ETH zürich

ETH Library

NADABS

A component container for the .NET (compact) framework

Master Thesis

Author(s):
Slaviša, Maslić

Publication date:
2004

Permanent link:
https://doi.org/10.3929/ethz-a-004696405

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information and Communication Systems Research Group

Masterthesis

NADABS
A Component Container for the .NET (Compact) Framework

Maslić Slaviša
smaslic@student.ethz.ch

March 2004

Supervising Professor: Prof. Gustavo Alonso
Supervising Assistant: Andreas Frei

To my parents

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Jadabs 1
1.3 Nadabs 2
1.4 Microsoft .NET Framework 2
1.4.1 Common Language Infrastructure 2
1.4.2 Common Language Runtime 3
1.4.3 Assembly 3
1.4.4 Application Domain 4
1.4.5 Microsoft .NET Compact Framework 4

2 NADABS **6**
2.1 Requirements 6
2.2 Design 7
2.2.1 The Structure of Nadabs 7
2.2.2 Components 9
2.3 Implementation 11
2.3.1 Loading and Unloading in .NET 11
2.3.2 Remoting 14
2.3.3 Service Registry 16

3 Communication Layer **18**
3.1 Messaging 18
3.1.1 JXME 18
3.1.2 Implementation 19
3.2 WLAN 20
3.2.1 UDP Component 21
3.2.2 TCP Component 23
3.3 Bluetooth 24
3.3.1 The RFCOMM Workaround 24

4 Benchmarking **26**
4.1 Benchmark Design 26
4.2 Results 28

5 Conclusion and Future Work **30**
5.1 Conclusion and Discussion 30
5.2 Future Work 31

A Build with NAnt 33
A.1 Installing NAnt 33
A.2 Building Nadabs for the Compact Framework 34
A.3 Building Nadabs for the full Framework 34
A.4 Building Components 35

B Build with Visual Studio .NET 36
B.1 Building Nadabs 36
B.2 Create Components 37

C Benchmark 38
C.1 Benchmark Environment 38
C.2 Data Ascertainment 39

Preface

Abstract

The evolution of the PDAs in the past years is comparable to the evolution of PCs. They evolved from small pocket sized organizers to multipurpose devices. Most of the available applications provide functionality that can not be changed at runtime or that can be used only in one particular environment. If we assume that users often change their environment, such applications would need to be adapted. By using a component infrastructure as the basis for adaptation we are able to add and remove new behavior as components without interfering the user.

An important characteristic of computer networks is that they may be heterogeneous. For example, a wireless ad-hoc network might include Linux laptops, Linux PDAs and PocketPC PDAs. By providing a heterogeneous environment cross-platform interaction can be supported.

In this master thesis we present an implementation of a component container for pocket devices, particulary for devices that run on Microsoft's PocketPC platform (e.g. HP iPAQ 5550) which comes with the .NET Compact Framework. The container allows components to be inserted into the device where they are started and stopped in an ad-hoc manner. This enables devices entering a new unknown environment to be adapted to the environment's common components. To accomplish this requirement a messaging component was migrated from a java infrastructure supporting two different communication protocols: TCP and UDP.

Report Structure

Chapter 1 deals with the general motivation, the related work and gives some insights into the Microsoft .NET Framework. Chapter 2 shows the requirements, the design and the implementation of the adaptive middleware platform Nadabs. Since the .NET Compact Framework is too restricted for our purpose, we have implemented two slightly different versions of Nadabs. To provide some basic communication features we introduce the C# version of the JXME messaging layer in Chapter 3. It also includes the description of the UDP and the TCP components. Since one of the proposed version of Nadabs uses Microsoft's remote method invocation infrastructure, we compare the performance of local and remote method invocation in Chapter 4. Finally, in Chapter 5, we present our conclusion and a list of potential improvements and directions for future work.

Appendix A shows how the container and the components can be built using the NAnt build tool. In Appendix B we show the build procedure using Microsoft Visual Studio .NET. Finally, Appendix C gives additional information on the benchmarks.

Acknowledgement

I want to thank my supervising assistant Andreas Frei for his support, his helpful ideas and for proof reading this report. I am grateful to my family that always supported me.

Chapter 1

Introduction

1.1 Motivation

The evolution of the PDAs in the past years is comparable to the evolution of PCs. They evolved from a small pocket sized organizer, enough to save phone numbers and appointments, to a device that can be used for a lot of purposes, e.g. telephony, video-streaming or games.

But most of the applications nowadays are installed statically, i.e. they are designed for a special purpose. Moreover, it is possible that these applications can be used only in one particular environment. If we assume that users often change their environment, such applications would need to be adapted.

Dynamic adaption of a device to the changes in the environment has become a challenge for the application running in such a device. An application should therefore adapt transparently without interfering the user. This would facilitate the development of applications for mobile devices because the developer can focus on the main aspects of the application while the location dependent or adaptive aspects are acquired when needed from the environment where the application runs. Such behavior should be added and removed without affecting other running applications.

By using a component infrastructure as the basis for adaptation we are able to add and remove new behavior as components. Such components can also be looked of as services or libraries which are registered and de-registered as required by the environment.

1.2 Jadabs

There exist implementations which address adaptive infrastructures, for example JXTA[11], Jini[21] or CCM[20]. But they are too heavy to run on a mobile device or is a specific solution for a specific platform.

This motivated the supervising assistant of this master thesis, Andreas Frei, to implement an adaptive middleware infrastructure for adaption of mobile devices that runs even on a PDA. It is a small footprint container called the Java ADhoc Application BootStrap, Jadabs. Jadabs is an component container that can accept and dynamically deploy a wide variety of components wrapped as Java archive files (.jar). With this software adaptations can take place through

components that are deployed at runtime. They can add new functionality or may even replace applications that are already activated in the device. A messaging layer and an event system are responsible for the communication and the deployment in an ad-hoc infrastructure.

1.3 Nadabs

The main drawback of Jadabs is that it does not run on a PDA with the PocketPC platform because there are not any implementations of a full Java Virtual Machine for the WindowsCE operating system. To support even PocketPC devices this master thesis' goal is to implement a component container for the .NET Compact Framework that behaves in the same way and has the same functionality as its Java counterpart. The project is accordingly named: .NET ADhoc Application BootStrap (Nadabs). Nadabs uses primary the .NET Compact Framework but it can also be built with the full .NET Framework.

The development of Nadabs as a counterpart of Jadabs is a first step towards a heterogenous environment that supports different platforms. In such an environment components are deployed depending on the platform of the target device. Combining Nadabs and Jadabs could achieve this idea.

To provide interoperability between Jadabs and Nadabs both container have to provide the same messaging layer that defines the message formats and communication interfaces. Since Jadabs uses JXME for this purpose, we ported it to .NET.

1.4 Microsoft .NET Framework

The .NET Platform is in essence a new development framework that provides an application programming interface (API) to the services and APIs of classic Windows Operating Systems, while bringing together a number of disparate technologies that emerged from Microsoft during the late 1990s (e.g. COM+, DCOM, ASP web development framework, support for new web services protocols such as SOAP, WSDL, and UDDI). It supports language independence and language integration by introducing two specifications:

- **Common Type System (CTS)** All .NET components must obey the CTS, which enables inheritance from classes, catching exceptions and take advantage of polymorphism across different languages.
- **Common Language Specification (CLS)** The CLS provides a set of basic rules that are required for language integration. Compilers that conform to the CLS create objects that can interoperate with one another.

1.4.1 Common Language Infrastructure

In August 2000, Microsoft, HP and Intel co-sponsored the submission of specifications for the Common Language Infrastructure (CLI) and C# programming language to the international standardization organization ECMA[8] to allow third-party vendors implement the CLI for a platform other than Windows.

	.NET/CLR	**Java/JVM**
Common Feature	both abstract the underlying platform	
Supported Programming Languages	All languages that are CTS and CLS conform	Java
Optimized for JIT	Yes, JITed on first method invocation	JIT and interpretation is supported
Packaging	Assembly	JAR files
Classloading	Assemblies have to be loaded into their own application domain to support unloading	`Classloader` class
Remoting	Proprietary Remoting over TCP/HTTP/SOAP	RMI and CORBA

Table 1.1: CLR vs. JVM

The CLI Working Document is available on MSDN[6], covering the CIL[1], CTS, CLS and more. Currently, there are two projects called Mono[9] and DotGNU Protable.NET[10] which are open source implementations of the .NET Framework. They are designed for different operating systems and not only for Microsoft Windows. Microsoft itself has published a shared source version of CLI that can be used to investigate the programming language concepts and to explore how the .NET technology works.

1.4.2 Common Language Runtime

The most important component of the .NET Framework is the Common Language Runtime (CLR). This is Microsoft's implementation of the CLI. It provides a managed environment in which programs are executed with security checks, object activation, garbage collection, etc. Since the CLR is similar to Java's JVM, table 1.1 summarizes the main differences between the CLR and the JVM.

Figure 1.1 shows the architecture of the .NET Framework. On top of the CLR, there are some additional layers. They are known as Framework Class Library (FCL) and provide an object-oriented API to all the functionality that the .NET platform encapsulates. The Framework Base Classes are similar to the Java Class Library and provide a variety of basic functionality.

1.4.3 Assembly

The basic unit of deployment and versioning is an assembly. It consists of a manifest, a set of one or more modules and an optional set of resources. The manifest describes the assembly in detail, e.g. its identity, the types it references, its classes, methods and so forth. These information are needed for component integration in .NET. The advantage of the metadata is that you do not need

[1]The Common Intermediate Language is generated by all .NET compilers, is a processor-neutral instruction set and is compiled to native code prior to execution.

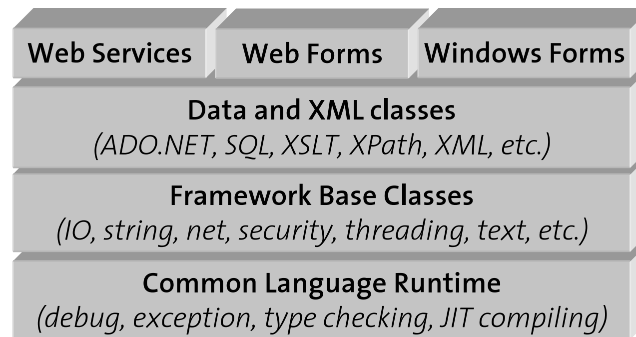

Figure 1.1: The .NET Framework Architecture

neither the source code nor any header files to access or inspect assemblies. They are completely self-describing.

Another part of an assembly is the platform independent Intermediate Language (IL) code. Microsoft calls it the Common Intermediate Language (CIL) which is a language-abstraction layer similar to the bytecode that is generated by Java compilers. Any language may be converted into IL, so .NET is language-neutral.

1.4.4 Application Domain

Running applications are modelled in the CLR as application domains. They act as managed mini-processes and thus provide an execution boundary for fault isolation, types and more. It is possible to run several application domains in a single process with the same level of isolation that would exist in separate processes (see figure 1.2). A benefit of this concept is that individual applications can be stopped without stopping the entire process. Loaded application code can be unloaded by tearing down the enclosing application domain.

However, the isolation implies that two application domains can not directly access each other's code or resources. Instead, interaction uses the .NET remote method invocation infrastructure that either provides proxies to remote objects or copies the desired object into the calling application domain. It even does not play a role whether the remote application domain is on the same machine or not.

1.4.5 Microsoft .NET Compact Framework

The .NET vision[2] doesn't cover just the desktop PCs, although it is often referred in the press as "Microsoft's platform for Web services." However, it also involves resource constrained devices. To increase the usefulness of these devices, good applications and connectivity are required. In the past, the development of this kind of software required special skills and special training.

The .NET Compact Framework is Microsoft's approach to overcome this problem. It is fully integrated into the Visual Studio .NET and enables devel-

[2] The .NET platform provides connectivity between different systems

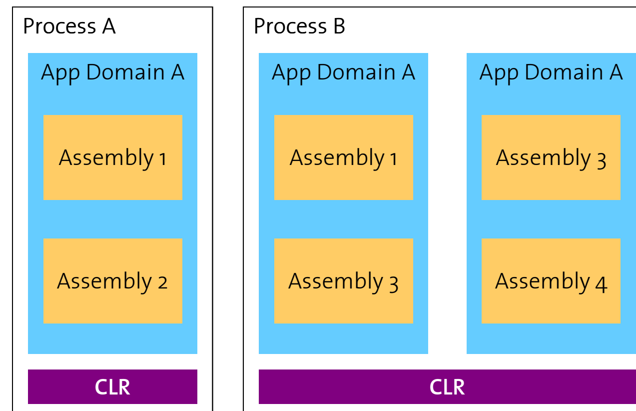

Figure 1.2: More than one application domain can be hosted in the same process. They provide the same level of isolation that would exist in separate processes.

opers that are familiar with this programming environment to develop software for PDAs using the same programming languages, paradigms and models.

Microsoft assures that .NET Compact Framework and .NET Framework have more in common than differences [2] if you use the most common features of .NET. However, the so called *Compact CLR* has a file size that is about 12% of the full CLR. To achieve this reduction of the footprint, .NET Compact Framework supports only a strict subset of the full framework libraries. The following list contains the main differences that play a role in this master thesis:

- **Memory** The .NET Compact Framework is optimized for battery-powered systems and avoids heavy use of RAM and CPU cycles.
- **Application Domain** Many Application Domains may coexist in a process, but once they are loaded, they can't be unloaded.
- **Remoting** is not supported, which means that cross-domain communication is not possible.
- **Input/Output** Since the application runs in embedded devices in which the underlying file system may differ, there are constraints and restrictions on the I/O model (e.g. it doesn't provide file change notification)

A more detailed list provides Microsoft on its MSDN homepage [3]. The impact of these restrictions will be discussed in the following chapters.

Chapter 2

NADABS

A goal of this work is an ad-hoc component container called *Nadabs* (.NET AD-hoc Application BootStrap). It conforms to the Jadabs API and is implemented in C#.

This chapter covers the requirements, the design and the implementation. Further, it deals with the differences implementing the container for the full .NET Framework and the .NET Compact Framework.

2.1 Requirements

To specify the requirements we have to narrow down the possible scenarios. We want to focus on highly mobile users that often change their environment. They use mobile, resource-constrained devices that have at least a Wireless LAN module integrated and that are most of the time enabled. The different environments provide different components that have to be trusted and safe. Given this scenario we can define the requirements for the container:

Managed Code The container has to be written in managed code. This means that things like garbage collecting or array bounds checking are handled by the CLR. This also facilitates the development of components.

Un-/Loading Components Components have to be seamlessly loaded and integrated by the container at runtime. However, the small memory of PDAs necessitates also an unloading mechanism to prevent memory overflow. Further, components should be replaceable by newer versions.

Small Footprint This requirement affects the memory space and the CPU of the device. The aim is to provide as much functionality as necessary to be CPU and memory space efficient.

Lifecycle Management The components do not have a predefined lifecycle. The user can stop them manually or they stop running when the user has left the corresponding environment.

Service Registry The loaded components are managed in a service registry where they can be looked up by other components.

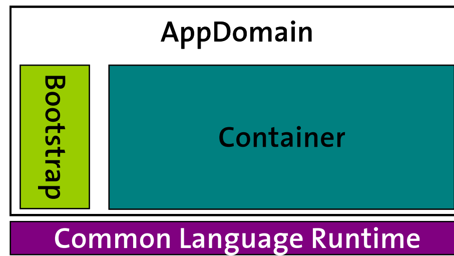

Figure 2.1: Nadabs' integration in .NET environment

Long Running Service Ideally, the container never stops running until the user stops it manually or turns off the device. This implies that the container should release unused components as soon as possible to avoid memory overflow.

Interoperability with Jadabs To provide a cross platform infrastructure Nadabs should at least use the same messaging layer as Jadabs.

The container itself should come with a bootstrap mechanism which provides a Graphical User Interface (GUI) where the user can set the working directories and that shows what components are loaded. The container should also be started and stopped using the GUI. Thus, the bootstrap should act as a client and the container as a server.

2.2 Design

Although we model our container based on the design of Jadabs, there are some crucial differences between Java and .NET that forced some changes in the container's design and the structure of the deployable components.

2.2.1 The Structure of Nadabs

Nadabs is very similar in design to Jadabs. Both aim to provide all functions needed at a small footprint. We have even named most of the classes equal to their Jadabs counterpart. This allows easy adaption of improvements to both of them. Figure 2.1 shows the integration of Nadabs in the .NET environment.

Bootstrap On the top of Nadabs, there is a small executable that allows to set some parameters that are used to initialize the container at startup. The presented bootstrap comes with a GUI but developers are free to change it for example to read the configuration from standard input or from a configuration file.

Container To guarantee that a container instance always works with the same unique objects during its lifetime we propose using the Singleton pattern for most of the included classes. The following listing introduces the classes and their purpose.

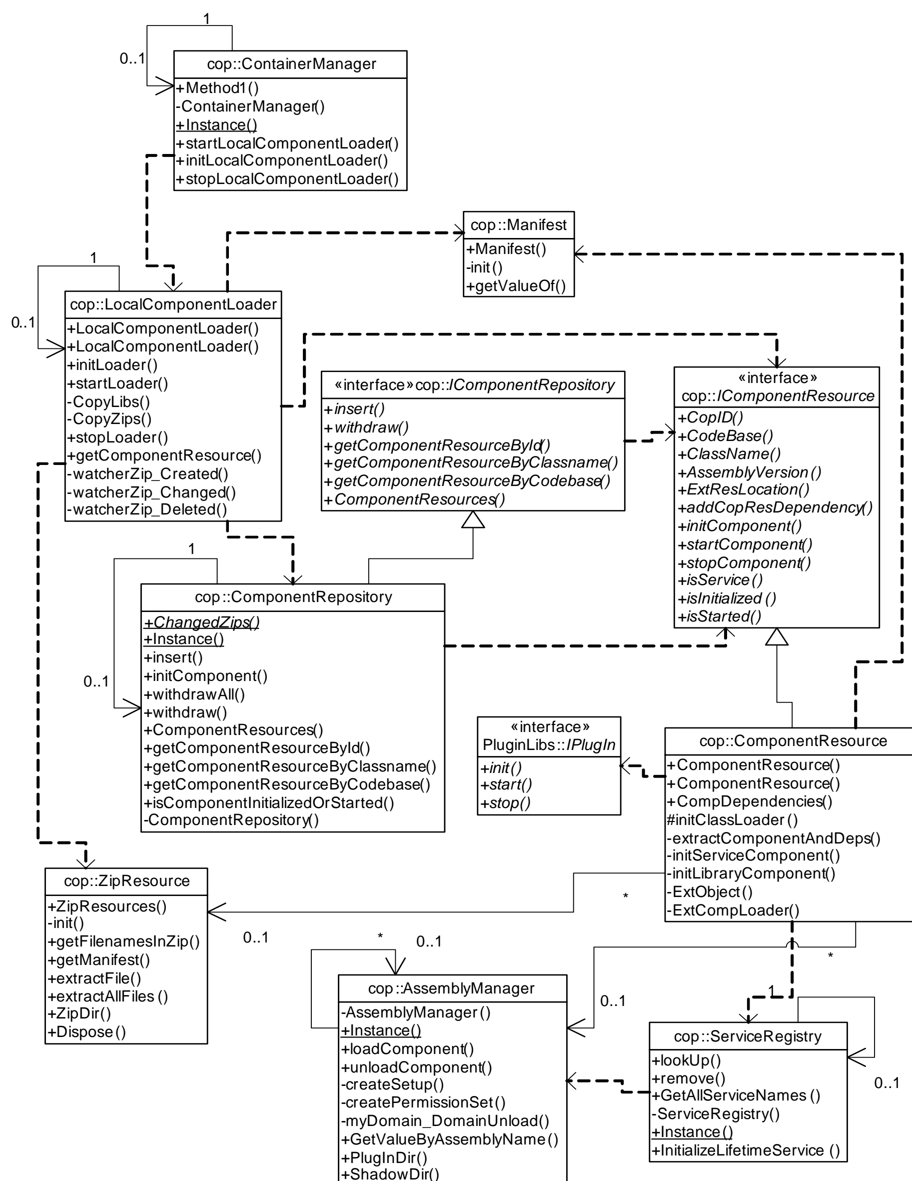

Figure 2.2: UML diagram for the container

- **ContainerManager** This class forms the entry point of the container. It exposes three methods which initialize, start and stop the *LocalComponentLoader* instance.
- **LocalComponentLoader** This class sets at its initialization the paths for the persistent zip repository and for the persistent component repository. The former one stores the new zipped components that have to be loaded by the container. Therefore, the *LocalComponentLoader* listens for any

changes in this directory and reacts depending on the incoming event (e.g. if a new zip file was created then load it). The persistent component repository is used for storing the unzipped components.

- **ComponentResource** Once the *LocalComponentLoader* detected a valid component it creates an instance of a *ComponentResource* and initializes it with the information that is necessary to load the component. Since there are two types of components (services and libraries) this class provides a possibility to initialize, start and stop services and a mechanism to load libraries. Therefore, every loaded component is represented as an instance of this class.

- **ComponentRepository** To manage the *ComponentResource* instances this class stores them into a hashtable using the component's codebase as key. Removing a component triggers the withdraw method which stops the component's running threads and removes all extracted assemblies from the disk.

- **AssemblyManager** Loading and unloading components is provided by this class. It manages the loaded components in a hashtable.

- **ServiceRegistry** This class provides a local service registry where components can lookup their dependencies. Actually, it searches the *AssemblyManager*'s hashtable for the desired component and creates an instance of the looked up type which is returned to the caller.

Jadabs uses heavily the classloading mechanism to load packages independently of each other. Nadabs had to use an appropriate loading mechanism. Since the smallest loadable unit is an assembly, we had to create classes that support dynamic assembly loading. Therefore, the *AssemblyManager* and the *ServiceRegistry* classes were created. Moreover, the concept of a service registry does not yet exist in Jadabs.

2.2.2 Components

Jadabs components are packaged into jar files. Basically, two types of components are supported: services and libraries. Services provide a class that implement the well known interface *IComponent* which also defines the entry point. Each service has a manifest included into its jar file that provides necessary information for the loading mechanism, e.g. the entry point or the version of the component. This information is added when the component is built. A library does not include information on the entry point. This allows Jadabs to separate it from the service type.

Nadabs components have a similar structure: they are not packaged into jar files, but they are also zipped. Such a zip archive contains a manifest file and at least one assembly. Nadabs supports two types of components:

Service They can be started and stopped at any time by the container. To accomplish this feature, they have to implement a globally known interface called *IPlugIn*. This interface declares the following methods:

Listing 2.1: The IPlugIn interface.

```
public interface IPlugIn : IDisposable {
    void init(IServiceRegistry svc);
    void start();
    void stop();
}
```

- **init** calls the initialization method of the assembly and passes a reference to the service registry to the component. This reference can be used for lookups.
- **start** starts the service (e.g. threads).
- **stop** stops the service, i.e. sets it ready to be unloaded (e.g. stop all associated threads and remove all event handles).
- **dispose** optionally disposes associated objects. It is inherited from the *System.IDisposable* interface.

The class that implements this interface defines the entry-point of the service.

Library This type of a component can be consumed by the services. They do not need a reference to the service registry because it is assumed that they do not rely on any other library.

Although every assembly has already a manifest built in, we suggest to add our own manifest file to every component. One reason is that it is not possible to add any additional information to this built-in manifest. Another reason is that the corresponding assembly has to be loaded into an application domain to inspect its built-in manifest. If we assume that we want to inspect hundreds of assemblies without using them every assembly remains loaded until the application is stopped, since unloading a single assembly is not supported in .NET. Obviously, it is more efficient regarding the memory to parse a small text file and to decide whether the component should be loaded compared to loading every component.

The manifest is used to describe the component, i.e. its type or the entry-point. The container can use this information to load it correctly. Since it is an ordinary text file, it can be created manually with a text-editor and includes the following entries:

- **Content-Type** This field indicates the content of the component. Default is *nadabs-cop* which indicates a Nadabs component.
- **Implementation-Version** This field is used to compare equally named components against their version. It is the same version number as defined by the *AssemblyVersion*-attribute used in the AssemblyInfo.cs file.
- **Main-Class** defines the entry-point, i.e. the class that has to be instantiated first and that implements the *IPlugIn* interface.

- **Main-Assembly** is the assembly that the container has to load before it can initialize the Main-Class.
- **Class-Path** declares the assemblies that have to be included tino the zip file, since the Main-Assembly has static references to them.

The *Main-Class* attribute is not used if the component is a library. Moreover, the container checks if the *Main-Class* is set to decide whether to treat a component as a service or as a library.

2.3 Implementation

The programming language C# has been designed in conjunction with the .NET Framework. Even the .NET Compact Framework supports C# which allows sharing of the code. However, the Compact Framework libraries are stripped-down which implies rewriting code or adding new classes to overcome these restrictions. Fortunately, C# supports conditional compilation via preprocessor directives that facilitates including or excluding code fragments.

In this section we discuss the implementation issues of the container and the impact of the Compact Framework restrictions.

2.3.1 Loading and Unloading in .NET

One of the main capabilities of Nadabs is to adapt or enrich a device with dynamically loaded components. The assemblies that are included in a component have first to be extracted into a directory and are loaded afterwards.

The framework supports dynamic loading of assemblies in different ways[1]. However, once an assembly is loaded into an application domain it can not be unloaded without to tear down the enclosing application domain. The .NET developers argue that unloading a single assembly is too expensive (e.g. the CLR would have to track all references on this assembly or it can not optimize the assembly loading anymore).

The next bigger unit that can be loaded and unloaded is the application domain. It can be easily created and unloaded as an usual object. Two or more application domains can communicate with each other using the Remoting infrastructure provided by .NET. Moreover, remote objects can be passed around like local objects, hiding almost completely the existence of process or network boundaries. Thus, our proposal is to load every component into its own application domain and unload it when the components stops running.

This idea seems to be the most reasonable one if we want to unload components. However, the .NET Compact Framework does not support unloading application domains nor remoting. It is possible to create application domains and to start executable assemblies but inter-domain communication is not provided.

These facts caused us to implement slightly different loading mechanism for pocket devices and devices supporting the full framework. Particularly the

[1] See the MSDN documentation for the Assembly.LoadFrom() and Assembly.Load() methods and Suzanne Cook's weblog on this topic: http://blogs.msdn.com/suzcook/archive/2003/05/29/57143.aspx

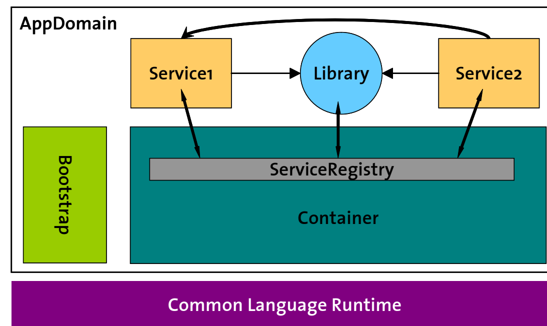

Figure 2.3: The environment of a Single Domain Container

AssemblyManager.loadComponent(string fileName, string classname) method had to be adapted. We named the resulting containers accordingly:

Single Domain Container This version of the container does not involve multiple domains and remoting. Thus, every assembly is loaded into the default application domain and remains there until the container is shut down. Since libraries and services differ in the way they have to be loaded, *AssemblyManager.loadComponent(string fileName, string classname)* method checks whether the *classname* parameter is set. This is the case if a service has to be loaded. The listing below shows the loading procedure of the two component types.

- **Libraries** are just loaded into the default application domain using the *Assembly.LoadFrom* method. Its return value is a reference to a *System.Reflection.Assembly* object which is added to the *AssemblyManager*'s internal hashtable. A *null* reference is returned to the caller.
- **Services** are also loaded with the *Assembly.LoadFrom* method whose return value is put into the hashtable. Additionally, an instance with the passed *classname* is created. This instance is then returned to the caller.

Since the assemblies can not be unloaded, we simplify the unloading mechanism. The *AssemblyManager.unloadComponent(string fileName)* method just removes the entry in the hashtable, nulls the reference to the assembly and thus deactivates the assembly. This solution works with both Frameworks. But there is an additional issue applying to both framework: as long as an assembly is loaded it is locked, i.e. it can not be deleted or overwritten if a newer version of the component is available. Therefore, we propose two solutions:

- The assemblies have to be saved in a separate directory where they can be loaded from.
- Give the assemblies unique names.

The first solution is not recommendable because it can end up in a memory overflow if the assemblies are not removed manually after the container was shut down. Further, we are using the *Assembly.LoadFrom* method to load our

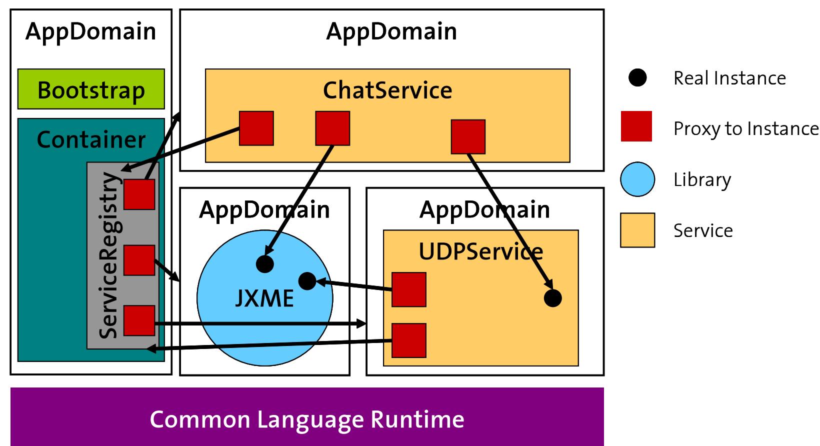

Figure 2.4: The environment of a Multi Domain Container

components which creates a separate context for every directory where assemblies are loaded from. Since assemblies in one context are invisible to other in a different context, every component has to copy shared libraries to its own directory and thus wasting memory space.

We have implemented the second solution, e.g. we add the version number to the assembly's name, and copy all of them to the persistent component directory. This eliminates redundancy and saves memory space but the user still has to remove the files manually after the container was shut down.

Multi Domain Container This container needs the full .NET Framework because the remoting infrastructure and advanced application domain functions are used. The *AssemblyManager.loadComponent* checks whether the component is a library or a service using the passed *classname* parameter. In both cases a new application domain is created where the corresponding assemblies are going to be loaded into.

In case of a library just a new entry is added to the hashtable with the assembly name as key and the reference to the new application domain as value. The assemblies are loaded when they are looked up the very first time.

Services on the other hand are not just loaded; also an instance of the requested class is created. These operations are integrated in the method *AppDomain.CreateInstanceAndUnwrap(string assemblyName, string typeName)*. It returns an object that can be casted to the requested type although this object is just a proxy to the real instance in a different application domain. Further, a new entry is added to the hashtable containing the same values as in the case of a library.

The unloading mechanism is quite simple: once all threads of a running service have being stopped, the enclosing application domain is ready to be unloaded. The method *AssemblyManager.unloadComponent(string fileName)* invokes *AppDomain.Unload(AppDomain remoteDomain)* which tears down the remote domain. Libraries are assumed having no running threads. Thus they are always ready for unloading.

Since loading and unloading of components can be done successfully, we are able to present the replacing mechanism. It is used to replace a loaded component by a newer version[2]. Basically, replacing means to stop and unload a running component and then to load and start the new version. Since unloading an application domain removes any locks on the loaded assemblies, we are able to delete these files from the persistent component repository.

However, no one can exactly predict when all the locks will be released. To handle this problem, we can enable the shadow copying property for every application domain. This prevents assemblies in the persistent component repository to be locked while they are loaded. Instead, the CLR copies the assemblies to a user defined shadowed directory and loads them from this location. These directories can be deleted programmatically when a component is unloaded or when the container is stopped.

2.3.2 Remoting

Although Nadabs is using just a small part of the .NET remoting capabilities, we want to give an overview of the most important parts. For further reading we refer to [1].

Since .NET was among other things designed for distributed computing, a remote procedure call infrastructure was integrated. Microsoft named it *.NET Remoting*. Remoting comes into play when an accessed object resides in a different application domain because of the strict isolation of domains. These domains can be hosted in the same process, in different processes that run on the same machine or on different machines. In Nadabs, we are just considering domains that are hosted in the same process.

The remoting architecture is based on five core types of objects:

- **Proxies** masquerade as remote objects and forward calls.
- **Messages** contain the necessary data to execute a remote method call.
- **Message Sinks** allow custom processing of messages during a remote invocation.
- **Formatters** are message sinks as well and will serialize a message to a transfer format like SOAP.
- **Transport Channels** are also message sinks. They transfer the serialized message to a remote process, e.g. via TCP.
- **Dispatcher** invokes the requested method on the remote object.

Figure 2.5 shows schematically the message flow through the presented objects when a remote method is invoked.

There are two ways of passing objects between application domains:

1. **Marshal By Value (MBV)** A class has to implement the *ISerializable* interface or be tagged with the *[Serializable]* attribute. A marshaled object is a copy of the original one but the two copies change their state independently.

[2]It is assumed that the new version has the same name like the old one, but the version entry in the manifest file is higher then the one in the old component's manifest.

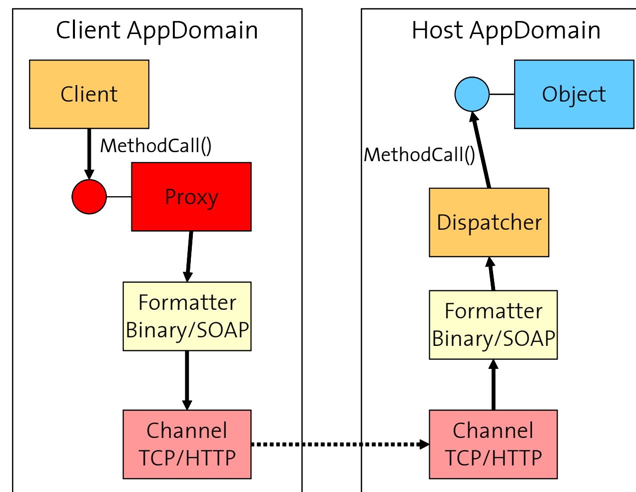

Figure 2.5: Schema of the message flow when a method is remotely invoked.

2. **Marshal By Reference (MBR)** A class must derive from *MarshalByRefObject*. All access to the object is done via proxies that forward calls to the real instance except for objects in the same application domain. Nadabs components use MBR to acces remote objects

The developer of a locally distributed application (e.g. Nadabs) does not have to be concerned about all details of remoting. Proxies, sinks and channels are automatically created by calling the *AppDomain.CreateInstanceAndUnwrap* method on the application domain where the assembly has to be loaded. This creates a new instance of the passed type using the default constructor. Since the client-server model rules the remoting-world, this kind of remote object type is called *Client Activated*, i.e. every client has its own instance in the server domain. Table 2.1 shows the characteristics of all existing types.

If *Server Activated Singleton* and the *Client Activated* object types are used one has to be aware of the lifetime of the objects which is managed with so called leases. Every lease has a default timeout of five minutes. When it expires the associated object will be garbage collected. The lifetime can be controlled by the client or by the object itself. An object-provided lease is for example the return value of the virtual *MarshalByRefObject.InitializeLifetimeService()* method. To get an indefinite lease a class has to extend the *MarshalByRefObject* class and override this method with a *return null* statement. This is what Nadabs components provide (see listing 2.2).

Basically, there are three different assemblies to create a remoting environ-

Listing 2.2: The overridden InitializeLifetimeService method.

```
public override object InitializeLifetimeService() {
    return null;
}
```

Remote Object Type	Characteristics
Client Activated	One object per activating client
Server Activated Singleton	All clients get the same object
Server Activated Single Call	A new object created and destroyed every method call

Table 2.1: Comparing remote object types

ment:

- A server assembly that contains a class that derives from *MarshalByRef-Object*.
- A client assembly that consumes the *MarshalByRefObject*.
- A shared assembly that provides interfaces or base classes.

The interface in the shared assembly has to be implemented by a class in the server assembly. The client assembly on the other side uses this interface to access the methods provided by the proxy to the server assembly. For example, consider the *AssemblyManager.loadComponent* method when a service (e.g. MyService) has to be instantiated. The assembly containing the service is the server assembly. As mentioned, every service has to implement the *IPlugIn* interface from the *PluginLibs* assembly which is the shared assembly. The container assembly is the client assembly because it consumes the created proxy using the *IPlugIn* interface to access MyService.

2.3.3 Service Registry

Until now we have not yet discussed the case when components depend on each other. The problem here is that a component does not know whether the necessary services or libraries are available at the time of initialization. Further, the libraries can be loaded in different application domains or in different contexts.

The *ServiceRegistry* class was therefore created as part of the container to provide the necessary information for the components. It was designed as a singleton to avoid inconsistency and acts as a server object that is consumed by the added components (the client objects). The clients access the service registry through a reference that is passed at their initialization with the *IPlugIn.init(IServiceRegistry svc)* method. In the multi domain environment this object is a proxy that is automatically created at client side because the *ServiceRegistry* class derives from the *MarshalByRefObject* class and crosses the boundary of its application domain.

To access this object the *IServiceRegistry* interface is used. The *ServiceRegistry* class implements this interface whose *lookUp()* method handles the client requests and works similar to the *loadComponent()* method in the *AssemblyManager* class. It accesses the hashtable of the *AssemblyManager* to check whether the desired service or library is available. If this is the case, it tries to create an instance of the wanted type using the *Assembly.CreateInstance* method for the Single Domain Container and *AppDomain.CreateInstanceAndUnwrap* for the Multi Domain Container which automatically loads the assembly into the

corresponding application domain if it is looked up the very first time. The reference to the instance is then returned to the client. If a component was not found *lookUp* returns *null*. If the component was found but it does not provide the wanted type (e.g. the *CreateInstance* method failed) an exception is thrown that has to be handled by the client.

Chapter 3

Communication Layer

Communication plays an important role in the mobile world and it is important that the devices use the same "language", i.e. they have to use the same message format and communication protocol. Nowadays, most of the portable devices support several communication interfaces, i.e. WLAN, IR or Bluetooth which use different protocols. as such it should be possible to change the communication protocols at run-time. This chapter gives therefore an insight into the JXME messaging interface and shows the implementation of a UDP and a TCP component.

3.1 Messaging

Distributed applications have to use a common communication layer to provide minimal communication abilities. This layer at least defines the basic protocols and message formats and provides interfaces to access a network infrastructure. However, it can enclose more functionality depending on the intended purpose. For example, including a security component would enable financial peer to peer services or a component transfer protocol would allow moving components across the network.

Since we want to achieve interoperability with Jadabs at least at communication level, we had to implement the same communication layer. Jadabs uses a platform called JXME which is a subset of JXTA, an open-source project initiated by Sun Microsystems. It was modified to fulfil Jadabs' requirements. Therefore we give first a short introduction to JXTA and JXME and then discuss our implementation for Nadabs.

3.1.1 JXME

The JXTA[1] project[11] was intended as a communication platform for distributed applications. The vision and the purpose of JXTA is summarized in the following quotation:

> *JXTA technology is a set of open protocols that allow any connected device on the network ranging from cell phones and wireless PDAs to*

[1] Short for juxtapose

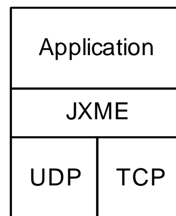

Figure 3.1: JXME layer and its underlying network access components

> *PCs and servers to communicate and collaborate in a P2P manner. JXTA peers create a virtual network where any peer can interact with other peers and resources directly even when some of the peers and resources are behind firewalls and NATs or are on different network transports.*

Since it is implemented in Java using the full J2SE, it does not run on most of the mobile devices because they include J2ME which is a subset of J2SE. In order to integrate the mobile devices into the JXTA peer to peer networks the JXTA for J2ME (also known as JXME) project [12] was proposed.

Considering the constraints imposed by mobile devices, the *proxied* JXME emerged. It has a small footprint, a minimal API and uses binary messages that are conform to JXTA's binary message format. The main drawback is that this version needs a full JXTA proxy which does all the work for the mobile device (e.g. networking, discovering etc.).

In the early stages Jadabs used the full JXTA platform as messaging layer. It even runned on a PDA with the *blackdown JRE* [13] but it was too heavy for the PDA's CPU. It took some minutes to start up the environment and to create a peer group. On the other hand the proxied JXME was no solution because Jadabs was designed for peer to peer networks without any proxies.

Thus a slightly different version of JXME was implemented for Jadabs in the diploma thesis of Daniel Käppeli [5]. Among other things the dependency of the proxy was removed, but the message format remained the same. Figure 3.1 shows the JXME layer and the underlying network access components.

3.1.2 Implementation

Interoperability between Jadabs and Nadabs at the messaging level is achieved by implementing a C# version of the Jadabs-JXME. It was designed to run as a component in both container versions. This implied to generate as a shared assembly containing an interface for JXME called *IJxme* and a server assembly *Jxme* implementing this interface.

IJxme The API of JXME is designed to be as simple as possible and therefore consists only of three interfaces:

- **IMessage** defines methods to create and manipulate JXTA messages.
- **IElement** defines methods to construct and manipulate the basic components of JXTA messages.

- **IPeerNetwork** defines methods that can be invoked on the JXTA network.

This structure is almost equal to the Jadabs version except for the *IMessageListener* interface that does not exist in IJxme because .NET uses delegates and events for notifying. The delegate defines a reference type that can be used to encapsulate a method with a specific signature. It is very similar to function pointers in C++. The event specifies the delegate that will be called upon its occurrence. Our *IPeerNetwork* defines an event named *ProcessMessage* that calls the *ProcessMessageHandler* delegate. Listeners that want to register themselves to the *ProcessMessage* event have to implement a method with the signature defined by the *ProcessMessageHandler* delegate.

We have included the JXME specific exception handler to this assembly because they have to be accessible everywhere the interfaces are accessed. Since C# does not allow to define constants in interfaces, we have implemented a new class called *MessageField* that contains all the constants that JXME defines.

Jxme This assembly provides only two classes: Message and Element[2]. Both of them are derived from the *MarshalByRefObject* class to allow remote instantiation and method calls. However, this resulted in some changes of the JXME classes.

- Public-static methods are turned into public-instance methods.
- Some *init* methods are added that can be called if an instance was created using the service registry to initialize the new instance, since these instances can only be created using the default constructor.
- The method *Message.getElementObject* was omitted.
- For the multi domain case, *InitializeLifetimeService()* was overridden to give the instances infinite lifetime.

To preserve message-compatibility to Jadabs-JXME the classes *StringBinaryReader* and *StringBinaryWriter* are ported to read and write strings of given length byte for byte from and to a stream. C# provides only writing and reading strings to streams that are prefixed with their length. Since we do not know what the exact type of the prefix is, we were forced to do these operations like Jadabs-JXME.

3.2 WLAN

The Wireless Local Area Network (WLAN) was one of the most important trend during the past few years and still is. Many companies prefer WLAN due to its flexibility and public HotSpots[3] are springing up like mushrooms. Even pocket devices are more and more equipped with a WLAN interface (e.g. HP iPAQ

[2] We have omitted the abstract PeerNetwork class which is only used to implement listener operations.

[3] A HotSpot is a public WLAN access point which anyone can use (not for free). They are situated at overcrowded area such as hotels, airports or train stations.

5550). Another advantage is that the devices can communicate in an ad-hoc manner, i.e. an infrastructure is not necessarily needed.

We discuss in this section the implementation of two communication components that use WLAN as underlying medium. Since we are dealing with JXME peer-to-peer systems, we accordingly named the corresponding classes xPeerNetwork whereas x stands for the communication protocol.

3.2.1 UDP Component

The UDP component enables a device interacting with a single peer or with a peer group using multicast UDP-sockets (see figure 3.2). It is designed as a service because it has to care about any running threads, particularly when the component is going to be shut down. Therefore the component consists of two parts: a managing one and a networking one. We discuss these parts in the following paragraphs.

Management The managing part of this component is encapsulated in the *UdpService* class. It implements the *IPlugIn* interface and extends the *MarshalByRefObject* to be accessible for the container.

This class has to assure that any objects (particularly *UDPPeerNetwork* objects) that want to access other components use the same proxy to the service registry for lookups. This is achieved by assigning the service registry reference to a static field of the affected classes. In our implementation only the *UDPPeerNetwork* class requires the lookup method. The reference is assigned in the *UdpService.start()* method using the *UDPPeerNetwork.SetServiceRegistryProxy* property.

Another purpose is to assert that the component is completely shut-down when the component is going to be unloaded. This affects particularly stopping threads and disposing objects (i.e. message queues). Therefore, the *UdpService.stop()* method notifies all running *UDPPeerNetwork* instances with a *StopThreads* event to immediately stop all running threads and stop queueing messages. The corresponding *StopAllThreadHandler* delegate is defined in the PluginLibs assembly.

Network The *UDPPeerNetwork* class is responsible for the network access. It can be instantiated by other components or client assemblies using the service registry which implies that it has to extend the *MarshalByRefObject* class if running in the multi domain environment. Further, it implements the *IPeerNetwork* interface from the IJxme assembly which allows other components to communicate with the JXTA environment.

To create a connection to a JXTA peer or a peer group, a PeerNetwork instance has to be initialized calling the *UDPPeerNetwork.create(string peern, string IpAddr, string arg)* method. The passed parameters are a peername, an IP-address and a local port to bind to. Based on the passed IP-address either multicast is enabled or not. It is checked whether it lies in the range between 223.xxx.xxx.xxx and 240.xxx.xxx.xxx. In this case the corresponding multicast group is joined and a TTL for the packets is set. Otherwise all messages are sent to the designated peer. Additionally, the current instance registers itself as a listener to the *StopThreads* event.

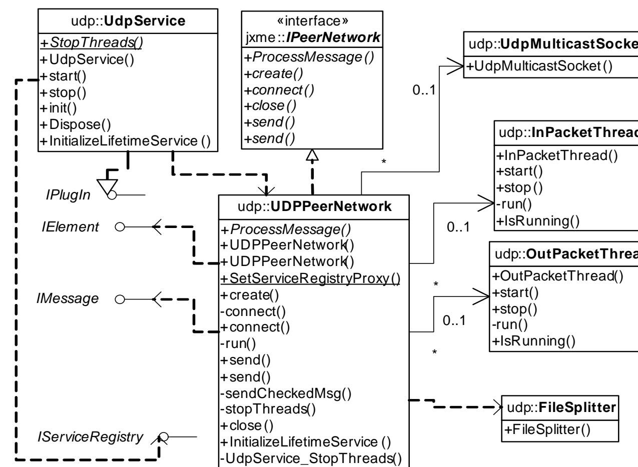

Figure 3.2: UML schema of the UDP component

Upon successful connection creation calling *UDPPeerNetwork.connect()* creates and starts three threads that operate on the previously instantiated data queues:

- **outPacketThread** dequeues a *DataPacket* instance from the outgoing queue and sends the byte array to the previously defined IP-address.
- **inPacketThread** listens on the specified port for any incoming messages. Upon arrival of a message a new *DataPacket* instance is created an enqueued into the incoming queue.
- **udpPnetThread** dequeues the received *DataPackets* from the incoming queue, creates a Message instance from the byte array and notifies the listeners with a ProcessMessage event which contains the message.

UDPPeerNetwork.send(string id, IMessage msg) allows sending messages across the network. It takes the *Message* object, serializes it into a byte array and creates a *DataPacket* instance which is enqueued into the outgoing queue. Since the size of a UDP datagram is limited to 64KB sending bigger messages will result in an error. Breaking up the message into smaller parts would solve this problem but we have not yet implemented this.

It is possible to run several *UDPPeerNetwork* instances that are listening to the same port on Microsoft Windows operating system. However, this is not the case on a WindowsCE device. Its communication layer does not allow to bind two sockets to the same IP-address and port combination.

After having finished the communication a call to *UDPPeerNetwork.close()* stops all running threads, empties all queues and closes the sockets. The handler of the *StopThreads* event also calls this method. Once all threads are stopped the instance removes itself from the list of listeners.

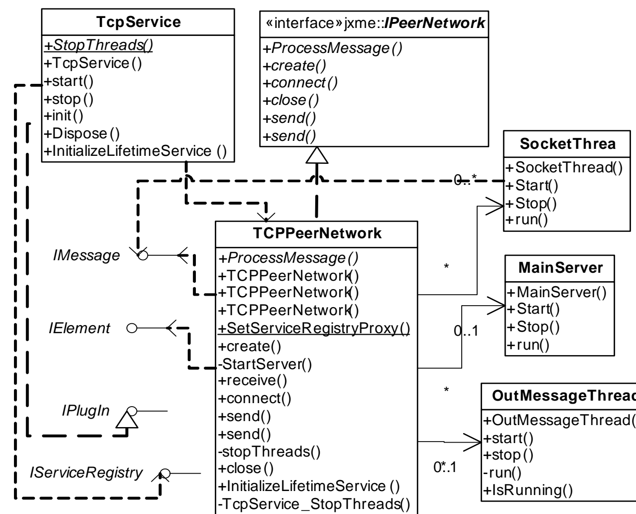

Figure 3.3: UML schema of the TCP component

3.2.2 TCP Component

The design of the TCP component is basically the same as the one of the UDP component. Instead of UDP sockets we are here dealing with TCP sockets which are useful for transmitting large amount of data. There is also a class called *TcpService* that manages the component and it has exactly the same function as *UdpService*. We want therefore only to discuss the networking part of this component to avoid redundancy.

The *TCPPeerNetwork* class is responsible for all the networking stuff. It also implements the JXME *IPeerNetwork* interface and extends the *MarshalByRef-Object* class to enable remote invocation. We have introduced new classes for the connection handling because of the connection oriented character of TCP.

Imagine the situation when multiple clients try to access an instance of *TCPPeerNetwork* at once. We have to accept these requests as soon as possible to avoid a timeout at client-side. Figure 3.3 shows schematically the design of the *TCPPeerNetwork* and its associated classes. In the following we are discussing this design.

- **Queues** Totally three queues are needed. The *socketPool* queue stores accepted connections, the *inMessageQueue* queues up incoming messages and the *outMessageQueue* stores messages that have not yet been sent.

- **MainServer Class** listens for connection requests on a predefined port. Whenever a connection is accepted a new *System.Net.Sockets.TcpClient* is created which is enqueued into the *socketPool* queue.

- **SocketThread Class** An instance of this class dequeues a *TcpClient* instance from the *socketPool*, opens a stream from which the message is read from, enqueues the message into the *inMessageQueue* and closes the stream and the *TcpClient* instance. We assume that a client sends only one message per connection.

- **TCPPeerNetwork.receive() Thread** dequeues the received messages and notifies the registered listeners.
- **OutMessageThread Class** dequeues a *DataPacket* from the *outMessageQueue*, opens a socket to the host using *System.Net.Sockets.TcpClient*, writes the byte array to the corresponding stream and closes the stream and the socket.

If a peer wants to send a message over TCP it has to know the exact IP-address and the port of his communication partner because it is obvious that using a connection oriented protocol like TCP does not allow multicasts. The IP-address, the port and the message have to be passed to the *TCPPeerNetwork.send(string address, int port, IMessage msg)* which creates a *DataPacket* instance and puts it into the *outMessageQueue*.

3.3 Bluetooth

Another emerging wireless technology is the Bluetooth standard [15]. It is supported by the Special Interest Group (SIG) which includes among others Agere, Ericsson, IBM, Intel, Microsoft, Motorola, Nokia and Toshiba. Its specification is available to everybody on the webpage where we found the following general description:

> *Bluetooth wireless technology is a short-range communications system intended to replace the cable(s) connecting portable and/or fixed electronic devices. Key features are robustness, low power, and low cost. Many features of the core specification are optional, allowing product differentiation. The Bluetooth core system consists of an Radio Frequency transceiver, baseband, and protocol stack. The system offers services that enable the connection of devices and the exchange of a variety of classes of data between these devices.*

To develop Bluetooth applications, a Bluetooth stack and the corresponding for the corresponding programming language is necessary. For Java there exists a standardized Bluetooth API which was designed by the Java Community Process[4] (JCP). Daniel Käppeli has implemented JXTA over Bluetooth[5] for Jadabs using this API.

Unfortunately, .NET has no Bluetooth API and there are no open source communities that are working on it. However, the Bluetooth-enabled PDAs come with a proprietary Bluetooth stack and there are no freely available APIs. We have only found some commercial software development kits but the prices are quite high (900$ - 2000$). Since it was not the goal of this thesis to implement an own Bluetooth stack for PocketPC or to investigate the existing stack, we have looked for a workaround.

3.3.1 The RFCOMM Workaround

The idea of this approach is to communicate over a Bluetooth serial connection with other peers. Therfore, the radio frequency COM[5] (RFCOMM) layer is

[4] http://jcp.org

[5] The well-known serial COM port

needed which is included in the Bluetooth specification and most of the proprietary Bluetooth stacks have implemented it.

However, .NET does not support serial connections, i.e. it has to be implemented using unmanaged components provided by the underlying operating system. Luckily, there is an API for serial connections freely available from the Open NETCF homepage[22] which simplifies the implementation with C#.

The idea is to implement a component that connects to a virtual serial port which is provided by the operating system and to send messages over this connection. The characteristics of this solution is listed below.

- Since COM was used originally for wired connections, only point to point connections are possible (like TCP).
- Discovery of devices is not possible.
- The Bluetooth devices have to be manually paired before connecting is possible.

Facing these facts we have decided not to implement this approach because it is even more restrictive than TCP where more than one connection can be handled concurrently.

Chapter 4

Benchmarking

During the implementation of the container we have found different solutions due to the restrictions of the .NET Compact Framework namely the Single Domain Container (SDC) and the Multi Domain Container (MDC) (see section 2.3.1). Since both versions are running on a PC and the MDC is using remoting to access components, we are able to compare the method invocation on different components in a MDC and in a SDC version. This information is useful for calculating the overhead that the remoting infrastructure adds.

4.1 Benchmark Design

The benchmark is designed as a set of components. Thus they can be loaded by the container like every other Nadabs component. This facilitates the development because we do not have to include an application domain related code into the benchmark classes. The current build of the container specifies whether SDC or MDC is used. This benchmark needs three assemblies (see figure 4.2):

- **IBenchmark** contains the *IBenchmark* interface which defines a set of methods that take different parameter types and have different return types (see listing C.1).
- **BaseBenchmark** contains the *BaseBenchmark* class which implements the *IBenchmark* interface and extends the *MarshalByRefObject*.

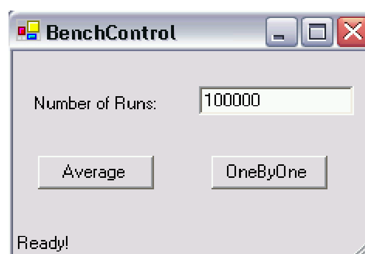

Figure 4.1: GUI for the Benchmark component

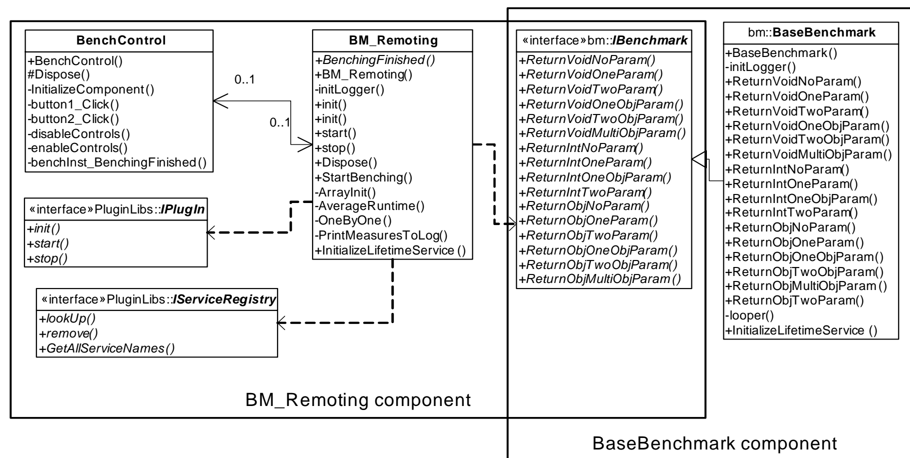

Figure 4.2: UML schema of the two benchmark components

- **BM_Remoting** contains the *BM_Remoting* class which invokes the *IBenchmark* methods and logs the measured times into a file. Additionally, the *BenchControl* class provides a simple GUI where one can set the number of runs and choose the measurement method (see figure 4.1).

The measurements are started when the *BM_Remoting.StartBenching(bool isAverage, int NofRuns)* method is called. Depending on the *isAverage* parameter one of two measurement method is chosen:

- **AverageRuntime()** This method takes the actual time, invokes a benchmark method *NofRuns*-times and takes again the actual time. The difference of the two measurements can be divided to get the time needed for one invocation.

- **OneByOne()** This method takes the actual time, runs a benchmark method once and takes again the actual time. The difference is thus the time one single invocation lasts. This is done for every benchmark method *NofRuns*-times.

Since the compiler is trying to optimize our benchmark methods, we had to add statements that can not be removed by the compiler. Therefore, the same log call from the *log4net*[19] package is included in every benchmark method which assures a measurable constant load. Since the CLR compiles the code just in time (JIT), we start the *BM_Remoting.OneByOne()* method with 1000 runs to assure that all methods are compiled when the real measurements start.

The benchmark is split up into two components: the *BaseBenchmark* library component which includes the *BaseBenchmark* and the *IBaseBenchmark* assemblies and the *BM_Remoting* service component that includes the *BM_Remoting* assembly. The former has to be loaded first.

Method Type	Slow-Down factor
Group1	26–32
Group2	244–268
Group3	146–177

Table 4.1: Slow-down factors for different method types in MDC compared to the SDC

4.2 Results

We have measured the invocation time for 16 different methods. They can be split up into three groups:

Group1 These methods take basic types, e.g. an *int* type, or void as parameters and as return values.

Group2 These methods use only objects for parameters and return values.

Group3 These methods combine basic types and objects.

Only the method *AverageRuntime()* was used for the measurements. The other one does not return meaningful results, since the resolution of the .NET timer is lower than the duration of a method invocation (every measurement using *OneByOne()* returned 0ns). The test was repeated 20 times with the same number of runs: 100'000 runs for the MDC and 10'000'000 runs for the SDC.

Method invocation using the SDC is fast and almost constant, i.e. the number and kind of the parameters do not play a role. On the other hand, remoting is found to significantly slow down method invocation. Moreover, this slow-down factor depends on the parameters and return values of the method. If only basic types are used the invocation is about 30 times slower compared to the invocation in SDC. Combining objects and basic types results in a 150 times slower invocation. For passing and returning objects the slow-down factor is about 250. Table 4.1 summarizes these results.

Not only the parameter type is critical but also the number of passed parameters. Figure 4.3 shows the average execution time for all methods. It grows linearly the more parameters are passed. This is obvious, since remoting uses channels for communication and parameters have to be serialized onto these channels one by one.

Another remarkable point is that the MDC measurements have a nearly constant standard deviation compared to the SDC version (see figure 4.4). We assume that this is due to the machine's timer and its resolution.

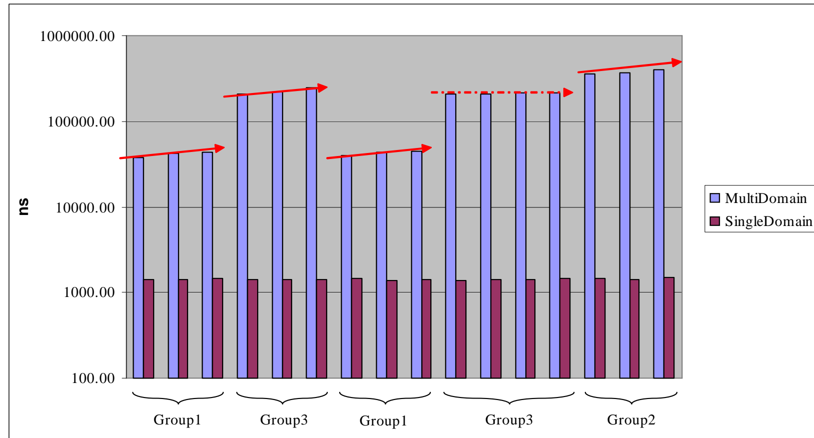

Figure 4.3: Average invocation time

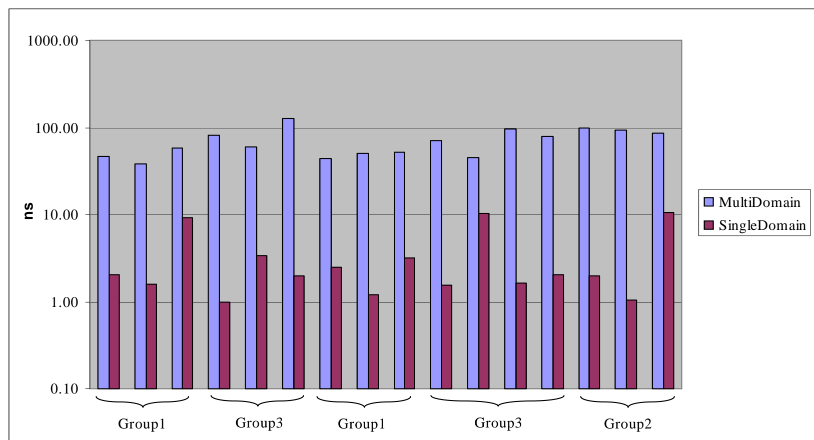

Figure 4.4: Standard deviation of the invocation times

Chapter 5

Conclusion and Future Work

The primary goal of this master thesis was to implement a lightweight component container within the .NET (Compact) Framework. Additionally, a JXME messaging and communication layer had to be adapted to allow interoperability with Jadabs. These goals have been achieved to a large degree.

In this final chapter we present our conclusion and discuss the differences implementing the container within the two frameworks. This is followed by a discussion of open tasks and future work.

5.1 Conclusion and Discussion

In this project we have successfully shown how a lightweight component container can be implemented for the full .NET Framework and for the .NET Compact Framework. It is completely component-based and service oriented and the first of his kind written in C#.

The main feature is that any new components can be pushed to the device over the wired or wireless network which are then installed (or they update existing ones) without any third party intervention. The system has not to be shutdown or rebooted. It even guarantees that a faulty component does not affect the rest of the system.

The full .NET framework provides all necessary concepts and libraries that are required for a component container. On the other hand, using the restricted Compact Framework (CF) does not support all required concepts. It is quite obvious that the current remoting infrastructure would extremely slow down a resource constrained device if we look at the benchmarks. Remote invocation is about 30 times (if basic types are used) to 250 times (if the return types and parameters are objects) slower than a local method invocation. However, if future developments of the .NET Framework improve remoting then it would be possible to implement the desired behavior also on pocket devices.

To visualize the differences regarding the requirements we have summarized them in table 5.1. The presented footprint includes only the container and the PluginLibs assemblies. If the bootstrap assembly is taken into count the footprint would be 48KB for the .NET Compact Framework and 60KB for the

Requirement	.NET	.NET CF
Managed Code	√	√
Small Footprint	40 KB	40 KB
Service Registry	√	√
Loading Components	√	√
Unloading Components	√	-
Lifecycle Management	√	-
Long Running Service	√	-
Remoting	√	-
JXME	√	√

Table 5.1: Comparing of the container implementations in .NET and .NET CF

other one. Jadabs's file size is about 28KB but notice that it is a compressed archive.

All shown differences are basically due to the lack of a remoting infrastructure in the Compact Framework. Thus we do not want to discuss them in detail. The interoperability with Jadabs is also guaranteed by the JXME layer implementation which is also the first porting to .NET.

5.2 Future Work

Since this implementation is one of the first approach to a dynamic component container in .NET (Compact) Framework, there are still a lot of open tasks. We want to cover some of them in this section.

Event System for Nadabs

Since we are considering distributed and mobile systems, a suitable paradigm is needed that allows devices or even components to interact with each other. An event-based system as proposed in [16] seems to be a solution to this issue. Jadabs already has such a system that is based on JXME. Thus it is obvious that porting this system to .NET would increase the interoperability with Jadabs or even enable cross-platform interactions.

Bluetooth Support

Bluetooth is slowly finding its way into most parts of consumer electronics. It is designed for short-range ad-hoc communication which fits very well into this context. Thus, it may be adequate to launch an open-source project which concerns the development of C# API for Bluetooth. This would provide an additional way of interaction between involved devices.

Security

Until now, we have never discussed security aspects of the container. But we think that this topic should be covered as soon as possible because in the proposed implementation anyone could send components to a device and there are

no restrictions for these components (e.g. all files on a device can be accessed and deleted). Even the communication is completely insecure.

The full .NET Framework provides some security features which have already been integrated into the actual implementation. For example when a new application domain is to be created then we first define a security policy which contains a permission set. This set can specify file access or code execution permissions. Thus we are able to restrict the operations of the components.

All this does not apply to the Compact Framework because it does not include any security features. Thus, we propose to add a security layer that checks permissions before any components are pushed to the device. Moreover, this layer could be used to encrypt the communication in a peer group.

Component Evolution

To replace a running component all dependent component have to be stopped and restarted again. Since there are services that should not be shut-down, we propose to migrate the services' state instead which means that the running services use the replaced component without being stopped.

Our service registry and the concept of proxies may be a first step in this direction. We could adapt the proxies to be notified whenever the referenced component is being replaced. The proxy could perform a service registry lookup to get a reference to the new component. In .NET we could override the *System.Runtime.Remoting.Proxies.RealProxy* abstract class to add our evolution extension. Again, this does not apply to the Compact Framework.

Appendix A

Build with NAnt

NAnt is a free .NET build tool. In theory, it is kind of like Make, but without Make's wrinkles. Many different runtime frameworks are supported, for example Microsoft .NET Framework 1.1 or Mono. In practice it is a lot like Ant[18].

We have generated a NAnt build file to build the different containers and components from the command line. Since the Microsoft .NET SDK can be downloaded for free from the Microsoft homepage, the expensive Microsoft Visual Studio is not necessary for development. The SDK includes all tools needed for compiling C# code.

When NAnt is started without any parameters it looks for a `build` file type in the current directory, for example `nadabs.build`. If it was found the default task is executed. We refer to the NAnt documentation for further information.

A.1 Installing NAnt

- Download NAnt from sourceforge[17]
- Unzip the archive to a folder
- Add `nant-folder\bin` to the `PATH` environment variable.

Adapt Nadabs Buildfile

- Get the `nadabs` module from CVS
- Open the `nadabs.build` file with a text editor
- Change the values of the following five properties:
  - **cabwiz.dir** The path to the Cabinet Wizard utility which generates cab files for deployment
  - **cf.basedir** The path to the .NET Compact Framework base libraries
  - **f.basedir** The path to the .NET Framework base libraries
  - **cab.dest.path** The path where the generated cab is put
  - **cab.log.path** The filename with its full path of the Cabinet Wizard log-file

A.2 Building Nadabs for the Compact Framework

There are two ways to build the container. The first one creates automatically a deployable cab file the other one just compiles the code and generates the corresponding assemblies.

Build and Deploy

- Open a command prompt and change to the Nadabs root directory
- Execute `nant.exe` without any parameters
- Copy the `.\bin\bootstrap_PPC.ARMV4.CAB` file to the PocketPC device using ActiveSync (only for devices with an ARM processor)
- Open the Explorer on the device and change to the directory where the cab file is copied to
- Click on the cab file which installs Nadabs into the `Program Files` directory on the device

Just Build

- Open a command prompt and change to the Nadabs root directory
- Execute `nant.exe CFContainer`
- Copy the files from the `.\bin` directory the PocketPC device using ActiveSync

A.3 Building Nadabs for the full Framework

Since two versions of the container are implemented, two NAnt targets are defined to build them.

Build Multi Domain Container

- Open a command prompt and change to the Nadabs directory
- Execute `nant.exe FContainer`

Build Single Domain Container

- Open a command prompt and change to the Nadabs directory
- Execute `nant.exe SingleFContainer`

A.4 Building Components

We have defined NAnt-targets which build components and zip them automatically. The following list gives an overview of the available targets and the corresponding commands:

- **Jxme:** `nant CreateJxme`
- **Udp:** `nant CreateUdp`
- **Tcp:** `nant CreateTcp`
- **EventService:** `nant CreateEvent`
- **ChatService:** `nant CreateChat`
- **Benchmark:** `nant CreateBm`

All component targets have the same structure which facilitates adding target for new components. Just copy an existing target and adapt filenames, dependencies. To build these components for the full framework add the option `-D:buildNet=true` to the nant commands, for example:

```
nant -D:buildNet=true CreateJxme
```

Appendix B

Build with Visual Studio .NET

To develop .NET applications we recommend using Microsoft's Visual Studio .NET. It facilitates the creation of different projects types which are integrated into a so called solution. The projects are showed in a tree view which gives a good overview. Figure B.1 shows the structure of the Nadabs solution used in this thesis.

Another benefit is that programming a PocketPC application is as simple as programming a desktop application. Installing *Smart Device Extensions (SDE) for Microsoft Visual Studio .NET* adds all tools to Visual Studio that are needed to write applications for the .NET Compact Framework. It comes with a very useful pocket device emulator that behaves as a real PocketPC device would. It is even possible to debug these applications on the emulator or on a real device (needs ActiveSync) and deployment (cab files or direct install on the real device) needs only two clicks.

B.1 Building Nadabs

To build Nadabs with Visual Studio do the follwing:

- Get the `nadabs` module from CVS
- Change to the Nadabs root directory and double click on the `nadabs.sln` file. This opens Visual Studio and imports all projects into the Nadabs solution.
- Right-click on the solution symbol and select *Rebuild Solution*

The compiled assemblies are stored into the directory which is generated in the same directory where the Nadabs root directory is. The `bootstrap.exe` which starts Nadabs can be found in the `binaries\bootstrap` respectively in the `binaries\bootstrapF` directory. The latter one targets the full .NET Framework.

Create deployable Cab-File

- Right-click on the **bootstrap** project
- Select *Build Cab File*

This creates several cab files for different pocket device processors in the `nadabs\ch\ethz\iks\bootstrap\cab\Debug` directory. This file can be copied and installed on the target device.

If the target device is connected to the PC using ActiveSync, the application can be deployed by selecting the *Deploy* option instead the *Build Cab File*.

B.2 Create Components

Components have to be created manually according the following steps:

- Right-click on a component-project, e.g. **Jxme**
- Select *Rebuild*
- Edit the manifest file if necessary
- Copy the manifest file from the project's source directory to the corresponding binary directory (e.g. from `nadabs\ch\ethz\iks\jxme\Jxme` to `binaries\Jxme`
- Create a zip archive which has the same name as the component's main assembly (e.g. **Jxme.zip**)
- Add the manifest file and all listed assemblies in the manifest's *Class-Path* attribute to the zip archive

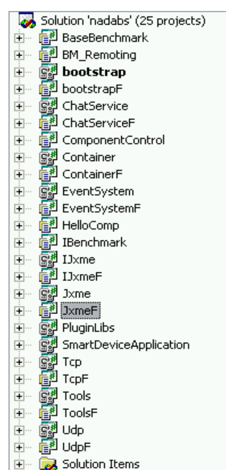

Figure B.1: Nadabs' Solution structure

Appendix C

Benchmark

This chapter gives some additional information on the benchmark environment and contains additional figures.

C.1 Benchmark Environment

The benchmark was run on a DELL Latitude C640 laptop with an Intel Pentium 4-M CPU at 2.0GHz. The physical memory of this device is 256 MB DDR-SDRAM. We use the .NET Framework 1.1 and the Microsoft Windows XP Professional operating system. The screensaver and the power management are disabled and no other application than the container is running.

We compiled both versions of the container and the two components with the command line compiler (csc.exe) using the NAnt[17] build tool. This ensures that no debugging information is included into the binaries and that all sources are built against the right framework.

Listing C.1 shows the `IBenchmark` interface which are implemented by the `BaseBenchmark` class. They are all invoked methods during the measurements. The name of the methods is composed of the return value and the parameter types. For example, `ReturnIntTwoParam` means that an integer type is returned and two integer are passed to the method. On the other hand `ReturnVoidMultiObjParam` is a method that does not return anything and that that takes several objects as parameter. In our case five objects are passed to this method.

Listing C.1: The IBenchmark methods.

```
public interface IBenchmark{

    void ReturnVoidNoParam();
    void ReturnVoidOneParam(int val);
    void ReturnVoidTwoParam(int val1, int val2);
    void ReturnVoidOneObjParam(object val);
    void ReturnVoidTwoObjParam(object val1, object
       val2);
    void ReturnVoidMultiObjParam(object val1, object
       val2,object val3, object val4,object val5);

    int ReturnIntNoParam();
    int ReturnIntOneParam(int val);
    int ReturnIntOneObjParam(object val);
    int ReturnIntTwoParam(int val1, int val2);

    object ReturnObjNoParam();
    object ReturnObjOneParam(int val);
    object ReturnObjTwoParam(int val1, int val2);
    object ReturnObjOneObjParam(object val);
    object ReturnObjTwoObjParam(object val1, object
       val2);
    object ReturnObjMultiObjParam(object val1, object
       val2,object val3, object val4,object val5);

}
```

C.2 Data Ascertainment

This section includes some figures which are not shown in chapter 4.

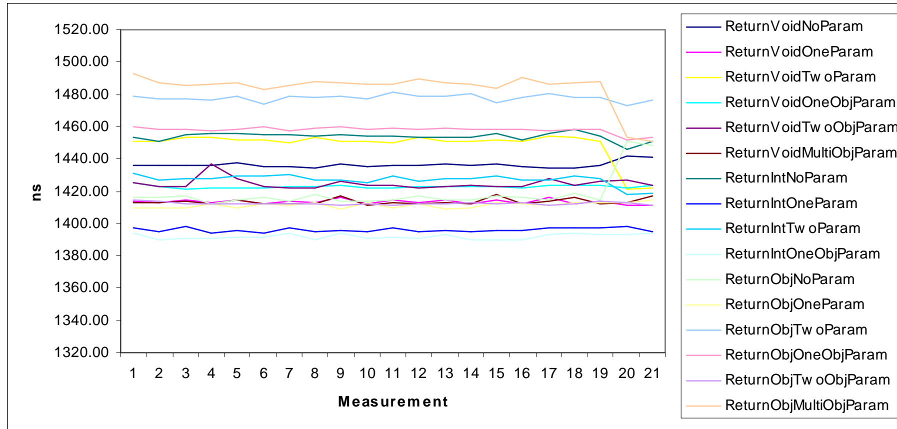

Figure C.1: Single Domain Container

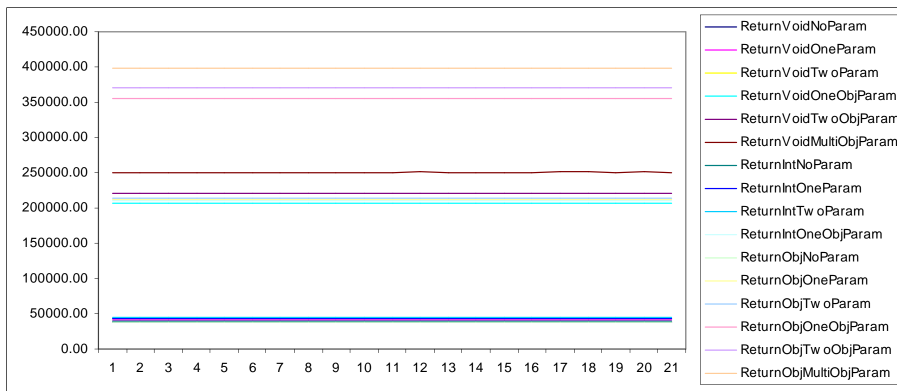

Figure C.2: Multi Domain Container

Bibliography

[1] Ingo Rammer. Advanced .NET Remoting, apress 2002

[2] Similarities with the .NET Famework
http://msdn.microsoft.com/library/default.asp?url=
/library/en-us/dv_evtuv/html/etconsimilaritieswithnetframework.asp

[3] Differences with the .NET Framework
http://msdn.microsoft.com/library/default.asp?url=
/library/en-us/dv_evtuv/html/etcondifferenceswithnetframework.asp

[4] Andy Wigley, Stephen Wheelwright. Microsoft .NET Compact Framework (Core Reference), Microsoft Press 2003

[5] D. Käppeli. JXTA over Bluetooth, Diploma Thesis, ETH Zurich, August 2003

[6] ECMA and ISO/IEC C# and Common Language Infrastructure Standards, http://msdn.microsoft.com/net/ecma

[7] Shared Source Common Language Infrastructure 1.0 Release, http://msdn.microsoft.com/net/sscli

[8] ECMA Homepage, http://www.ecma-international.org/

[9] Project Mono web site,
http://www.gotmono.com and http://www.go-mono.com

[10] Project DotGNU Portable.NET web site,
http://www.gnu.org/projects/dotgnu/pnet.html

[11] Project JXTA web site, http://www.jxta.org/

[12] Project JXME web site, http://jxme.jxta.org/

[13] Project blackdown web site, http://www.blackdown.org/

[14] Project #ziplib web site,
http://www.icsharpcode.net/OpenSource/SharpZipLib/Default.aspx

[15] The official Bluetooth membership web site, https://www.bluetooth.org/

[16] A. Frei, A. Popovici and G. Alonso. Event based systems as adaptive middleware platforms. In *Workshop of the 17th European Conference for Object-Oriented Programming*, July 2003

[17] Project NAnt web site, http://nant.sourceforge.net/

[18] Project Ant web site, http://ant.apache.org/

[19] Project log4net web site, http://logging.apache.org/log4net/

[20] Object Management Group. CORBA Component Model, v3.0. Online Documentation, 2003, http://www.omg.org/technology/documents/formal/components.htm

[21] Sun Microsystems. Jini Network Technology. Online Documentation, http://wwws.sun.com/software/jini/

[22] OpenNETCF community web site, http://www.opennetcf.org/

List of Tables

1.1 CLR vs. JVM 3

2.1 Comparing remote object types 16

4.1 Slowdown factors 28

5.1 Comparing the containers 31

List of Figures

1.1 The .NET Framework Architecture 4
1.2 Isolation of application domains 5

2.1 Nadabs' integration in .NET environment 7
2.2 UML diagram for the container 8
2.3 Single Domain Container 12
2.4 Multi Domain Container 13
2.5 Remoting schema 15

3.1 JXME layer 19
3.2 UML schema of the UDP component 22
3.3 UML schema of the TCP component 23

4.1 GUI for the Benchmark component 26
4.2 Benchmark components 27
4.3 Average invocation time 29
4.4 Standard deviation of the invocation times 29

B.1 Nadabs' Solution structure 37

C.1 Single Domain Container 40
C.2 Multi Domain Container 40

Listings

2.1 IPlugIn interface 10
2.2 Overridden lifetime metho 15
C.1 Benchmark methods 39