components running in a remote computing device.

### 7.1.1 Coupling

One of the important issues is how to decouple the view and controller so it is easy dynamically to exchange one view with another view or one controller with another controller or both at the same time. When using .NET the view and the controller are decoupled by means of defining a delegate in the view, which is implemented in the controller. The principles of delegates make it easy to attach e.g. controllers to a view. Unfortunately it is not possible to define a delegate in an interface so the controller must know the specific view or an abstraction. A possibility is to make an abstract view declaring the delegates but this seems unnecessary and prevents one from using the GUI designer tool in the .NET framework.

### 7.2 Distributed MVC

The distributed MVC pattern shall enable the three roles to exit on different nodes. The view could be on a PC showing a picture and the controller could be on a PDA. The model could be on another PC. In this case the communications between the three roles or components are separated by some kind of network connection.

The first key points to address are:

1. How to implement the connection between the view, controller, and model objects. If they all are deployed in the same runtime environment\(^1\) on one physical computing device it is simple. In the case of this thesis they have to be able to be deployed in different runtime environments on different computing devices.

---

\(^1\) Runtime environment is for .NET is Virtual Execution System (VER) and Java Virtual Machine (JVM) for Java.
2. How to control that the specific views and controllers fit together.
3. How to discover which views and controllers are available at a location.

### 7.2.1 MVC Structure

As described in section entitled Basic MVC Pattern the MVC pattern involves three roles and the component are related to one another as showed on Figure 7-3 below.

When one or more of the MVC components are located on other computing devices a proxy object will be created on the computing device where the component is not located and where the real object is located a skeleton object will be created. The works are same principles as Java’s RMI. This situation is shown on figure 7-4.

When the PDA controls the zooming action on the PC a ControllerSkeleton is instantiated on the PDA side and a ControllerProxy is instantiated on the server side. This situation is shown on Figure 7.5 below.
When the PDA uses the keyboard on the PC the model and view are located on the PDA and the controller is located on the PC. When the controller is located on the PC a ControllerProxy is needed on the PDA side. This situation is shown on Figure 7-6 below.

The overall strategy is that the MVC components existing on another computing device than where the model is placed exists as proxies together. The MVC components placed on other computing devices than the model exists as skeleton component. The communication between the proxy and the skeleton is encapsulated within the proxy and skeleton components.

This strategy can be expressed in the following rules:

- The model component exists where it is placed
- Components exist as a normal or a proxy object when placed together with the model
- A component placed on another computing device than the model is split in two objects: a proxy object and a skeleton

Performance wise there is some issues needed to be addressed depending on the placement of the MVC components. If the scenario is zooming in a large bitmap file like an X-ray it can introduce a performance bottleneck if the view and model are placed in different computing devices. Each time a zoom command is issued the picture in the model will be changed, the view will be notified and the view will fetch the picture from the model making the picture being transmitted over the network. In the zooming scenario it is realistic that the PDA is only playing the roles as controller and there is no reason why the model and view cannot be placed together. Another solution would be having the model contain a register over all X-rays and the view should only have the X-ray transmitted once and could handle the zooming internally. If the model and the view are places on the same computing device or if the view only gets the X-ray...
ray transmitted once the only thing occurring on the connection layer is to transmit zoom in or out or zoom factor change events.

### 7.2.2 MVC Component Factory

One of the key issues in the architecture is how to create a new session type and how to create a view and a controller, which first of all fits the specific model (Session Type) and secondly fits together. Analyzing the problem and relating it to the patterns specified by [R8] the following creational patterns could be used: Abstract Factory, Builder or Factory Method. The Builder pattern is to be used when creating complex objects and have the same construction process manage to create different representations of the objects in question. The different MVC objects are not complex but the issue is to assure that the objects fit together. The intent of the Abstract Factory pattern is to provide ability to creating families of related objects [R8]. An example of families of objects is user interface toolkits that support multiple look-and-feels. Another example within the context of this domain is MVC components within a specific service. When e.g. a controller for a picture service have the same interface as a controller for a keyboard service they can be considered as related objects but from different families. Having an abstract factory to create a “Model View Controller Pair Factory” or another service factory solves the issue concerning how to assure that the model fits together with the view and controller.

The responsibility of the specific service factories is to assure that a pair of view and controller fits together with the model in question is it e.g. the responsibility of the PictureMVCFactory to assure that the model, view and controller fits together within the family of picture service.
The PictureMVCFactory will assure that the model supporting the view and controller will be created on request.

All MVC components implement both a generic component interface and an extended component service interface special for the service in question. Within the factory namespace the common component interfaces are defined: IModel, IView, IController. A e.g. picture controller will further more implement an IPictureController interfaces including what is special for a picture controller.

If any unsupported combination of view and controller location is received in a service request a FactoryException will be thrown containing a proper error message.

### 7.2.3 MVC Structure Handling

I have encountered the following problems:

- **Problem:** When a specific MVC triad is created the components have to be attached to each other in an exact order each time they are created not to get any race conditions. First the model has to be in place, next the view and last the controller. Then the triad is ready for events to be received by the controller that calls services on the model etc.

- **Problem:** When the specific MVC components are created it depends on the location of the other components if the component to be created is a normal component or proxy components or skeleton component.

Summarizing on the MVC component creation process as described until now. The first step is depending on the service type to be able to create objects within the family of the specific service type e.g. picture or keyboard service. Next step is to create the three objects in question. Creating each of the objects alone is simple and will be handled by the specific component factory. Between the steps there is a need for controlling the composite MVC structure and to known if it is a normal or a proxy or a skeleton component. Analyzing the problem again and relating it to the patterns specified by [R8] the Builder pattern comes in handy.

A complex object in the present context is a composite object and when creating a composite object the same process will create different representations depending on the location of different components. The function of the Builder is to know the structure of the object and whether the objects are normal, proxy or skeleton objects. The creating of the specific object is still handled by the MVC Component Factory. The Builder pattern thereby solves the two described problems.

The description of the design of the Builder is in the following section entitled MVC Builder.
7.2.4 MVC Builder

The MVCBuilder administrate the references between the individual components and handles switching from one set of MVC components to another set of MVC components. Figure 7-9 above shows the design of the different controllers. When the MVCBuilder e.g. has a reference to an IController it do not know whether it is an IPictureController or an IKeyboardController. The MVCBuilder references to the IView and IModel work after the same principle.

When switching between two MVC structures all components are first disabled and when all the new components are in place they are enabled. Components not being exchanged are reused e.g. if it is only the view that is exchanged the model and controller are reused from last MVC structure.

The MVCBuilder only enables a new triad MVC if all components are available at their specified locations. It does not start up if e.g. a view if the corresponding controller is not available.

One of the key problems is to be able dynamically to exchange one MVC component with another MVC component as described in [Q3]. The MVCBuilder makes sure components can be exchanged due to not having a permanent binding between the Builder and the component implementation.
7.3 Session Manager

The session manager is responsible for receiving internal or external session request. A session request specifies the requested service type and the locations of the individual MVC components. The session manager knows the active session and its specification. The session manager will accept any session request that is doable by closing the old session and starting the new one. A session will always occur in one physical room and all components from one triad MVC are used in that specific location (Room). The user can decide whether a session switch is suitable due to having all physical components within eyesight.

A session request specifies three services in a combination. These services are all part of the MVC triad. This way a session request provides an abstraction placed over the abstraction of a single service. In this respect the session manager function as a facade to the services. As described in [R8]:

*Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.*

When a session request is accepted the locations of the requested services are returned to the requester who by use of the supplied information can establish a peer-to-peer connection with the services in question.

If the service is not located in the same context as the session manager, the client must repeat the session request to the session manager located together with the service. The service request is used to start up the requested service components where after these are ready to receive e.g. user events. The responsibility for creating and starting the MVC components are delegated to the MVCBuilder.

7.3.1 Session Manager Control View

The session manager is represented by a session control view where the session type, view and controller location can be selected. When selected they need to be activated. See Figure 7-10 below.

![SessionControl](image)

*Figure 7-10 SessionControl*

7.4 Session Manager Proxy

The PDA computing devices have a session manager proxy, which forwards session request to the real session manager. The organization of the session manager proxy is similar to the real
session manager. All local services are registered by the local discovery service. The local MVCBuilder will handle startup of local service exactly as on the server side.

### 7.4.1 PDA Controller View

The session manager proxy is represented by a session service view. The SessionService view is shown in Figure 7-11 below in the left screen shot.

A PDA user can activates a session request by selecting a session type: picture service or keyboard service, specify the location of the view and the controller and issue the session request by tapping on the activate button.

In case of a picture view session the user will have the following possibilities as shown in the SessionService screen shot right in the figure. If the picture view is on the PC and the controller is the PDA then the user input to the PDA controller is via a zoom-in button, a zoom-out button and track bar to change the zoom factor.

When using the PDA to control a nearby screen it would be very useful if it could be done without having to look at the PDA screen. The users natural focus is on the PC screen where the zooming can be observed. The zooming control could be handled by making activations on the PDA display with the stylus as for example:

- A straight upward line equals zoom in
- A straight downward line equals zoom out
- A clockwise circle equals zoom factor incremented with 10%
- A counter clockwise circle equals zoom factor decremented with 10%
The present design is not user-friendly and could be improved by implementing the above example. The primary focus in this thesis is software architecture and making it user-friendlier will not be handled.

The keyboard service is located at the last tab in the tabs in the bottom the screens shoot in Figure 7-11.

### 7.5 Design of Model

The key design issues concerning the model are:

- To enable easy exchange of one model with another model dynamically
- To make it transparent if the switch between models is from an internal to an external model
- To be able dynamically to attach or detach a new view or controller
- Being part of a WLAN with limited bandwidths it is important that large models are not repeatedly transmitted over the WLAN as part of a service but must be kept within the wired LAN.

If the model is a picture model, the model needs to be implemented as a monitor to prevent race conditions due to e.g. a external component running in its own thread requesting the picture when the picture model is in the middle of loading or updating the picture. This goes for all models due to the controller and view running in different threads. The controller is making updates to the model and the view requests a new version of the model. This could happen if the user via the view request a second update before the first update is done.

### 7.6 Design of View

The key design issues concerning the view are:

- To enable easy exchange of one view with another view dynamically
- To make it transparent if the switch between views is from an internal to an external view
- To have low coupling between a view and a controller and between a view and a model. The view can attach itself to the model. The view offers a number of delegates for subscribing to events. The controller in question uses these.

Depending on the context the users want to see models in different ways. Separating the model and the view is therefore always a good idea. The view only contains the GUI components. All logic executed on data is delegated to the controller. An indirect benefit in doing this is that non-visual objects are usually easier to test than visual ones. The separation of the view and the model makes it easy to test the domain logic using a unit test tools.

The following two pictures in Figure 7-12 and 7-13 shows a simulated PDA picture view followed by a normal PC picture view. These are implemented to show the ability to exchange view internal on the PC.

The Picture View on the PDA has been design with the focus on user interactions without the use of the keyboard. The zoom factor can on from the PDA be controlled via a track-bar control instead of keying in the value.
**Figure 7-12** Simulated PDA View

**Figure 7-13** PC Picture View
### 7.7 Design of Controller

The key design issues concerning the controller are:

- To enable easy exchange of one controller with another controller dynamically.
- To make it transparent if the switch between controllers is from an internal to an external controller.
- To have low coupling between controller and view and between controller and model.

The controller attaches itself by the model and adds event listeners to the view. The controller can exchange model and view to other objects of the same kind dynamically.

First of all the picture controller defines the zooming actions.

Only the MVCBuilder and the controller themselves know whether the session involves local or remote controllers. For the other MVC components it is transparent whether it is local or remote.

After the controller is created it must enable subscription on actions performed by the user. The user action comes in form of events. The way to collect these events is by implementing a delegate defined by the picture view. When an event is raised the subscribing class method handler is invoked through the delegate.

The following code snippets show how to implement a delegate to the zooming in and out.

**View:**

```csharp
// Next line defines the delegate type
public delegate void ZoomHandler(Object IPictureView, ZoomDirection zoomDirection);

// Next line defines a reference to the zoomhandler if any
public event ZoomHandler Zoom = null;
```

**Controller:**

```csharp
// Next line defines the signature of the zoomhandler in question
void Zoom(Object o, ZoomDirection dir);

// Next line defines how to start subscription
zHandler = new PictureView.ZoomHandler(this.Zoom);
view.Zoom += zHandler;
```

Delegates are easy to use but tie the view and controller together. It would be much better to be able to define a delegate type within an interface. When the delegate type must be declared inside a class definition the controller need to know the inside of the view instead of just a view interface. Basically you only need to define the signature of the delegate type and it would have been very useful being able to define this in an interface instead of using inheritance. An alternative is to implement this subscription as the observer pattern and not use delegates.

If the PictureController and the PictureView are separated and live in different computing devices the PictureController is made up by two classes: a PictureControllerProxy in one computing device and a PictureControllerSkeleton in the computer device where the
PictureView is deployed. The design of the PictureControllerProxy and Skeleton are shown in the following two UML diagrams.

The communication between the proxy and the skeleton is by means of Sockets on TCP/IP. When invoking a method on a service proxy, the proxy simply initiates a remote communication to the service skeleton object. This invocation pattern follows the same general semantic as the Java RMI model or the .NET Remoting.

```
+SetZoomFactor(in o : object, in zoomFactor : ZoomFactor)
+Zoom(in o : object, in dir : ZoomDirection)

«interface»
MVC::IController
+SetSubscriptionsOnView(in view : IView)
+DelSubscriptionsOnView()

«interface»
IPictureController
+SetZoomFactor(in o : object, in zoomFactor : ZoomFactor)
+Zoom(in o : object, in dir : ZoomDirection)

Controller::PictureController
+SetZoomFactor(in o : object, in zoomFactor : ZoomFactor)
+Zoom(in o : object, in dir : ZoomDirection)
+SetSubscriptionsOnView(in view : IView)
+DelSubscriptionsOnView()
+PictureController(in model : IModel)

Controller::PictureControllerProxy
+PictureControllerProxy(in model : IModel)
+Connect(in hostIP : long, in port : int)
+Run()
+WaitForRequest()
+HandleEvent(in e : string)
-GetEvent(in ar : IAsyncResult)

PictureControllerSkeleton
+PictureControllerSkeleton(in serviceLocation)
-~PictureControllerSkeleton()
```

Figure 7-14 Picture Controller Hierarchy

The picture controller interface: IPictureController defines all public functionality used when controlling zooming in a picture. The PictureController class implements all functionality defined by the interface. The PictureControllerSkeleton class is an extension to a PictureController and uses all the functionality implemented in the PictureController. It interacts the same way as the PictureController with the other relevant MVC components. The extension of a PictureControllerSkeleton is the ability to receive events from a remote PictureControllerProxy. The relationship between a PictureControllerProxy and a PictureControllerSkeleton is via a TCP/IP connection implemented by a server connection and a client connection. The PictureView and the PictureModel is not aware that the PictureController is a remote component.

The initiation and sequence of a session with a PictureControllerProxy and skeleton involved occur as shown on the following activity diagram figure 7-15.
Figure 7-15 Initiate Session

The design must support the MVC triad to be able to be deployed on different computing devices. This means that the communication channels defined in Table 1 below must be in place.

<table>
<thead>
<tr>
<th>Communication Channel</th>
<th>Information</th>
<th>Connection type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between controller and model</td>
<td>Picture Service: Zoom in or out</td>
<td>One way</td>
</tr>
<tr>
<td></td>
<td>Set zoom factor</td>
<td></td>
</tr>
<tr>
<td>Between model and view</td>
<td>Picture Service: Update (model updated) GetPicture (get updated picture)</td>
<td>Bidirectional</td>
</tr>
</tbody>
</table>

Table 1 Communication Channels
Table 2 below specifies the content of a service request from a client to a server. The service request first contains an identification of the service type e.g. ‘P’ for picture service followed by specification of the requested locations of the individual MVC components. A MVC component is specified by ‘M’ for model followed by the names of the computing devices the service is requested upon e.g. ‘PDA’. This is a structured text based protocol, which easily can be converted to e.g. SOAP or a similar protocol by adding XML tags.

<table>
<thead>
<tr>
<th>Service Request Protocol</th>
<th>Client request</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Picture service</td>
<td>“P M&lt;name&gt; V&lt;name&gt; C&lt;name&gt;”</td>
<td>P MPC VPC CPDA</td>
</tr>
<tr>
<td>Keyboard service</td>
<td>“K M&lt;name&gt; V&lt;name&gt; C&lt;name&gt;”</td>
<td>K MPDA VPDA CPC</td>
</tr>
</tbody>
</table>

One of the key features of a text-based protocol is that it results in loosely coupled systems. Tightly coupled systems are usually faster and safer but tend to make component maintenance much more difficult.
8 Implementation

In the previous two chapters the design of the distributed MVC architecture is described on architecture and component level. In this chapter the implementation of the design is described.

A composite device consists of a number of computing devices of which at least one of them is a PDA. This chapter will specify what specific devices are used for testing the design implemented.

A short installation and usage guide for the prototype is described in section entitled How to Install the Prototype and How to Use the Prototype.

8.1 Runtime Environment

The developed prototype has been tested on the following equipment:

      Intel PXA270 63,41 MB RAM  
      WLAN

PC    Zitech Athlon 64 3800+ with Windows XP Professional  
      1024 MB

PC    Medion 2000+ with Windows XP Professional  
      512 MB

WLAN Cisco Wireless-G Broadband Router 2,4 GHz (WRT54G)

The above computing devices are considered as standard hardware.

8.2 Selection of Platform

According to [R12] Bill Gates vision in 1975 was: “A computer on every desk and in every home” and his vision has in the mid-1999 been changed to: “Empower people through great software anytime, any place and on any device” and has resulted in Microsoft’s Compact Framework for supporting mobility. The visions of Microsoft does not cover making it easy to make composite devices and the vision is still solely based on the client-server paradigm. As described earlier the view and the controller are heavily coupled in the Microsoft framework.

I have chosen to use .NET C# because this technology directly supports both the PC and the PDA platform. The prototype could also have been implemented in a mixed environment using Java for the PC and .NET for the PDA or any other combination. This is due to the connection between computing devices being platform independent. This thesis will not try to compare the differences between the Java and the .NET framework. Middleware support for communication between distributed components is in Java implemented by RMI and in Microsoft’s platform implemented with .NET Remoting.

An implementation based on Java including a J2EE application server could also have been a valid selection. A J2EE application server is born with middleware services as pooling, networking and abilities as load balancing. A solution with two application servers running in a cluster would solve the issue with the solution having a single point of failure. Superficially it seems like Microsoft has the best development tools for the client side and Sun with its J2EE
architecture has the best solution for the server side. The combination of these solutions would be a perfect platform for the development described in this thesis. Java do also contains a SOAP implementation in its class library.

Making user interface for the .NET environment for PCs are different from making user interface in the .NET Compact Framework both due to size of screen and different I/O devices but also due to dissimilarity in platforms. One limitation in the .NET Compact Framework is that all GUI applications are full-screen applications, and furthermore not resizable. This makes it impossible to have a small session GUI component and a small picture view controller visible at the same time.

Due to a lot of other dissimilarities between functionality of the .NET Framework and the .NET Compact Framework it has not been possible to use the same code for server and client side. The only way to share code is to make the code compilation depending on some environment variable and compile it once for each of the two platforms.

### 8.2.1 Key Functionality of the Prototype

The developed distributed MVC architecture has the following key functionalities implemented:

- Ability to switch one service with another service internally in the PC or in a composite device composed of a PC and a PDA
- Ability to switch one MVC component with another MVC component
- Registration of available services local within a component or extracted from a XML configuration file
- Discovery of available services by contacting the session manager
- Infrastructure to handle creations of a composite device in form of a MVC structure composed by a mixture of internal and external MVC components

The prototype has implemented necessary functionality to execute the scenarios:

- Scenario 1: Zooming in Pictures
- Scenario 2: Using a PC as a Keyboard for a PDA
- Scenario 3: Publish a New Service for a PDA on a PC. This is described in section entitled Using the Framework

### 8.2.2 Limitations of the Prototype

The discovery service is implemented with the ability to register or delete a service registration. This facility is in the prototype only used for registration of internal service but nothing prevents the discovery service from receiving registration from external components. Furthermore external service can be registered and extracted from an XML file.

The scenarios I have tested with the prototype all resulted in a composite devices consisting of two computing devices. This is not a limitation and a composite device could easily consist of three computing devices. In the Scenario 1: Zooming in Pictures a situation could occur where the model view and controller should be on each own device. This situation is not tested with the prototype.
8.2.3 Using the Framework

To add a new MVC component to the developed prototype the following steps must be executed:

1. Extend and implement the generic MVC component interface e.g. the IController interface. This include making a specialized interface e.g. a IPictureController interface.
2. Make the component able to live both on the PDA and the PC platform. Implementing a component, proxy, and skeleton version of the component does this. This includes making the specific request responses interaction by use of a connection between proxy and skeleton.
3. Add the components implemented to the namespace in question and make the factory register the new components by the discovery service.

To add a new service in form of a new MVC triad to the prototype the following step must be executed:

1. Implement a specific MVCFactory by implementing the IMVCFactory interface.
2. Follow the steps to create a new MVC component for each of the MVC triads.
3. Add creations of the new MVCFactory to the Abstract factory.

The components implemented in the prototype as the discovery service, the two connection types, the SessionManager, the SessionRequest, the MVCBuilder, the XML service location extractor, the concept of factories etc. can be reused unchanged.
9 Discussion and Conclusion

9.1 Discussion

In connection with this thesis a proof of concept prototype has been developed implementing a distributed MVC architecture, which supports building composite devices.

The main purpose of the prototype is to evaluate whether it satisfies the problem statement and furthermore live up to the identified fundamental requirements of a distributed architecture as described in the chapter entitled Distributed MVC Architecture.

9.1.1 Evaluation Against Problem Statement

The first point in the evaluation is to draw a conclusion about the problem statement, by addressing each of the questions from section entitled Problem Statement and Scope:

Is the distributed MVC architecture suggested by Bardram and Christensen in [R4] a feasible architecture for a CDCE and what are the quality attributes and trade-offs of this architecture?

- A distributed MVC architecture is an extension to Service Oriented Architecture as the analysis in the section entitled Service Oriented Architecture concluded. The components of the MVC architecture make a service composition mechanism by creating a larger service. The MVC architecture provides the means to create a composite device, for which there is no obvious support in SOA.
- Within the focus of overcoming the physical limitation of a PDA or other small screen device by building composite devices to solve the specific tasks, the architecture fits its purpose. The distributed MVC architecture makes a good separation of concerns and makes the structure supporting the visual physical devices e.g. screen and keyboard, making the service easily understandable by the user.
- Some services less obviously fit into a pattern of the MVC architecture. A printer could be separated into similar components where the printer is the output device equal to the view; the PDA works as the controller and the model is the document. An example where the MVC architecture does not fit is if the PDA was a controller for a washing machine. This combination does not have an obvious view and model. Another example of a service not fitting is a “background rendering image service”. The MVC architecture is a good extension for some services, and other services that do not fit could be supported as stand-alone services.
- The architecture is decomposed into independent services where one service is not dependent on the state of other services. This enables the use of services as standalone or together in a MVC triad, which makes the architecture feasible for the CDCE.

Is it possible, with a distributed MVC architecture, to enable a PDA to be a “remote controller” of a screen nearby or using a PC keyboard as input device?

The answer to this question is yes as proven by the prototype. The PDA can use the PC screen for zooming in on a picture and adjusting the zoom factor. Furthermore the PDA can use the PC keyboard for entering text into a PDA application.
This implemented functionality is described in:
  Scenario 1: Zooming in Pictures and Scenario 2: Using a PC as a Keyboard for a PDA.

Is it possible at runtime to reconfigure the composition of the composite device and e.g. use another display or another input device? Or in more general terms, how to dynamically reconfigure the MVC roles?

The answer to this question is yes as proven by the prototype. The developed architecture supports both dynamic exchange of MVC components e.g. exchange one view with another, and also supports dynamic switching. The PDA can use the PC screen (picture service) and then be reconfigured to use the PC keyboard (keyboard service) instead. The MVCBuilder handles the responsibility for exchange of components or services.

Is it possible to construct the distributed MVC architecture so that it enables the MVC roles to communicate in an efficient way?

The developed prototype supports a loosely coupled interface where all MVC components are independent as long as they support the structured text-based protocol. Because of the protocol being text-based, it allows services to be built with different technologies e.g. .NET or Java. Extra nodes can be added or removed in this architecture. Furthermore, as with the Service Oriented Architecture, it adds in an extra layer providing a facade to all the business objects.

Is it possible to make the distributed MVC architecture so that it satisfies the expectations of a framework?

To make it possible that the described distributed MVC architecture, is a framework it has to have the following characteristics to be a framework within the area of CDCE:

- Easy to extend within the area of offering services in a CDCE. This means that it is easy to add new views or controllers to existing services.
- Easy to add a completely new service, which means adding a new MVC triad: a model together with one or more pairs of controllers and views. As described in Using the Framework

An application programmer reusing the framework only has to implement the component and interaction model needed, and the rest of the framework can be reused. The above two requirements are in this way fulfilled.

Is it possible to give the distributed MVC architecture the ability to easily add and use new service types?

New services are added as described in the section entitled Using the Framework. New services cannot be added on the fly without recompiling the framework. On the other hand, one cannot introduce a new service without compiling the application implementing the service. If e.g. a mobile phone, uses the discovery service to discover whether a picture viewer service exist the mobile device need to have the framework installed and to know the connection layer zooming protocol to use the service. It is easy to add new MVC components within an exciting service, but adding a new service requires implementing the MVC distributed components wherever there are needed. It is relatively simple to add a new service by implementing the service as described in the section entitled Using the Framework. It can be concluded that this requirement as reasonably fulfilled.
9.1.2 Evaluation Against Fundamental Requirements

The next point in the evaluation is to address the identified fundamental requirements of a distributed MVC architecture.

**Availability** is achieved by using a combination of WLAN and LAN enabling wireless devices to connect to the stationary devices located on the LAN. This is availability in the sense of being able to be connected. The communication means TCP/IP can either guarantee to deliver a message or detect the failure.

**Discovery Services** is implemented in the Discovery namespace and implemented by a combination of a Locator and a LocationLocater. By those two components it is possible to register a service specified by type and location, and further look up the service afterwards.

**Adaptability** is implemented by the discovery service, which handles discovery of services, and by the MVCBuilder that has the responsibility of exchanging services.

**Access Transparency** is enabled by the concept of proxy/skeleton components. From the perspective of the user of the proxy component it is transparent whether the service is local or remote. The communication between proxy and skeleton is encapsulated within the proxy skeleton pair.

**Location Transparency** is handled by MVCBuilder. The MVCBuilder is the only one knowing whether a service is local or remote and this issue is hereby hidden from the user.

**Location and Context Awareness** is handled by letting the user select between location names from a drop down list when requesting use of a service. It is recommended to automate the location and context awareness in case of the use in a large location or when the number of offered services grows large.

**Quality of Service** can be achieved within this framework and security issues need to be addressed.

**Framework Qualities** are handled and are addressed in the previous section entitled Evaluation Against Problem Statement.

9.2 Future Work

This section briefly describes how the MVC architecture can be improved and further developed.

9.2.1 Discovery Services

The area of discovery services is large enough to carry its own thesis. The aspects of service registration and lookup have been covered to some extent.

The connection between MVC components in the prototype is implemented with a structured text-based protocol. It would be interesting to exchange the present protocol with UPnP and discover whether there is a performance penalty by introducing XML generators and parsers.
Using UPnP would not have made any differences to the suggested architecture due to the protocol being text-based already.

9.2.2 Context Awareness
In the initial analysis of this architecture it contained the use of a context aware service. When the described solution is used in a small closed environment, it is easy for a user to overlook and select from a short list of possible locations and services e.g. selecting to use a PC screen from a PDA.
If this solution has to be implemented in a large environment where there are a number of locations the use of context awareness would be necessary and would improve the system. A possible location aware service would also be able only to suggest that the user select from the MVC components available in the room where the user is located.

9.2.3 Security
As described in the initial analyses of the requirements concerning security in a distributed architecture is a requirement, due to handling of personal data and further an obvious concern in a wireless network. As stated in the introduction, security issues are handled superfluously and are complicated by the IEEE802.11 standard having known security issues pending.

9.3 Conclusion
This thesis describes a distributed MVC architecture, which supports making Composite Devices in order to overcome the limitation of small screen devices by using available services in computing devices from the surrounding area.

The view and controller services in the MVC architecture are abstractions of corresponding visual object, e.g. the PC screen and the PC keyboard, making it easy for a user to understand.

The key result of this work is a prototype of a distributed MVC architecture with the following abilities:
- A proof of concept framework based on the principles of a distributed MVC architecture which satisfies the identified functional requirements and qualities
- A flexible infrastructure for creating and connecting participating components in a composite device structure
- A multi-server discovery service. The server role is played by the server in question containing the model

The presented prototype is tried out via three scenarios on a Windows based XP PC and a Windows Mobile client PDA.

The results of this thesis support the arguments in [R4] by implementing a proof of concept and further support the arguments presented by several analyses of related work.

When compared to the framework Jini the present framework has the benefit of being independent of the platform due to the interaction between components being based on a textual protocol.
As described in the section entitled Related Work, a next step in this work could be to implement the UPnP protocol instead of the present protocol due to the benefits of using an open and accepted standard protocol.

The suggested architecture is also closely related to what is suggested in work at Siemens Corporate Research as described in [R5] even though their architecture is based on technology which is not recommended, such as COM.

The conclusion is that the distributed MVC Architecture represents an improvement of user’s possibilities and an improvement over standard service architecture due to the support for building composite devices and administrating the relations between the components in the MVC triad.
### 10 Definitions

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context-aware information</strong></td>
<td>Location-context. Guide visitors to locations. Active badge to determine and track people moving around a building. Considering factors as the users identify, profile, location etc.</td>
</tr>
<tr>
<td><strong>Location-aware computing</strong></td>
<td>Access to home intranet from elsewhere.</td>
</tr>
<tr>
<td><strong>Mobile computing (nomadic computing)</strong></td>
<td>Performance of computing tasks while the user is on the move.</td>
</tr>
<tr>
<td><strong>Portable devices</strong></td>
<td>Laptop computers, Personal Digital Assistants (PDA), mobile phones, pagers, video cameras, digital cameras. Devices embedded in applications such as washing machines, hi-fi systems, cars, refrigerators.</td>
</tr>
<tr>
<td><strong>Server</strong></td>
<td>Refers to a running program (a process) on a networked computer that accepts request from a program running on other computers to perform a service and responds appropriately. The requesting processes are referred to as clients. The terms ‘client’ and ‘server’ apply only to the roles played in a single request.</td>
</tr>
<tr>
<td><strong>Service</strong></td>
<td>Distinct part of a computer system that manages a collection of related resources and present their functionality to users and applications.</td>
</tr>
<tr>
<td><strong>Situated computing</strong></td>
<td>Detection of resources and social events in the surrounding environment of the mobile user. User centric approach.</td>
</tr>
<tr>
<td><strong>Spontaneous networking</strong></td>
<td>Integrates mobile and non-mobile devices in a network.</td>
</tr>
<tr>
<td><strong>Ubiquitous computing</strong></td>
<td>Computing Devices pervasive in everyday object so they are scarcely noticed. To create an environment in which computers are invisible fulfilling tasks for us in the background without being object of interest.</td>
</tr>
</tbody>
</table>
## 11 Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSMA/CD</td>
<td>Carrier Sense Multiple Access/ Collision Detection</td>
</tr>
<tr>
<td>CDCE</td>
<td>Composite Device Computing Environment</td>
</tr>
<tr>
<td>CLR</td>
<td>Common Language Runtime for .NET</td>
</tr>
<tr>
<td>EAP type</td>
<td>Extensible Authentication Protocol Enable IEEE 802.1x authentication</td>
</tr>
<tr>
<td>Extended Service Set (ESS)</td>
<td>A complete 802.11 wireless network. An ESS is composed of different base service sets and their respective access points, as well as the distribution system. An ESS appears to be a single 802 network to the upper layers of the OSI model</td>
</tr>
<tr>
<td>IETF</td>
<td>The Internet Engineering Task Force (IETF) is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. It is open to any interested individual</td>
</tr>
<tr>
<td>ISP</td>
<td>Internet Service Providers</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SS/CD</td>
<td>Small Screen/Composite Device</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>WiFi</td>
<td>WiFi Alliance is an organization the wireless industry has created. The WiFi Protected Access (WPA) standard is a subset of the 802.11i draft protocol</td>
</tr>
<tr>
<td>WEP</td>
<td>Wired Equivalent Privacy</td>
</tr>
</tbody>
</table>
12 References


[R6]. WAP Wireless Application Environment Specification, Version 1.3


[R8]. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns (1995), Addison-Wesley.


[R28]. An Introduction to some Basic Concepts in IT Security and Cryptography, Ivan Damgaard, Aarhus Universitet 2004
### 13 Links

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[L7].</td>
<td>[L8].</td>
</tr>
<tr>
<td><a href="http://www.chat.1klik.dk/shownews.php?id=37902">http://www.chat.1klik.dk/shownews.php?id=37902</a>.</td>
<td>IDC expects the sale of portable PC to be higher than the sale of stationary PCs.</td>
</tr>
<tr>
<td><a href="http://www.wireless-world-research.org/general_info/bookofvisions/BookofVisions2000.pdf">http://www.wireless-world-research.org/general_info/bookofvisions/BookofVisions2000.pdf</a></td>
<td>The objective of the wireless world research forum is to formulate visions on strategic future research directions in the wireless field, among industry and academia, and to generate, identify, and promote research areas and technical trends for mobile and wireless system technologies.</td>
</tr>
</tbody>
</table>
14 Appendices

The attachment chapter primary describes the source code and how to install and start the prototype.

14.1 Source Code Documentation

The source code is documented. The source code is written in Microsoft’s IDE Version 7.1 and the IDE directive #region is used to make the source code more readable. The source code can of course be investigated using other tools than Microsoft’s IDE.

When a public member function implement an interface the source code documentation is done together with the interface in question and this documentation is not repeated together with the implementation of the interface.

Microsoft’s recommendations for source code documentation are used. This documentation combined with the object browser in the IDE gives a good overview of the code. The following XML tags are used in the code for documentation:

Describes a member for a type:
   <summary>MyMethod is a method in the MyClass class.</summary>

Specifies overview information for a class or other type:
   <remarks>Specifies overview information for a class or other type.</remarks>

Used in the comment for a method declaration to describe the return value:
   <returns>Returns zero.</returns>

Used in the comment for a method declaration to describe one of the parameters for the method:
   <param name="Int1">Used to indicate status.</param>

Used for describing exceptions:
   <exception cref="System.Exception">Thrown when... </exception>

14.2 Used Frameworks

The framework log4net-1.2.0-beta7 is used debugging and logging of information on the server side (PC). The setting of the logging is done in a Log4net.config file. The logging will be attached to a log file. If the log file is removed a new one is automatically created.

The unit test framework Nunit2.2 is used for unit test of all essential classes.

14.3 Source Code Organization

The way the package structures are implemented in Java and the way namespaces are implemented in .NET are alike but have differences. The import package directive in Java can define the usages down to a single class. Namespace will define access to all public classes.
within a namespace. “import Server.Connection.*” in Java is similar to “namespace Server.Connection” in .NET. In Java you are forced to have the filename equal to the class name, which is not necessary in .NET. Furthermore Java forces the package names to reflect the directory structure, which is not the case in .NET. I have followed the strict Java roles and made the file name equal to the class names and made the namespaces reflect the directory structure. In my opinion the “Java way” is more user-friendly. In Java it is not possible to have more than one public class in one file. In general this is also a fine rule but in case of small interfaces it is practical to have both interface and implementation in one file, which I have done in most cases.

The organization of directories is made as equal as possible on both server and client side. The directory structure reflects the structure of the namespaces regardless of the fact that it is not required by the .NET platform.

The directory structure on the server side is as follows:

```
src/server/Connection - the connection architecture layer
/Discovery - the discovery service including locator
/Picture - the picture service MVC components and factory
/Keyboard - the keyboard service MVC components and factory
/Main - main application
/MVC - MVC structure and abstract factory
/Session - the session manager
```

The directory structure on the client side is equal to the server side.

### 14.4 How to Install the Prototype

This chapter will describe how to install and configure the prototype and further documented with screen shots of what can be done with the prototype.

Attached to this thesis is a CD containing all source code and documentation: The CD is organized in the following directory structure:

```
doc – contains the thesis
literature – contains articles
deploy – window setup files for the PDA application and server
C#/lib – contains a library with used the used frameworks: log4net and NUnit
C#/server – contains the source code for the server side
C#/PDAApplication – contains the source code for the PDA side
```

### 14.4.1 Configuration of the WLAN

The WLAN router is configured as DHCP server and will automatically assign a network address for the nodes connected.

The WLAN must be configured with:

- A Wireless Network Name (SSID), which must be the same for all computing devices participating.
- One or more network key, which must be equal for the nodes participating.
- Data encryption set to WEP (For test purpose the encryption can also be disabled).
### 14.4.2 Installation on a PC

The deploy/SetupServer/Setup will install and unpack the server application in a directory selected during installation. Double clicking on the server.exe file starts the server.

The server will during startup read a Log4net configuration file “./Log4net.config” and will create a log file “log-file.txt”. The log file will contain debug information and any exception cached. The server can work without the PDA and show the ability to exchange view components and services.

During startup a XML configuration file location of the service address is read. The configuration file contains specification of available services. If not changed the IP address of the session manager will be as defined below in XML:

```xml
<?xml version="1.0" ?>
<LocationConfigurationList xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="Server.Discovery.Configuration">
- <LocationConfiguration_list>
- <LocationConfiguration>
  <Name>PC</Name>
  <IP>169.254.74.28</IP>
  <Port>65000</Port>
</LocationConfiguration>
</LocationConfigurationList>
```

### 14.4.3 Installation on a PDA

The deploy/pda/Setup file will unpack the PDA application in a directory selected during installation. The unpacked files which must be copied to a directory in the PDA. Tapping on the PDAApplication.exe starts the PDA application.

### 14.5 How to Use the Prototype

When started the server application shows a control view as shown below.

![Session Control View](image)

**Figure 14-1 Session Control View**

Without starting a PDA the server can demonstrate exchange of MVC components and services.

If the following is selected in the session control view on the PDA:
SessionType: PictureView, ViewLocation: Simulated PDA, Controller Location: PC followed by pressing Activate the following Window will appear.

![Figure 14-2 Simulated PDA View](image)

When the selection for ViewLocation is changed to PC followed by pressing Activate the View is changed to a normal PC view.

When the service selection is changed to Keyboard followed by pressing Active all MVC components are changed from picture view service to keyboard service. It is now possible to key in text into the keyboard view. The keyboard model will not save the text.

When stating the PDAApplication the following control view will appear.

![Figure 14-3 Session Control View](image)
Select the following:
   SessionType: PictureView, ViewLocation: PC, Controller Location: PDA followed by pressing Activate, a Window will appear on the PC containing a picture.

Then select the tab PictureView on the PDA and the following picture will now appear:

![PC Picture View Control](image)

**Figure 14-4 PC Picture View Control**

Zooming in the picture on the PC can now be controlled by the PDA by using the controls for zooming in or out or adjusting the zoom factor. If Log4net is enabled on the server side the communication between the PDA and the server is printed in a log file. Adjusting the zoom factor on the PDA will be reflected by the zoom factor of the PC view.

If the keyboard service is selected and activated the text typed on the PC keyboard will be displayed on this keyboard tab.