University of Ljubljana
Faculty of Computer and Information Science

Igor Habjan

Distributed Dependency Injection

diploma thesis

at the university study program

Mentor: Dr. Andrej Brodnik

Ljubljana, 2016

This page intentionally left blank.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Igor Habjan

Porazdeljeno vrivanje odvisnosti

diplomsko delo
na univerzitetnem študiju

Mentor: dr. Andrej Brodnik

Ljubljana, 2016

This page intentionally left blank.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

This page intentionally left blank.

Acknowledgments

It is a wonder this thesis has come to fruition; it is due to the encouragement of many people.

First of all, I would like to thank Dr. Andrej Brodnik for his time, valuable input and excellent guidance during this process.

I am grateful to all the people involved in the DeDiSys project[1]. In particular I would like to thank Klemen Žagar for being my guide during our collaboration[2], Dr. Karl Michael Goeschka for coordinating the project, Johannes Osrael for his work on the replication techniques and Lorenz Froihofer on the architecture.

I would like to thank my colleagues at work[3] for their help and friendship.

Finally, I take this opportunity to express my gratitude to my family and friends for their endless encouragement and moral support.

To my parents for all their love and care that was demonstrated somewhat oddly by sending me away to become a young student. This was many years ago and I am becoming to understand only now as a father myself. We were always welcomed and brought together by my grandparents and relatives.

My life partner and our baby boy deserve a particular note of thanks for their forbearance. I have only love and care to return their kindness.
This is for all the promises I have made!

I meant what I said and I said what I meant. An elephant's faithful one-hundred percent!

—Dr. Seuss, Horton Hatches the Egg

[1]This work has been partially funded by the European Community under the FP6 IST project DeDiSys (Dependable Distributed Systems, contract 4152, `http://www.dedisys.org`)
[2]Previous affiliation with Cosylab d.d., `http://www.cosylab.com/`
[3]Current affiliation with Evolve d.o.o., `http://www.evolve.si/`

This page intentionally left blank.

To my son

and to all who came before me
and to all who may follow.

Programming a computer is still one of the most difficult tasks ever undertaken by humans; becoming proficient in programming requires talent, creativity, intelligence, logic, the ability to build and use abstractions, and experience — even when the best of tools are available.

—Timothy A. Budd

This page intentionally left blank.

Contents

This page intentionally left blank.

Acronyms

.NET	*.NET Framework*
AOP	*Aspect-Oriented Programming*
API	*Application Programming Interface*
CAP	*Consistency, Availability, Partition tolerance*
COM	*Component Object Model*
CORBA	*Common Object Request Broker Architecture*
DCOM	*Distributed Component Object Model*
DeDiSys	*Dependable Distributed System*
DI	*Dependency Injection*
EJB	*Enterprise JavaBeans*
GUID	*Globally Unique Identifier*
GMS	*Group Membership Service*
ID	*Identifier*
IoC	*Inversion of Control*
IP	*Internet Protocol*
PC	*Personal Computer*

RMI *Remote Method Invocation*

RPC *Remote Procedure Call*

SQL *Structured Query Language*

TCP *Transmission Control Protocol*

URI *Uniform Resource Identifier*

URL *Uniform Resource Location*

XML *Extensible Markup Language*

Abstract

Title: Distributed Dependency Injection

Author: Igor Habjan

Applications nowadays are built of objects, which collaborate in order to provide their functionality, are interconnected by default and are by no means limited to a single domain of an application, a process or a computer. In this thesis a concept of dependency injection, which enables an object to explicitly declare and require its dependencies to be provided, is distributed across domain boundaries. In support of a distributed dependency injection we provide an external tool (a container) for assembling objects and resolving their dependencies (collaborators) from across domains. We provide a model in which a group of distributed dependency injection containers connect on behalf of the applications. We provide them with a middleware solution for seamless and fault-tolerant sharing of objects/dependencies between interconnected domains. A collection of support services (i.e. the distributed object replication middleware) transparently manages replication of objects created by the dependency injection principles across multiple computers. A fresh failover is ensured by invariable consistency upon invocations. This is temporarily relaxed during degraded situations (e.g. network failures) in order to achieve availability within the isolated groups. Recovery from failures is ensured by logging and check-pointing the state of the system on a regular basis; conflicting modifications are resolved. Our proof-of-concept implementation is an add-on to .NET Remoting middleware and an extension to the Unity Container.

Keywords: distributed systems, middleware, object-oriented programming, dependency injection, replication, .NET Framework.

This page intentionally left blank.

Povzetek

Naslov: Porazdeljeno vrivanje odvisnosti

Avtor: Igor Habjan

Aplikacije so dandanes izvedene na osnovi objektov, ki pri zagotavljanju funkcionalnosti vzajemno sodelujejo. Običajno so omrežene in presegajo omejitve domene posamezne aplikacije, procesa ali računalnika. V nalogi preučimo princip vrivanja odvisnosti, po katerem objekt zgolj jasno izrazi svoje odvisnosti do drugih objektov (sodelavcev) in pričakuje, da mu bodo ti priskrbljeni v trenutku, ko jih bo želel uporabiti; nadalje princip razširimo z možnostjo podajanja odvisnosti, ki se nahajajo v drugih domenah. V podporo principu porazdeljenega vrivanja odvisnosti zagotovimo ogrodje (vsebnik) za tvorjenje objektov in zagotavljanje njihovih odvisnosti preko domen. Predlagamo model sodelovanja med aplikacijami, po katerem vsebniki za porazdeljeno vrivanje odvisnosti delujejo kot povezana skupina. V ta namen jim zagotovimo vmesno plast programske opreme, ki omogoča enostavno izmenjavo primerkov objektov iz različnih domen, in je odporna na izpade v sistemu. Skupina povezanih podsistemov, ki tvorijo porazdeljeno vmesno plast programske opreme za repliciranje objektov, transparentno skrbi za replikacijo (kopiranje in razmeščanje na različna vozlišča v omrežju) objektov pripravljenih po načelu vrivanja odvisnosti. S striktnim zagotavljanjem usklajenosti (vernost) med kopijami (replikami) ob vsakem proženju metode dosežemo, da je ob izpadu nekega vozlišča, na voljo drugo vozlišče, kjer se nahaja verna kopija objekta. Usklajenosti med vsemi kopijami se zavestno odrečemo v primeru izpada omrežja, ko pride do izolacije vozlišč, s čimer jim omogočimo nadalje delovanje z uporabo izoliranih kopij. Uspešnost okrevanja sistema zagotovimo z beleženjem in shranjevanjem sprememb, razlike med

kopijami pa se ponovno uskladijo. Pri implementaciji prototipa izdelamo dodatek za vmesno plast ogrodja .NET ter razširimo vsebnik Unity.

Ključne besede: porazdeljeni sistemi, vmesna plast, objektno usmerjeno programiranje, vrivanje odvisnosti, replikacija, ogrodje .NET.

Razširjen povzetek

Porazdeljeno vrivanje odvisnosti je koncept sestavljen iz dveh pojmov: porazdelitev (angl. *distribution*) in vrivanje odvisnosti (angl. *dependency injection*). Skupna točka obema pojmoma je objekt.

Objekt (angl. *object*) opredeljujeta njegovo stanje in obnašanje. Objektno usmerjeno programiranje temelji na objektih in njihovem sodelovanju. Iz sodelovanja med njimi izhaja relacija odvisnosti. Pri opravljanju svojih funkcij vsak objekt uporablja še druge (sodelujoče) objekte (sodelavce/kolege), pri čemer je odvisen od njihovih storitev. Objekte uporabljamo pri izvedbi funkcionalnosti aplikacij, pri čemer interakcija med njimi poteka *zgolj* preko vmesnika (angl. *interface*), ki predstavlja nabor metod, ki jih kot storitve nudi drugim. Aplikacija predpostavlja, da za njo „stoji“ uporabnik in da se izvaja na računalniku.

Ob povečanemu številu objektov je v pomoč načrtovalski vzorec (angl. *design pattern*) za *vrivanje odvisnosti*. Osnovno načelo je, da objekt zgolj jasno izrazi svoje odvisnosti do drugih objektov in pričakuje, da mu bodo ti priskrbljeni v trenutku, ko jih bo želel uporabiti. Namen vrivanja odvisnosti je sam objekt razbremeniti skrbi „kateri“ so sodelujoči objekti, na „kakšen“ način in „kje“ jih pridobiti. Naloga „priskrbeti“ je prepuščena zunanji entiteti ali ogrodju; mi uporabimo vsebnik (angl. *container*). Bolj jasno povedano: objekt hrani reference na druge (odvisne) objekte kot interne spremenljivke, pričakuje pa, da bo odvisnosti pridobil „vrinjene“ preko konstruktorja ali nastavitvenih metod. Posledično lahko sam objekt nastopa kot odvisnost nekemu drugemu objektu. Končni rezultat verižnih navezovanj je kompleksen

graf objektov.

Vsebnik za vrivanje odvisnosti (angl. *dependency injection container*) poskrbi za tvorjenje objektov in vrivanje (angl. *injection*) vseh povezanih (odvisnih) objektov, po potrebi izvede tudi njihovo tvorjenje, ter deluje kot njih osrednje zbirališče. Vsebnik pa je v tem primeru, skupaj z vsemi objekti, ki jih nadzira, omejen zgolj na delovanje aplikacije, kateri služi.

Omenjena omejitev se pojavlja pri različnih aplikacijah, ki tečejo neodvisno, ali pa za svoje sodelovanje uporabljajo razne mehke oblike sodelovanja. Ker pa imajo aplikacije potrebo po skupnem usklajenem delovanju ter delitvi in vzajemni uporabi istih objektov, takšna omejitev/potreba pogosto vodi k sodelovanju v porazdeljenih sistemih.

Dandanes je že običajno, da so uporabniki, aplikacije in računalniški sistemi omreženi. Uporabniki so pri svojem delu in socialnih aktivnostih tega že toliko vajeni, da zlahka spregledajo celotno bistvo, to je da za njihovo omreževanje skrbi *porazdeljeni sistem* (angl. *distributed system*). Na primer: z uporabo brskalnika uporabnik dostopa do zmogljivosti strežnikov, njihove procesorske moči in „diskovja“, ter jih hkrati deli z mnogimi ostalimi uporabniki. Porazdeljeni sistem je tisti, ki uporabnika povezuje z oddaljenimi skupnimi viri (angl. *remote shared resources*). Gre za povezovanje pri prijavi v sistem, pri zagonu aplikacije, ipd. Pojmovanje je široko kot sam internet, ki je v bistvu porazdeljeni sistem. Bolj namenska je uporaba porazdeljenih sistemov na primer v spletnem bančništvu in zdravstvu, zelo specializirana uporaba pa za potrebe nadzornih sistemov (angl. *control sytstems*) kot npr. nadzor sistemov letenja in v vojaške namene. Pričakovano je, da so takšni sistemi zanesljivi in zagotavljajo nemoteno delovanje kljub prisotnosti napak in izpadom komponent.

Zasnova porazdeljenih sistemov z uporabo vmesne programske opreme omogoča enostavno uporabo skupnih virov preko omrežja. *Vmesna plast* programske opreme (angl. *middleware*) se preko omrežja poveže med računalniki in aplikacijam zagotavlja podporo za tvorjenje oddaljenih objektov (angl. *remote objects*), oddaljeno proženje metod (angl. *remote method invoca-*

tion, RMI), ipd. Najpogosteje se uporablja model oddaljenih objektov, pri čemer pa objekt (njegovo stanje) ni porazdeljen. Sam objekt je nameščen na strežniku, njegovi vmesniki pa se lahko uporabljajo v različnih drugih domenah/računalnikih. Strežnik je v tem primeru potencialna šibka točka, na kateri lahko sistem razpade na nedelujoče dele; zgolj zato, ker sistem za svoje delovanje potrebuje strežnik, ta pa je v sistemu zgolj en sam.

Kot odgovor na izpostavljene omejitve in šibkosti ponudimo koncept *porazdeljenega vrivanja odvisnosti* (angl. *distributed dependency injection*) in *vsebnik za porazdeljeno vrivanje odvisnosti* (angl. *distributed dependency injection container*). Z razširjenim konceptom vrivanja odvisnosti objektom omogočimo podajanje odvisnosti, ki se nahajajo v drugih domenah. Kot podpora temu konceptu se vsebnik za porazdeljeno vrivanje odvisnosti lahko uporablja v vseprisotnih aplikacijah: porazdeljeni vsebniki za vrivanje odvisnosti se povežejo med seboj, da si izmenjajo primerke objektov iz različnih domen in jih posredujejo aplikacijam, hkrati pa so odporni na izpade v sistemu.

V porazdeljenih sistemih se za izboljšanje njihovih lastnosti uporablja *replikacija* (angl. *replication*). Gre za proces večkratnega kopiranja (podvajanja) objektov, ki so razmeščeni na različnih vozliščih (angl. *nodes*) v omrežju. S tem se izboljša dostopnost (angl. *availability*) in zanesljivost (angl. *reliability*) sistema ter zagotovi odpornost proti napakam ter izpadom (angl. *fault tolerance*). Hkrati pa je replikacija proces ohranjanja vernosti kopij (*replik* (angl. *replica*)). Usklajenost (vernost) med kopijami se ohranja ob vsakem proženju metode – spremembe aplicirane na eni kopiji se razširijo na vse ostale kopije (propagiranje sprememb, angl. *update propagation*). Na tak način z replikacijo dosežemo, da je ob izpadu vozlišča, kjer se nahaja nek objekt, na voljo drugo vozlišče, kjer se nahaja verna kopija tega objekta (angl. *failover*).

Usklajenosti med kopijami, ki se sicer zagotavlja striktno, je možno začasno opustiti. Tako dosežemo večjo dostopnost objektov in s tem posledično dostopnost celotnega sistema. Usklajenosti med kopijami se zavestno odrečemo

v primeru izpada omrežja, ko pride do izolacije vozlišč (angl. *network partition*). Vozliščem v skupinah tako z uporabo izoliranih kopij omogočimo normalno nadaljnje delovanje, kljub prisotnosti izpada komponente sistema. Pri tem lahko pride do razlik med kopijami, ki se zaradi razdrtja sistema med seboj ne morejo usklajevati. Usklajenost med vsemi kopijami lahko ponovno zagotovimo, ko je napaka odpravljena. Postopek *ponovne uskladitve* (angl. *reconciliation*) zagotovi normalno nadaljnje delovanje sistema.

Velik del naloge posvetimo zagotavljanju robustne *porazdeljene vmesne plasti programske opreme za repliciranje objektov* (angl. *distributed object replication middleare*). Za objekte pripravljene po načelu vrivanja odvisnosti omogočimo skupno (porazdeljeno) uporabo preko različnih domen. Vsebniki za porazdeljeno vrivanje odvisnosti delujejo kot povezana skupina. Pri svojem delovanju uporabljajo vmesno plast, da jim za objekte, ki jih vsebniki tvorijo, zagotavlja replikacijo objektov ter dostop do objektov na različnih vozliščih.

Vmesno plast programske opreme sestavlja skupina spodaj navedenih podsistemov, ki primarno skrbijo za replikacijo.

Podsistem za proženje metod (angl. *invocation service*) Podsistem je zadolžen, da prestreže metode, ki jih na objektu proži aplikacija. Vmesna plast namreč priskrbi lasten način izvrševanja metode. Sprožena metoda je v izvajanje posredovana podsistemu za replikacijo.

Podsistem za replikacijo (angl. *replication service*) Podsistem za replikacijo je sestavljen iz dveh delov. *Upravitelj za replikacijo* (angl. *replication manager*) ima za nadzor nad kopijami objektov in je zadolžen tudi za njihovo podvajanje. Drugi del podsistema predstavlja *protokol za replikacijo* (angl. *replication protocol*), ki določa kako naj poteka tekoče usklajevanje med kopijami ob proženju metod, kakor tudi priprave ter sam postopek okrevanja po napaki.

Podsistem za replikacijo od podsistema za proženje metod v izvajanje prejme sproženo metodo. Upravitelj za replikacijo priskrbi ciljni primerek replike na kateri bo metoda izvršena. Ta primerek se lahko nahaja

v drugi domeni. Potem, ko je metoda izvršena na ciljnem primerku replike, se morebitne spremembe razpršijo na vse ostale kopije tega istega primerka, kot to določa protokol za replikacijo.

Podsistem za komunikacijo (angl. *communication service*) Podsistem skrbi za usklajeno komunikacijo med posameznimi člani skupine (angl. *group communication*, npr. pri usklajevanju replik), ki so lahko razporejeni na različnih vozliščih. Del tega podsistema skrbi za nadzor katera vozlišča skupine so dosegljiva (*group membership*), to je delujoča in znotraj iste izolirane skupine, v primeru izpada omrežja. Kot del nadzora je vključeno tudi javljanje sprememb v konfiguraciji vozlišč, torej tudi zaznavanje izpadov.

Pomožne komponente, ki niso ključne za replikacijo, vključujejo še imenski podsistem (angl. *naming service*), podsistem za aktiviranje objektov (angl. *activation service*) in podsistem za transakcije (angl. *transaction service*).

Replikacija s kopiranjem objektov in ohranjanjem vernosti kopij, zaznavanje izpadov, delovanje kljub prisotnosti izpada komponente sistema in okrevanje ob odpravi napake skupaj zagotavljajo *robustnost*.

Izvedljivost koncepta porazdeljenega vrivanje odvisnosti preverimo z implementacijo prototipa vsebnika za porazdeljeno vrivanje odvisnosti, ki za svoje delovanje uporablja porazdeljeno vmesno plast programske opreme za repliciranje objektov. V nalogi predstavimo podrobnosti implementacije, ki temelji na Microsoftovem .NET ogrodju z uporabo C# programskega jezika.

Prikažemo uporabo porazdeljenega vsebnika za porazdeljeno vrivanje odvisnosti pri izdelavi porazdeljenih aplikacij ter potrdimo različne scenarije zagotavljanja robustnosti.

Koncept porazdeljenega vrivanja odvisnosti je tako mogoče uporabiti že pri samem načrtovanju novega porazdeljenega sistema. Poleg tega je možno koncept vpeljati v že obstoječ sistem.

Prva možnost je uporaba v obstoječem porazdeljenem sistemu na osnovi oddaljenih objektov. Za sisteme grajene po tem modelu smo izpostavili problem šibke točke. Dodana vrednost uporabe porazdeljenega vrivanja odvisnosti v sklopu z vsebnikom za porazdeljeno vrivanje odvisnosti so izboljšana dostopnost in robustnost, ter dodatno še prednosti uporabe vrivanja odvisnosti kot načrtovalskega vzorca. Po drugi plati pa lahko pri aplikaciji, izdelani z uporabo vsebnika za vrivanja odvisnosti, pride do spremembe potreb, ki vodijo v prehod na porazdeljeni sistem. V tem primeru je zelo enostavno zamenjati običajen vsebnik s porazdeljenim.

Ključne besede: porazdeljeni sistemi, vmesna plast, objektno usmerjeno programiranje, vrivanje odvisnosti, replikacija, ogrodje .NET.

Chapter 1

Introduction

You want the truth? You can't handle the truth!

—A Few Good Men

The way programmers go about solving nearly all problems is by abstracting everything as an object. Objects, which may consist of state and related behavior, are used to implement the functionality of applications, to store data and to compose the user interfaces. Programs that are constructed by object-oriented principles are made of objects that interact with one another. This undoubtedly daunting task can sometimes be aided by reusable solutions to common problems, collected as best practices in the form of design patterns.

Application users, involved in their everyday work and social activities, are so accustomed to a "connected state" that they may easily overlook the entire significance of a distributed system. For example, using a search engine without regard for the servers that provide these; or simply logging into a computer or its many form factors; or running an application — an "app". It is the distributed system that connects the user with the remote resources. Like browsing a set of web pages[1]. In this well-known case, the Internet plays the role of a distributed system which enables users to access data from the World Wide Web services that make use of resources such as disks and processors on which they are implemented. Internet is actually a very

[1]Based on Coulouris et al. [15].

large distributed system. Programs running on the computers connected to it interact by passing messages. The Internet communication mechanisms (the Internet protocols) enable a program running anywhere to address messages to programs anywhere else, abstracting over different technologies; LAN, Wi-Fi, Bluetooth and mobile phone networks. This demonstrates the immediate need for distributed systems that are present in a wide range of applications, from daily life such as banking and health care applications to highly specialized distributed systems used in control engineering, air traffic control and military. Above all, such systems need to be reliable, which means they have to sustain failures.

1.1 Motivation

Interactions between objects create dependencies. One object has to know about another object — this is called a dependency. Dependency injection is a design pattern concerned with resolving dependencies. A fundamental philosophy is that an object requires its dependencies to be provided, rather than creating or finding them on its own. To be completely direct: an object stores references to other objects — its dependencies — as (instance) variables, and accepts its dependencies "injected" through constructors or setter methods. Typically these references-dependencies are isolated to a single application domain (see Glossary), process and computer.

In this thesis we consider cases where one object references other objects that reside outside of its own application domain, possibly on another computer. This is the case most commonly found in distributed object systems.

> The main focus of this thesis is a new concept of a distributed dependency injection in which objects require dependencies from across application domains, processes and/or computers.

Dependency injection container is the external "injector" entity that takes on the responsibility of creating objects and providing their dependencies.

Normally, these operations of the container are limited to the domain of a program that created them and survive only as long as that program continues to run. In order for the container to resolve dependencies distributed across application domains and to be accessible by multiple programs from multiple computers and its resources — i.e. the objects — shared, is a common motivating factor for distribution.

> The second focus of this thesis is to create a distributed dependency injection container that can resolve distributed dependencies and be used by multiple programs in order to share the objects that it manages.

Distributed object systems extend the object-oriented concepts across network boundaries while middleware aims to hide the complexities of distribution as much as possible. Distributed object systems allow objects to invoke methods on remote objects using the same syntax as for local invocations. Most object-based distributed systems use a remote-object model in which an object's state is not distributed: it resides at a single server computer, only the interfaces implemented by the object are made available on other computers. The server, in this case, is the system's potential weakness as a single point of failure. Replication, the process of maintaining multiple copies of the same object, is well-known to provide fault-tolerance for improved availability in case of node and link failures.

> The third focus of this thesis is to devise a reliable underlying distributed system middleware architecture that uses replication to achieve fault-tolerance.

Our proof-of-concept implementation is based on .NET framework using the C# programming language.

> The fourth focus of this thesis is the demonstration, validation and evaluation of the distributed dependency injection concept in building reliable distributed applications using a distributed dependency injection container.

The motivation for this thesis comes from a more personal view on the state of software engineering and programming as a discipline [18], which verges on art [35], of "*telling computers what to do*":

> We consider the dependency management in object-oriented design and application development to be the result of good practices in software engineering. We recognize the concept of dependency injection as a "means to an end" of managing stateful objects, by using dependency injection containers. And further find the limitations of a single application-computer reasons for distribution. For this we consider the object-based approach for the development of distributed systems as it has a common ground with dependency injection — objects are the fundamental resources. And finally, require a fault-tolerant distributed system for the task.

1.2 Contributions

The contribution of this thesis is four-fold:

First, a concept of dependency injection is extended across application domains. The extended concept supports objects which require dependencies that are distributed across application domains, processes and/or computers. The new concept is coined as distributed dependency injection.

Second, a new way for various applications to interact by sharing objects over interconnected domains that can reside on multiple computers to manage their dependencies is defined. This is via a distributed dependency injection container, to enhance the concept of a distributed dependency injection.

Third, a distributed object replication middleware architecture is presented. It is devised as an extension to .NET Remoting middleware architecture to support the fault-tolerant sharing of objects.

Fourth, a prototype implementation of a distributed dependency injection container is provided in .NET on top of the replication add-on to .NET Remoting. Using the prototype we demonstrate how the concept of a distributed

dependency injection can be used in building reliable distributed applications. We show the simplicity of the programming model by which replication is provided in a fully transparent manner. Replication and fault-tolerance are submitted to validation and further to empirical evaluation that shows that the replication and fault-tolerance do not account for a significant overhead as compared to existing remote-object interactions.

1.3 Structure of this Thesis

The remainder of this thesis is structured a follows: Technology that is the basis for this thesis in introduced in Chapter 2, along with the overview of related work and the description of our approach. Our first three main contributions are presented in Chapter 3. The concept of a distributed dependency injection is presented and afterwards the distributed dependency injection container in support of the concept, along with the distributed object replication middleware. This is followed by fulfillment of the fourth main contribution — the prototype implementation, validation and evaluation — in Chapter 4. Finally, a summary, conclusions and further work are presented in Chapter 5.

This page intentionally left blank.

Chapter 2

Technology Overview

After NT (New Technology) Technology comes .NET (dot net) Technology.

—Microsoft

This chapter provides an overview of the technology that is the basis for this thesis. Presented are the object technology and the reasoning behind object management with the use of dependency injection within the context of dependency injection containers and finally distribution of objects through replication with the assistance of middleware. Related work is over-viewed along with the .NET technology that is used in the implementation of the approach taken in this thesis.

2.1 Object Technology

The key feature of an object is that it encapsulates both data and code, and interacts with the outside world through well-defined interfaces. Furthermore, an object may implement multiple interfaces. Likewise, given an interface definition, there may be several objects that offer an implementation for it. This allows the introduction of new objects or the replacement of the same object with another (compatible) one.

Figure 2.1 shows how objects are involved in implementing the functionality of an application.

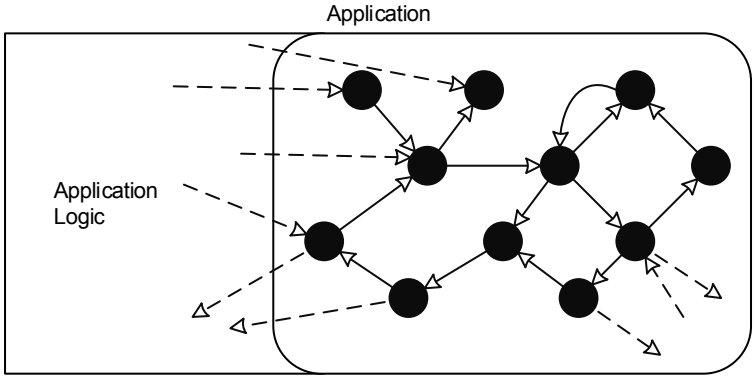

Figure 2.1: **Application composed of objects** that interact with one another.

2.1.1 Stateful Objects

A stateful object is an object that maintains and changes internal state over time. State alternation occur during the invocation of its methods. As the result, stateful objects can exhibit behavior that varies based upon what the object has already done. For example, a `GetNextItem()` method returns the next item each time it is called. This behavior is enabled by saving state associated with the current instance.

State is specific to the object. Different object instances of the same class in the same server process maintain their own state. In the case of an exclusive stateful object, state is specific to a given client application. Stateful non-exclusive objects maintain a common state across a set of client applications.

While stateful objects are very powerful, server-side failures can present problems. For example, let us assume that a client is scrolling through a set of records maintained by a server object. If the server fails its state could be lost. The object is capable of being reactivated but the appropriate record

would not be returned the next time `GetNextItem()` operation was called. Client applications might need to recreate their state within the new object. Another option would be for the object to persist its state and retrieve its state upon reactivation.

2.1.2 Principles of Object-Oriented Design

The desired properties of an object can be described as the five SOLID principles of object-oriented design [38] listed in Table 2.1. The principles, however, raise the following issue on dependency management [39]: "Because well-designed objects have a single responsibility, their very nature requires that they collaborate to accomplish complex tasks. This collaboration is powerful and perilous. To collaborate, an object must know something about others. Knowing creates a dependency. If not managed carefully, these dependencies will strangle your application."

In Figure 2.1 the application is performing its own dependency management. Poor dependency management leads to code that is hard to change, fragile, and non-reusable — the "code smells" [24].

One way to solve the issue is dependency injection, presented in section 2.2.

Single responsibility principle	A class should have one, and only one, reason to change.
Open/close principle	You should be able to extend a class's behavior, without modifying it.
Liskov substitution principle	Derived classes must be substitutable for their base classes.
Interface segregation principle	Make fine grained interfaces that are client specific.
Dependency inversion principle	Depend on abstractions, not on concretions.

Table 2.1: **Five SOLID principles of object-oriented design.**

2.1.3 Distribution of Objects

"Ever since they [objects] were created, folks have wanted to distribute them" [22]. Along the way [20], "all aspects of object orientation (concepts, languages,

representations and products) have reached a stage of maturity where it is possible to address complex issues of distribution using the object-oriented paradigm."

A distributed object is — with a bit of a background — presented in section 2.3.

2.2 Dependency Injection

Dependency injection is a design pattern concerned with resolving dependencies. A fundamental philosophy is that an object explicitly declares and requires its dependencies to be provided, rather than creating or finding them on its own. That is, the dependencies one object has towards another object to fulfill its own functionality; and relies entirely upon an external entity that is in control of providing (passing) dependencies to the object.

To be completely direct, in Listing 2.1: an object stores references to other objects — its dependencies — as (instance) variables (lines 3–10), and accepts its dependencies "injected" through the constructor (line 12) and/or a setter method (lines 9–10).

```
1  class SomeClass
2  {
3      private readonly IInterfaceOne first;
4
5      private readonly IInterfaceTwo second;
6
7      private readonly ClassThree three;
8
9      [Dependency]
10     public IInterfaceFour Four { get; set; }
11
12     SomeClass(IInterfaceOne first, IInterfaceTwo second, ClassThree three)
13     {
14         this.first = first;
15         this.second = second;
16         this.three = three;
17     }
18 }
```

Listing 2.1: **Stating dependencies for injection.**

Dependencies arise because objects are often a part of a set of collaborating components which depend upon other objects (*collaborators*) to successfully complete their intended purpose. In many scenarios, objects need to know "which" other objects to communicate with, "where" to locate them, and "how" to communicate with them.

One way of structuring the code is to have the logic of locating and/or instantiating the collaborators embedded within the class as a part of object's usual logic (*implicit dependencies*). Another way is to separate the construction logic from the object's behavior: firstly have the object publicly declare its dependency on other objects, and then have some "external" piece of code assume the responsibility of locating and/or instantiating the collaborators and simply supplying the relevant collaborator references to the top object when needed. Such *explicit dependencies* appear most often in an object's constructor, for class-level dependencies, or in a particular method's parameter list, for more local dependencies. Using explicit dependencies is considered to be an effectuation of dependency injection.

Dependency injection involves the four roles described below.

Object An object depending on the collaborators that it consumes.

Collaborator The collaborator object(s) to be used.

Interface The interfaces that define how the object interacts with the collaborators.

Injector The injector responsible for constructing the collaborators and injecting them into the object.

At any other situation the object itself could assume the role of the collaborator as a dependency to another object. The interfaces may truly be interface types, abstract classes or even concrete classes. It's only required that the object itself does not know which they are and therefore never treats them as concrete, say by constructing or extending them.

The injector introduces the collaborators into the object. Most often, it constructs also the object itself. Given that an object can later be regarded as a dependency to another object, an injector may connect together a very complex object graph.

The injector can choose to substitute different concrete class implementations of an interface at run-time. Being able to make this decision at run-time rather than compile time is the key advantage of dependency injection.

The injector is likely to be either hand coded or implemented using one of a variety of dependency injection frameworks. The injector may be referred to by other names such as: assembler, provider, builder, or spring. We will settle on using a "container".

2.2.1 Dependency Injection Container

Dependency injection container performs the task of an injector, i.e. creating, wiring and assembling the dependencies into an object graph.

In addition it assumes the responsibility for object life cycle management. For example, should a new instance be created every time (and forgotten)? Or should the same (singleton) instance be reused? The fact that it sometimes keeps a reference to the object after instantiating it, is the reason it is called a "container".

Dependency injection container provides methods to register type mappings and object instances, resolve objects and inject them with dependent objects. This is further described in more detail for the Unity container (subsection 2.2.2).

A dependency injection container is used by initially configuring type registration information. To *register*, Unity exposes a `RegisterType` method that maps an interface type to a concrete type. Later on, objects of the known types can be *resolved* from the container. The `Resolve` method causes the container to build an instance of the requested type. During this process the container will also inject any of the required dependencies into the object that it creates, forming a kind of a graph structure. Additionally,

Unity offers `RegisterInstance` method to register an existing instance with the container. This instance is shared along with the objects the container instantiates. The container is further responsible for maintaining instances and making them available for later retrieval.

There are many excellent dependency injection containers available for .NET. Like specific free containers: Castle Windsor[1], Spring.NET[2], Autofac[3], Ninject[4], and others. These vary in market share as well as design philosophies and purpose. The selected Unity[5] container is widely[6] used, well documented and has a semi Microsoft backing. A compact overview is given in the following subsection.

2.2.2 Unity Dependency Injection Container (Unity)

Unity container (Unity) [9] is a lightweight, extensible dependency injection container for use in .NET based applications. It provides support for injection mechanisms, through ability to register type mappings and object instances, resolve objects and manage object lifetimes.

Additionally, Unity offers a comprehensive XML configuration scheme that supports mappings defined and also exchanged at run time (i.e. without recompiling the application).

Finally, Unity can be extended to customize the registration and resolve processes and add new capabilities.

[1] `http://www.castleproject.org/`
[2] `http://www.springframework.net/`
[3] `http://autofac.org/`
[4] `http://ninject.org/`
[5] `https://github.com/unitycontainer/unity/`
[6] According to this completely unscientific Internet poll, Unity is the most widely used dependency injection container: Oliver Sturm, Poll Results: IoC containers for .NET. 2010, `https://oliversturm.com/poll-results-ioc-containers-for-net/`

2.2.3 Using Dependency Injection with Dependency Injection Container

Applying the dependency injection (DI) principles provides a number of advantages, mostly following from reduced coupling between objects. Dependency injection pushes the concept of separation to the point where one object is unaware even of the other objects that it depends on to fulfill its function. It does not even know what classes implement them or how to instantiate them, the only common ground is the communication provided by their interfaces. This separation of concerns is also known as loose coupling [46] and relates [9] to all SOLID principles from Table 2.1 [42]: "The idea of not explicitly knowing is central to DI. More accurately, not asking for dependencies and instead having them provided". Otherwise known as the Hollywood principle: "Don't call us, we'll call you".

Further benefits include [19]:

- Objects can be reconfigured by changing the implementations of their dependencies without any modification to the components themselves.
- Objects can be more easily tested by substituting mock implementations of their dependencies. While mocking is not new, the component design encouraged by dependency injection makes it particularly easy to substitute mock objects.
- Each object's dependencies are explicit, appearing as formal arguments of the constructor and initialization methods, so the component's interaction with the rest of the system is largely self-documenting.

As [4] states, dependency injection and dependency injection container are two completely independent concepts, and emphasizes:

$$\text{Dependency injection} \neq \text{using a DI container} \tag{2.1}$$

Regarding their relationship, the site posts the following matrix:

	No DI container	**DI container**
No DI	Spaghetti Code	Service Locator Anti-Pattern
DI	Manual Dependency Injection	Enhanced Dependency Injection

The bottom line here is that application frameworks that support dependency injection are not required for use of dependency injection principle.

Dependency injection merely prescribes ways to handle the dependencies between objects. Dependency injection works in the same way that "parameter passing" works. This certainly does not require a dependency injection framework and can be performed manually. For this, the `main` method makes a fairly good injection point. What makes it in part a good design is that the configuration is not spread throughout the code base. It's confined to one place per application.

It is, when a system is designed to use dependency injection, with many classes requesting their dependencies via their constructor (or methods), helpful to have a tool dedicated to creating these classes with their associated dependencies. Dependency injection container provides automatic dependency injection and, by doing the "wiring" for us, takes care of a lot of the application's infrastructure.

The most effective way — from the developer's view-point, to benefit the most — is when dependency injection is used within the context of a dependency injection container [9]: Modern business applications consist of custom business objects and components that perform specific tasks or generic tasks within the application. The key to successfully building these types of applications is to achieve a decoupled or very loosely coupled design. Where such applications benefit is that dependency injection is an interface programming technique based on altering class behavior without changing the class internals. Developers code against an interface for the class and use a container that injects dependent object instances into the class based on the interface or object type. Loosely coupled applications are more flexible and easier to maintain. They are also easier to test during development.

The correct way of using a dependency injection container is to make sure that application code is unaware of any container [4]: "a container should not affect the structure, design or implementation of your code except in a few isolated, well-defined places."

In Figure 2.2 an application is shown using a dependency injection container for dependency management. By doing so — in contrast to Figure 2.1 — the core application logic is — as per dependency injection principle — relieved of having to locate and/or create the objects and their dependencies. It's only but a few boundary case objects that are of the primary interest to the core application logic. From the container, the application logic acquires references to the objects it requires, as shown in empty circles. While remaining managed by the container, the objects, and their further dependencies — the latter shown as minimized circles — are of lesser significance to the application logic.

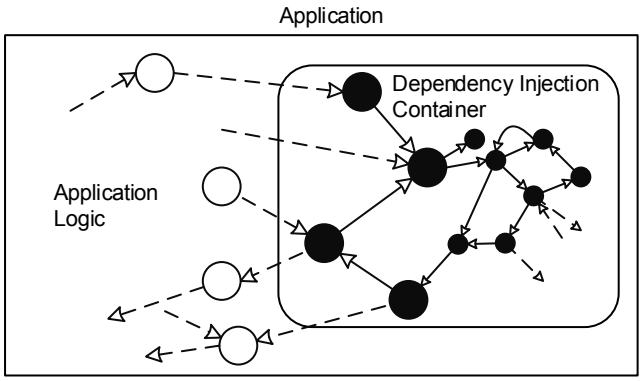

Figure 2.2: **Application using a dependency injection container** for dependency management.

2.3 Distributed Systems

Distributed systems are used by applications to provide their users the processing logic and the support of execution across multiple computers. Computers that are connected by a network may be on separate continents, in the same building or in the same physical machine. There are many considerations to be made before making the decision to construct a distributed system; as there are many issues with the use of distributed systems. However some non-functional requirements cannot be achieved by a centralized system, and lead to distribution. In particular: resource sharing and fault tolerance.

An independent view of the distributed system would be[7] [20]: a collection of components that reside on different computers. Each host executes components and operates a distribution middleware, which enables the components to coordinate their activities in such a way that users perceive the system as a single, integrated computing facility. Components are autonomous; there is no master component that possesses control over all the components of a distributed system.

Utilization of middleware accounts for interaction among components across a computer network at a higher level than by means of ports and sockets and streams. As a layer, implementing general-purpose services, it extends over multiple computers to offer each application the same capabilities.

While best known for its client-server approach, MacDonald [37] argues that distributed computing is not nearly as uniform, and is often misrepresented equating it with multi-tier or stateless design. With the remote communication process abstracted away by the middleware the terms 'client' and 'server' apply only to the roles played in a single remote invocation: When the client invokes an operation upon the server a request for an operation to be carried out is issued in a message from the client to the server. The server accepts the request and responds appropriately and a reply is sent in a message from the server to the client.

Distributed software requires the underlying distributed system to be

[7] According to Emmerich [20].

reliable. Building reliable systems from unreliable components requires fault tolerance: detecting failures, masking failures, tolerating failures and recovering from failures. Components can fail independently (node failure) and link failures may lead to network partitions, effectively splitting a system into parts that are not able to communicate. With the client-server approach the server is a potential weakness as a single point of failure and a potential performance bottleneck. Replication, the process of maintaining multiple copies of the same entity (data item, object), is well known to provide reliability through fault tolerance for improved availability in case of node and link failures; as further discussed in the 2.3.3 subsection below.

Distributed object systems are built using the object-oriented paradigm and allow objects to invoke methods on remote objects using the same syntax as for local invocations (i.e. objects collaboration across multiple domains, multiple computers). The separation between interfaces and the implementing objects allows for an interface to be placed at one machine, while the object itself resides on another machine — organization of a *distributed object*.

Similar to the principles on object design (Table 2.1) a list on the distributed design is provided in Table 2.2 [8].

Principle 1:	Distribute Sparingly
Principle 2:	Localize Related Concerns
Principle 3:	Use Chunky Instead of Chatty Interfaces
Principle 4:	Prefer Stateless Over Stateful Objects
Principle 5:	Program to an Interface, Not an Implementation

Table 2.2: **Five principles of distributed design.**

Before moving on to the middleware that is based on objects, here are a few other types of middleware worth mentioning[8] [20]: (i) *Transaction-oriented middleware* supports transactions across different distributed database systems. It may be used to maintain replicated databases on different servers and provides fault-tolerance and load balancing. Updates to these databases

[8]According to Emmerich [20].

are synchronized using distributed transactions; for this the two-phase commit protocol is used. (ii) *Message-oriented middleware* supports the communication between distributed system components by facilitating message exchange. Messages are sent in order to request execution as are the results transmitted via another message. The message delivery takes place asynchronously, as the client continues the processing as soon as the middleware has taken the message. This achieves de-coupling of client and server and leads to more fault tolerant and more scalable systems. A strength of message-oriented middleware is that it supports multi-casting; it can distribute the same message to multiple receivers. Both middleware types are used in distributed object middleware, as transaction service and as reliable multi-cast for group communication. (iii) *Remote procedure calls (RPCs)* are also considered a type of middleware. Many distribution aspects of objects have their origin in the Remote Procedure Calls (RPCs); clients can call procedures that run on remote machines.

2.3.1 Distributed Object Middleware

The *middleware* associated with distribution of objects provides means for objects to interact with each other. With distributed objects it is relatively easy to hide distribution aspects behind an object's interface. This implies masking the complexity and low-level primitives of the underlying networks, hardware and operating systems; the likes of sockets and byte streams, the ISO/OSI model and the TCP/IP transport protocols. It includes operations such as instantiating new objects, passing object references, invoking methods on objects and de-allocating them.

The three most prominent examples of object-oriented middleware are, best known as: CORBA, COM and Java/RMI.

The *Common Object Request Broker Architecture*, simply referred to as *CORBA* [1], is a specification defined by the Object Management Group, an open-membership, not-for-profit consortium with hundreds of participants. It is designed to facilitate the communication of systems that are deployed

on diverse platforms and enables collaboration between systems on different operating systems, programming languages, and computing hardware. Implementations are offered by various third parties. It is the middleware of choice for mission-critical and high-availability applications.

Common Object Model (COM), that includes distribution capabilities, or Distributed COM (DCOM) [2], is a framework for distributed objects on the Microsoft platform implemented on top of the various Windows operating systems. Since virtually all Windows applications make use of its functionality it is perhaps the most widely used middleware related to distributed systems.

Java's *Remote Method Invocation (RMI)* [6] extends the Java object model to provide support for distributed objects in the Java language. It is the most likely choice for (internet) applications that need to be portable across a large number of platforms.

While CORBA and DCOM allow for objects to be written in different languages, Java/RMI supports only Java language. Detailed descriptions and comparisons of the three systems, each with its own merits and disadvantages, are easy to find, with the provided references as starting points. And as a conclusion, a slightly biased criticism of all three systems: Many of CORBA's APIs are far larger than necessary and the extent of the CORBA specification is overwhelming as it is actually quite surprising there is still any agreement at all ([33] is a meaningful reference). DCOM and Java/RMI are both additions to an already existing technologies: DCOM had a well-established and difficult model to build on, while staying compatible with even the early Window systems (and it "is not hard to criticize" DCOM for its poor design decisions). As RMI was added to Java after the original release of the platform this also produced some short comings with the integration.

A relatively late entry to distributed technologies is called *.NET Remoting*, acting as .NET replacement for DCOM. It introduces a slew of much-needed refinements, including the capability to configure a component in code or through simple XML files, communicate using compact binary messages or platform-independent SOAP, and control object lifetime using flexible

lease-based policies and is also completely customizable and expendable.

2.3.2 .NET Remoting

Microsoft's .NET framework offers a complete package of tools and technologies for developing applications. This includes distributed applications, through an extensible distributed object computing middleware infrastructure of *.NET Remoting*. It has always been a part of the .NET framework from the beginning and provides deep integration with the underlying platform. It allows objects to be written in different languages — the ones supported by the .NET framework. Related to application domains, no direct communication can be achieved across application domains. The term *remoting* refers to the use of these technologies to invoke methods on and share data with objects running in another *application domain*. In .NET, any object outside the application domain of the caller should be considered remote.

An instance of the remote object is made available to the client application using a *proxy* pattern [27]. To the client, this proxy seems to be a local instance of the remote object, but it simply references the remote object on the server. The client proxy communicates with recipient proxy object at the server side which in terms makes the call to the remote object. The same works in reverse when proxy objects handle the information that is returned from the server, and is handed to the calling client application. This is further described in more detail and Figure 2.3 depicts this process.

A remote object inherits from `MarshalByRefObject` class, defined in the `System` namespace. Upon creation the client receives a reference to a proxy object that stands in for the remote object. As for the server side the "*object is marshaled*": the .NET Remoting infrastructure publishes a server proxy at a Uniform Resource Identifier (URI) to listen for incoming requests and further control the access to the remote object. Client proxy forwards invocations to the correct remote object instance by using the corresponding URI.

For remote object invocations .NET Remoting employs the method-call-

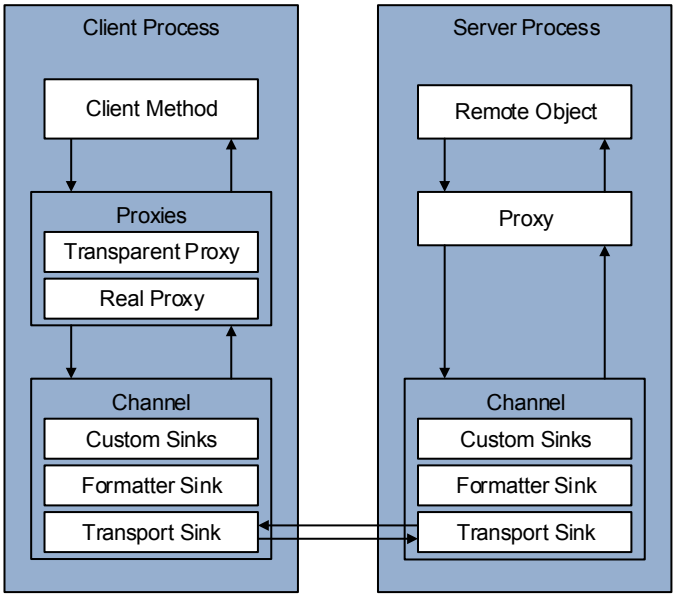

Figure 2.3: **The .NET Remoting Architecture.**

as-message concept: Method calls issued by the client are intercepted by a local instance of `TransparentProxy` class that is presented to the client by the framework. The method call (including the call parameters) is transformed into a message-object. A message internally uses a dictionary that holds named properties associated with values that describe various aspects of the called method: information such as the URI of the remote object, the name of the invoked method, and parameters if any. This is conveniently hidden behind a single `IMessage` interface, defined in the `System.Runtime.Remoting.Messaging` namespace. Such a message is forwarded to a second proxy type by calling its `Invoke` method. This instance of `RealProxy` class (`System.Runtime.Remoting.Proxies` namespace) is then responsible for forwarding the message to the .NET Remoting infrastructure for eventual delivery to the remote object.

The transfer of the message between the applications across the remoting boundary is done by a so-called message *sink chain*. This is a

series of concatenated sink objects. A sink will basically receive a message from the preceding sink, apply its own processing, and delegate any additional work to the next sink in a chain. Part of the chain is supplied by channel objects, including a formatter and a transport sink that take care of the actual physical message transportation. *Channels* are registered per application domain on either side of the remoting boundary before — and are shared by — any objects are registered. For this purpose there are three basic interfaces for message sinks: `IMessageSink`, defined in the `System.Runtime.Remoting.Messaging` namespace, and `IClientChannelSink` and `IServerChannelSink` defined in the `System.Runtime.Remoting.Channels` namespace.

Processing on the first sink is initiated by the real-proxy object. The message is then passed from one sink to the next. At some point in the chain, the message reaches a formatter that will serialize (*marshal*) the message along with the values in the dictionary to a defined stream format. Ultimately the message reaches the transport sink at the end of the chain. The transport sink is responsible for establishing a connection (through sockets and ports) to its counterpart on the server side and sending the byte stream. The transport sink on the server passes the byte stream through a similar sink chain on the server side until it reaches the formatter sink, at which point the message is deserialized from its point of dispatch to the remote object itself. This actually makes the method call on the remote object. When the method call returns, the same procedure is used to return results. The return result and any output arguments are packaged into a return message which is then passed back down the sink chain for eventual delivery to the proxy in the caller.

There are several message object types that implement the `IMessage` interface: `ConstructionCall` and `MethodCall`, and their respective return messages. The message types are serializable as required for message transportation. This requires also that all method call parameters are passed by value and should be marked with the `[Serializable]` attribute, or

implement the `ISerializable` interface.

To summarize, the framework fulfills the task of middleware. The complexities of calling methods on remote objects and returning results are handled by automatic creation and management of proxies, sinks and channels. These also provide for the extensibility of .NET Remoting. It is possible to create and plug in proxy classes that customize proxy creation, object marshaling (making the object available remotely), and other proxy-related tasks. Sink chains, that are customizable only in the section provided by the channel objects, can be used to manipulate the message objects.

> The described is a remote-object model in which object's state is not distributed: it resides at a single server computer, only the interfaces implemented by the object are made available on other computers. Indeed, .NET Remoting does not provide an adequate support for replicating objects that could be used for fault-tolerance.

2.3.3 Replication

For fault tolerance, high degree of availability, increased reliability and scalability, a distributed system allow replication of resources. If one replica fails other replicas can stand in, thus allowing the system to operate even in the presence of faults. Additionally, more resources are available for efficient use — load balancing between replicas leads to better performance. Also having a copy nearby can hide much of the communication latency problems. Scalability problems often appear in the form of performance degradation upon a growing load when more components are added to the system and more concurrent requests are introduced. The system can be scaled by introducing more replicas, preferably placing them on new hosts.

Regarding replication and fault tolerance we point to Table 2.3. CAP theorem [28] states that it is impossible for a distributed computer system to simultaneously provide all three of the guarantees. However — in any

Consistency	All nodes see the same data at the same time.
Availability	Every request receives a response about whether it succeeded or failed.
Partition-tolerance	The system continues to operate despite arbitrary partitioning due to network failures.

Table 2.3: **CAP theorem guaranties.**

situation — , the stronger guarantees are provided for two of these properties, the weaker guarantees can be provided for the third.

Replication [32] takes place, by creating several copies — replicas — spread across different nodes. After that, replication is a continuous process of maintaining replicas consistent upon invocations by propagating updates. There is often a trade-off to availability by relaxing consistency during degraded scenarios. This is the case with link failures, when nodes are split into partitions: Availability is gained by allowing "normal" operations among the available replicas. At the same time — by allowing separate updates — the consistency among all replicas is broken. Full consistency is re-established after the failures are eventually repaired; through reconciliation: In preparation, recovery from failures is ensured by logging and checkpointing the state of the system on a regular basis. This requires persistence management and that the updates are expressed in such a way that they can be merged. Operations that lead to conflicting updates need to be detected and resolved, even discarded.

2.4 Related Work

Distributed dependency injection is to our knowledge a new construct. We conclude this since no previous occurrences or references have been found.

However separately distributed system and dependency injection are well known. Dependency injection is a well-established design pattern that is implemented in the form of various containers. Replications has been used in several middleware systems for distributed objects. However, most of the

work has been focused on CORBA.

In the scope of .NET environment, Seidmann [47] implemented object replication; however, in the context of a distributed shared memory, while we focus on a flexible replication middleware. Reiser et al. [43] developed a replication framework for services. However, this approach is not suitable for stateful objects or concurrent accessed by more than one client since total order of the invocations is not guaranteed. Noted are also the efforts of the XPrevail (`http://xprevail.sourceforge.net/`) project as a prevalence layer with replication capabilities in its infancies.

2.5 Approach Taken in this Thesis

With our approach we provide means for objects to declare and require their dependencies to be provided across application boundaries in a dependency injection manner. To provide for these dependencies a container is made accessible to various programs interconnected from multiple computers. A distributed object system is employed in object dependency management. An off-the shelf middleware solution for distributed objects is extended by a collection of services that support:

- Truly — through replication — distributed object/dependency management.
- Transparent fault tolerance — node and link failures are considered.
- Propagation of updates upon modifications.
- Trade-off between availability and consistency during degraded scenarios.
- Recovery from failures by logging and checkpointing.
- Resolution of conflicting modifications.

The architecture is based on the DeDiSys (Dependable Distributed Systems) research project [3, 48], which is focused on optimizing dependability in distributed software and service systems. Among others, the project aims at balancing (trading) availability and consistency in partitioned environments.

The studies are oriented towards providing fault-tolerance through add-ons for various middleware. To this end the research defines a platform independent architecture [25, 40] and conducts a comparison [26] of refined designs of EJB [36], CORBA [49, 11] and .NET [31, 41] software prototypes.

A modern and highly endorsed .NET technology has been chosen for a proof-of-concept implementation. Unity container is extended over application domains by leveraging our object replication middleware extension to .NET Remoting. The C# programming language has been chosen for implementation using Windows as underlying operating system.

Related to the five principles of distributed design, presented in Table 2.2, we oppose the 4$^{\text{th}}$ principle on "fullness" of the object. It contradicts our approach as we focus on dynamic objects involved in the dependency injection that poses no such limitation. At the same time we would like to promote the remaining principles as they encourage that the distribution is taken into account by the designers. The 3$^{\text{rd}}$ principle on the granularity of interfaces actually contradicts and should be balanced against the **I** (Interface segregation principle) of the SOLID principles from Table 2.1. The 5$^{\text{th}}$ and the **D** (Dependency inversion principle) principles are particularly of interest as both strongly endorse the dependency-injection principle.

This page intentionally left blank.

Chapter 3

Distributed Dependency Injection

Dependency Injection is all the rage.

—Robert C. Martin ("Uncle BOB")

This chapter promotes our concept of a distributed dependency injection. In its support, we provide a distributed dependency injection container. A distributed object replication middleware is employed by such a container in (distributed) object/dependency management.

3.1 Introduction

In object-oriented application development dependency injection is a well-known creational design pattern. The term is credited to Martin Fowler for his article "Inversion of Control Containers and the Dependency Injection Pattern" [23] that popularized it.

Dependency injection containers are used by applications for instantiating and managing objects on their behalf. A significant limitation of these containers is that their operations, and the objects they instantiate, are confined only to the domain of a program that created them and survive only as long as that program continues to run. In response to this limitation

we conceive a new way for various applications to interact by distributed sharing of objects over multiple interconnected computers to satisfy their dependencies.

3.2 Distributed Dependency Injection

As the name implies, distributed dependency injection encompasses the following concepts: the notion of distribution and the notion of dependency injection, and the key component, where the two concepts converge, is an object.

In dependency injection, an object explicitly declares and requires its dependencies to be provided, rather than creating or finding them on its own. This principle can be extended (distributed) to a group of interconnected applications as follows: the distributed dependency injection allows an object to explicitly declare and require its dependencies to be provided across application/domain boundaries.

> Distributed dependency injection extends the dependency injection principles of stating dependencies, and having them provided from, across domains.

3.3 Distributed Dependency Injection Container

The primary task of a distributed dependency injection container is that of an injector (subsection 2.2.1) that takes care of assembling and managing objects on behalf of the application, including methods to register type mappings and object instances, and resolve objects (injecting them with dependent objects in the process).

The distributed dependency injection container acts as an extension to the usual container mechanisms. Type registration information, in the form

of type mappings and object instances, is shared by multiple applications.

Used by the applications spread over a group of interconnected nodes to share objects amongst them: the distributed dependency injection containers connect between each other on behalf of the applications in order to exchange the type registration information and the objects that they create and manage.

Figure 3.1 shows four applications using a distributed dependency injection container that spreads across all four. From the container, the application logic acquires logical object references, as shown in empty circles. The means to resolve the reference are the same as for objects in Figure 2.2 where the container is used exclusively by a single application. The reference is used by the application logic as if it were of a local object. The logical objects, shown in filled circles, which remain managed by the container, retain their state and references to other logical objects. With the distributed dependency injection container the objects are distributed through replication, shown in empty pentagon shapes. This makes objects (their state and references to other objects) from each application available to all the applications participating in the object exchange.

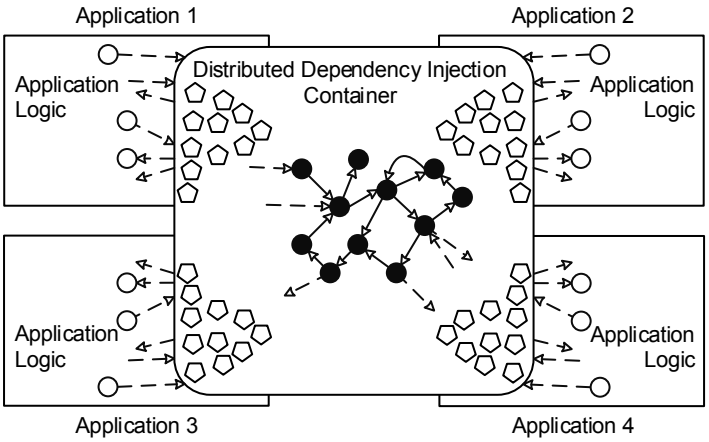

Figure 3.1: **Distributed dependency injection container.**

The distributed dependency injection container assembles objects

> by providing their dependencies, or collaborators, from across domains and manages objects in the way that allows them to be shared among various applications running on multiple interconnected computers.

3.4 Use Cases

We identify three primary uses for a distributed dependency injection:

The first is involved with the decision making process. Distributed dependency injection could be considered as an approach to building a new distributed system.

Two other cases consider an existing system/application. With the first of these two let us assume an existing distributed system, built using the object-oriented (remote-object) paradigm. For remote-object model we have exposed (section 2.3) a single point of failure. The added value of introducing a distributed dependency injection within the context of a distributed dependency injection container into such a system would be the high availability and fault tolerance along with good design guidelines per dependency injection principle. From the other perspective, an existing application, built using a dependency injection framework could reach the limitations of a single process or have other needs for distribution. In this case, substituting a regular dependency injection container with a distributed one is an easy task — much simpler than providing for all the remaining distribution requirements.

Let us take a look at the system model that supports these use cases.

3.5 System Model

This section presents our model of the *distributed dependency injection container system*.

Various applications using the distributed dependency injection container are connected by a network. The containers are allowed to dynamically join

and leave the group of connected distributed injection containers; but are hosted by nodes that are known in advance, e.g. manually configured.

Each container employs the distributed object replication middleware that is the topic of the next section 3.6. When requested, the container resolves a logical object reference that represents a group of replicated object instances. Replication is fully transparent and hidden behind the logical object's interface.

The task of the middleware is to replicate objects among the connected containers. For replication we assume up to about 30 nodes. For objects that are controlled by the middleware full replication is used. In full replication all objects are replicated across all nodes; the whole object state is replicated. We employ a passive replication model. In passive replication [13, 29] requests are only processed by one primary copy. Updates are then propagated to the secondary copies. However, this is relaxed for read-only operations that can be served by any secondary copy. Synchronous update propagation is used; that is a primary copy must propagate any updates immediately, before the result of the operation that has caused the update is returned to the client. We use an operation transfer approach to propagate updates [44].

We make use of the primary per partition replication protocol [10] to allow operations in all partitions to continue. If a primary copy of an object is not reachable, the protocol promotes a secondary copy to a temporary primary copy. The protocol also includes a reconciliation protocol that restores consistency when partitions are merged. Conflicts that occur when different replica object instances of the same object are written to in different partitions are resolved.

A partially synchronous system model is used. In this model clocks are not synchronized, but message time can be bound [15].

A node can experience a certain failure behavior as a whole ("pause-crash model" [16]) and as connected to other nodes via communication links that can fail as well ("link failure model" [45]). As we cannot distinguish between a failed node and an isolated node [21] until recovery time, every failure is

treated as partitioning.

3.6 Distributed Object Replication Middleware

In distributed systems, middleware is a software abstraction layer that spans over multiple computers and offers each application the same interface (Figure 3.2). Depending on the purpose and/or the level of abstraction, middleware is used to hide complexities of, for example, (i) network communication or (ii) distribution of objects. The first example is of an earlier middleware, based on the client-server remote procedure calls through sockets. The second example is an abstraction based on the object-oriented principles and is of particular interest to us.

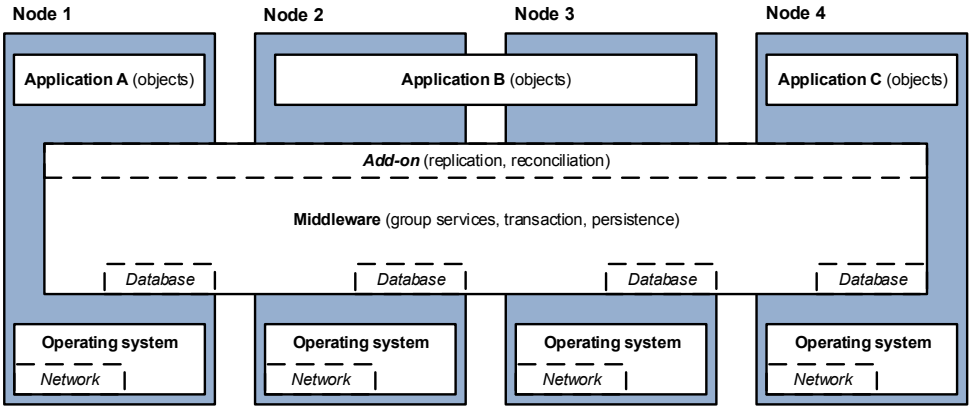

Figure 3.2: **A distributed system middleware.** The middleware layer extends over multiple machines, and offers each application the same interface.

Distributed objects middleware provides remote object invocation, which allows an object in a program running on one computer to invoke a method of an object in a program running on another computer. Its implementation hides the fact that messages are passed over a network in order to send the invocation request and its reply. Most object-based middleware solutions (like

off-the-shelf CORBA, Java/RMI and .NET Remoting) support this model in which object's state is in fact *not distributed*: it resides at a single server computer, only the interfaces implemented by the object are made available on other computers. The server, in this case, is the system's potential weakness as a single point of failure.

In our case the middleware is concerned with providing *the replication of shared data objects*. Replication accounts for redundancy and, if well managed, achieves fault tolerance, high availability and a high degree of reliability. We provide the replication capabilities as an add-on collection of support services to a generic middleware solution (Figure 3.2) of .NET Remoting.

Related to the previous section we provide a middleware solution for distributed dependency injection. The primary requirement for middleware is to provide object-sharing support for distributed containers. We identify the use of dependency injection with stateful objects. This fits the object-based approach for the development of distributed systems where objects are the resources being shared.

In principle this could be accomplished by a single server hosting all objects made available for remote invocations. In such deployment, if the server should fail, it would render the system, depending entirely on the server, useless.

> We contribute a fault tolerant middleware approach in which an object's state is replicated across multiple domains through a collection of support services.

3.6.1 Representation of Entities and their Identity

With regard to replication of objects we have to distinguish between them when referring to a logical object instance or one of its replica instances. An application is presented with a *logical object reference*. It represents an abstract view of the object it has created. *Replica reference* is used to refer to the actual replica object instance. A replica is a copy of an object's state on one of the system nodes.

Both logical object and object replicas are internally associated with the same *system-wide unique identifier* — a Globally Unique Identifier (GUID) is used. A *node* is identified by an IP address and a port number. A single object replica is then identified by the object identifier joined by the node identifier.

With respect to the group communication and membership service, a node will often be referred to as a *member* of a group, participating in the current membership view.

3.6.2 Architecture Overview

The components directly responsible for replication and fault-tolerance, the *invocation service*, the *replication service*, and the *group membership and communication services* are provided within the gray area of Figure 3.3. Other important components are the *naming service* for maintaining object identities and allowing for name to object bindings; the *activation service*, responsible for object instances and references; the transaction service for managing distributed transactions; and are not of immediate interest with respect to our solution.

The primary activity of the system is the continuous process of replication upon invocations. That is, after the objects are activated and their replica copies created and spread throughout the system.

Replication upon invocations is intended to assure the consistency between replicas. Consistency is achieved by propagating updates with each call initiated on the logical object: calls on the logical object are intercepted by the invocation service and diverted through the replication service. The call is primarily processed by one (primary) replica object instance. Updates are then communicated throughout the system and all (secondary) replica object instances of the logical object updated accordingly. The underlying communication service provides reliable communication.

Invariable consistency among all replica copies is needed to provide a fresh failover in case of failures. Fault detection is provided by the group member-

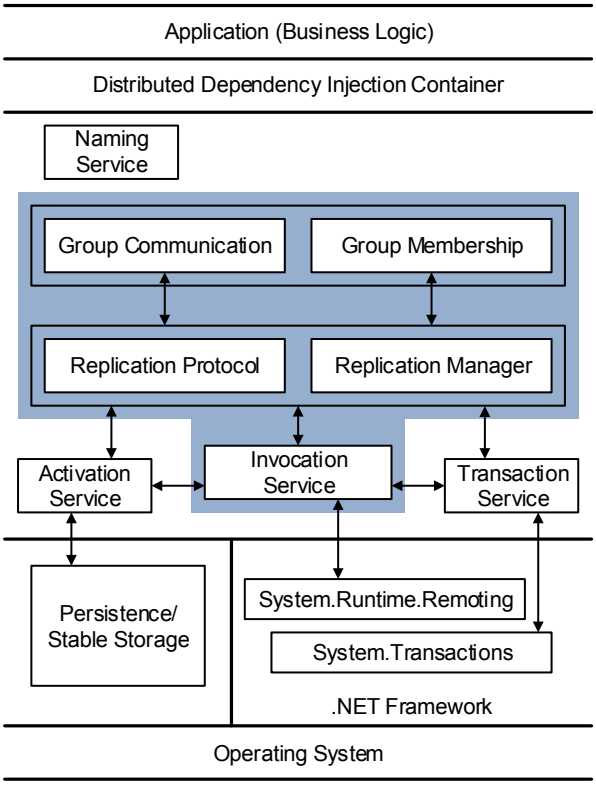

Figure 3.3: **System architecture.** The core components along with their interactions and interactions with the .NET Framework.

ship service. Degradation (node or link failure) of the system and any other membership changes are reported throughout the system. Reconfiguration of the system that is the result of failure is reconciled, i.e. consistent system view is established to enable the system to operate in degraded mode (subsection 3.6.3). Recovery from failures that result in reunification of replicas that were (temporarily) unreachable involves reconciliation that enables the system to resume operations in normal mode.

Replication process, consistency among replicas, detection of failure, operation in presence of failure and recovery from failure together account for fault tolerance.

The system view (subsection 3.6.4) is also continuously maintained ("in

sync") by all nodes in the system. As new objects are activated in the system, new replicas created, nodes join or leave, this information and events are synchronously exchanged between all nodes. During this process the nodes that accordingly incorporate the new information into their respective system views inessentially replicate the state of the replication manager, naming and activation service components.

The remainder of this section describes the components essential to replication and fault-tolerance, the three major system states, persistent system view and further activities of the system.

Invocation Service The invocation service provides the invocation logic used for invocation of methods within the system. It further provides the possibility to intercept object invocations (e.g. to perform middleware tasks) and transmits additional data with an object invocation (e.g. the object identifier).

Replication Service consists of a replication manager and a replication protocol The responsibility of the replication service is with the replicas.

The replication manager maintains track of all the replicas. It maintains a mapping between global object IDs and replica IDs with their location and role.

The replication protocol supplies the logic for maintaining the consistency among all the replicas of a single object. This includes the propagation of updates upon invocations and also preparation for reconciliation and the process of reconciliation upon recovery.

Group Membership and **Communication Services** The group membership service knows which nodes are reachable, i.e. not crashed and in the same partition. This information is needed for example by the replication manager, group communication, and the transaction manager.

Additionally, the group membership service ensures a consistent membership view within a single group (in our case within a partition). It monitors

membership changes caused by voluntary (join or leave) changes or due to failures (crashed or unreachable nodes). This also provides fault detection as changes are notified to the group members.

The group communication service provides reliable communication among group members. This is used for example by the replication protocol for propagation of updates.

3.6.3 Three Major System States

We consider the distributed system to be in one of three major system states:

The system is in normal mode when all nodes are reachable. During the degradation of the system into several network partitions we consider the system to operate in degraded mode, i.e. due to link or node failures. Activities performed after reunification or recovery from such a failure are summarized as reconciliation. This also covers multiple failures. After reconciliation, the system continues to operate under either normal or degraded mode. Normal mode is reached only after all failures are successfully repaired (full recovery).

The behavior of the system is modeled by the finite state machine (Figure 3.4).

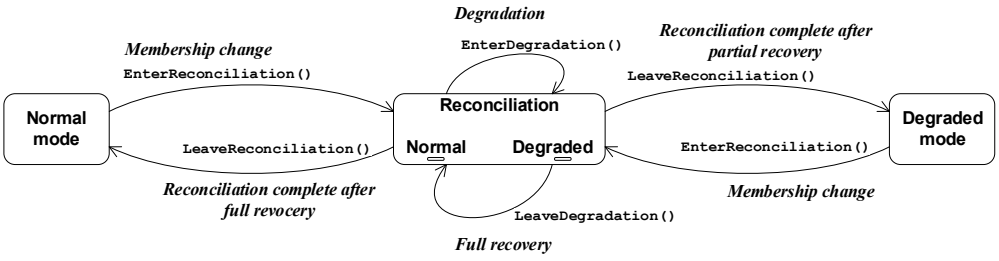

Figure 3.4: **System modes.**

- States reflect the status of the system.
  - State to reflect operation in normal mode.
  - State to reflect operation in degraded mode.

  - State to perform the reconciliation activities. Internally this is composed of two additional states for each operation mode.
    * Normal reconciliation state to perform activities after full recovery.
    * Degraded reconciliation state to perform activities after only partial recovery.
- Transitions between states.
  - A membership change when in normal or degraded mode signals transition to reconciliation. Depending on the nature of this change the reconciliation enters.
    * Normal reconciliation on full recovery.
    * Degraded reconciliation on only partial recovery.
  - Reconciliation after full recovery signals transition to normal mode.
  - Reconciliation after partial recovery signals transition to degraded mode.

3.6.4 Persistent System View

The data stored by the components, the naming service, the activation and the replication services constitutes a system view. The model is presented in Figure 3.5 and the data are persisted to a persistent storage.

A system view of a single replica is a subset of the full system view. A replica's system view of an object instance that is identified by an object identifier (GUID) consists of: (i) the single instance record, (ii) all its replica records and (iii) all its version records.

Instance entity stores object's identifier as a GUID, and its bindings to the assembly qualified name of object's type (class) and human readable name.

Replica entity stores properties about replica object instances managed by the replication manager.

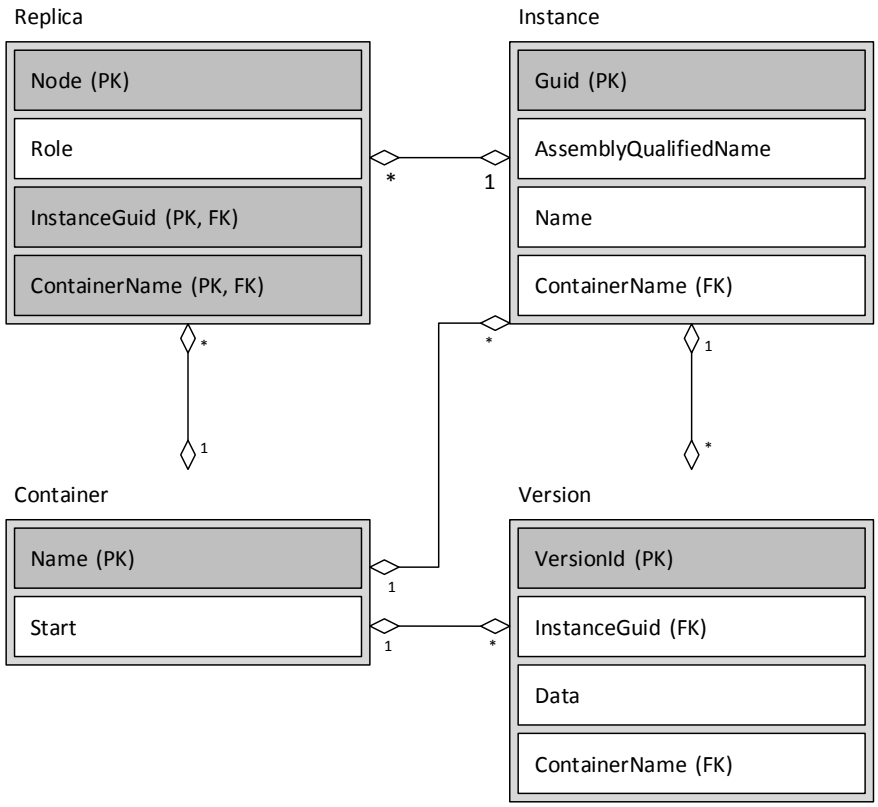

Figure 3.5: **Persistence Model.**

Version entity stores snapshots of a replica object instance's state taken at various points, for example, after each invocation on the replica object instance.

3.6.5 Activities of the System

This section describes the activities of the system in response to actions initiated by the client application, responses to group membership changes and the ongoing processes in the system to provide replication and fault tolerance.

Activation When the client application requests a reference to an object (it may do so by passing also a human readable convenience name) the object of the requested type is activated by the activation service. Objects are activated using the proxy design pattern.

Behind the abstract view of a proxy-object the following "installation" occurs: (i) For each new object in the system a new object identifier (GUID) is generated by the naming service. The identifier is internally stored by the naming service along with the actual type (class) of the object and name. (ii) Both the proxy-object reference and the actual object instance are stored and further managed (e.g. persisted) by the activation service. (iii) A snapshot of the actual object's state is stored into the version list. (iv) An invocation mechanism is set up by the invocation service as per the invocation pattern. This makes the object instance available for invocation by publishing it at a well-known URI (comprised from its identifier and the identifier of the hosting node). (v) By the replication manager the actual object's state is copied to multiple nodes in the system. (vi) Finally, the client application is presented with a proxy object reference.

For an already existing object, which was introduced into the system by another client application, a replica of that object has already been installed — by the replication process — on the node local to the client application. The activation process differs in that: (i) The object's type and name are already known to the naming service and an existing object identifier is resolved. (ii) The object reference and the actual instance for that object identifier are already known to the activation service. (iii) The invocation mechanism has already been set up by the invocation service. (iv) The client application is presented with a proxy-object reference resolved from the activation service.

The returning proxy object (logical object from 3.6.1) is able to intercept calls at the moment of invocation. The object identifier is embedded in the proxy object and is automatically known for each method invocation by the client application.

Replica Creation Replication is provided for each object in the system and the object's state is replicated across all nodes.

Upon the creation of a new object (activation process) a new object identifier (GUID) is generated by the naming service and stored (i.e. persisted) along with the actual type (class) of the object and its name. Also during the activation process an initial object snapshot is added to the version list.

Replication manager is employed to replicate the object's state throughout the system. A replica is first locally installed as a primary. After that a snapshot of the replica's system view (subsection 3.6.4) is propagated throughout the system using the group communication service.

Nodes that receive the propagated replica's system view extend their own system view with the received system view of the newly created replica. In this view the actual node on which the initial object was created is already set as the node hosting the primary replica. Further all remaining recipient nodes are added as nodes hosting secondary replicas as they all also install and host a replica of the object.

This results in a unified view of the partition upon creation of each new object: all nodes host replicas of all objects created within the partition, with the invocation mechanisms setup and with consistent information on (i) the object identification, type and name (ii) location of the primary and secondary copies and (iii) version list of all objects.

This means that subsequent activations of the object on any node will return a logical reference to the object replicated in the system. The system view is maintained consistent also on membership changes with the process of reconciliation on membership changes. The consistency between replicas of a single object is maintained by the update propagation process upon invocations.

Invocations When the application uses its logical object reference to invoke operations on the object this is intercepted by the invocation service local to the call initiator. The object identifier is retrieved from the proxy object.

Based on the object identifier, the local replication manager node supplies the location of the designated replica reference to execute the call (the primary copy).

The primary copy for the invoked object may reside on a different node. Hence, the information about the invocation is forwarded to the invocation service on that node. On the remote node the operation is placed on the actual replica object instance causing changes to its state. Afterwards the replication protocol is informed of performed operation to propagate the changes throughout the system (propagation of updates). Finally the control is transferred from the remote node back to the node local to, and the results of the invocation returned, to the call initiator.

For read-only invocations, which do not affect the object's state, the replication manager is asked to supply the location of any (secondary) copy. Also the update propagation is not necessary in such cases.

Propagation of Updates The process of propagating updates is initiated after each modifying call that is executed on the designated (the primary) replica object instance, to propagate updates to other (secondary) copies residing in the system.

Propagation of updates is initiated by the replication protocol component on the node where the primary replica copy resides. With the operation transfer approach to update propagation, the method call is along with parameters (including the object identifier) packed into a "communication" message. Included also are the results of the execution on the primary replica. The update propagation message is broadcast throughout the system using the group communication service.

The message is picked up and handled by the replication protocol component on the recipient nodes. Locally on each recipient node, the replica object instance is retrieved from the activation service and the "update" is applied. The results can be compared to the results of the execution on the primary replica instance (found in the update propagation message).

All nodes (including the sender) create a snapshot of the replica object instance involved in the invocation. The snapshot is appended to the version list.

Reconciliations of Membership Changes We assume that when a node is notified of a membership change it is presented with the current group view. The nodes listed are ordered and all nodes are presented with the same listing order. We use this feature to reach a *global consensus* among the multiple nodes in the partition. The delegation of responsibilities (i.e. which node does what) in any given situation and on all matters may be reached based solely on the ordered listing of nodes.

The activities of the system upon membership changes are coordinated by the replication protocol component on each node. At the occurrence of a membership change, the system preliminary enters reconciliation. Afterwards the change is examined for effects to the system and reconciled. Additionally, the system (on each node) keeps track of nodes that are holding the system in degraded mode and a list of those that gracefully left while in degraded mode; the two lists are also subjected to reconciliation.

Join This may be an introduction of a new node into the system or rejoining of a faulty node.

In order to comply with the full replication approach, the newly joined node has to install a copy of each object replicated in the system. By global consensus the first listed "old" node is selected to present the newly joined node with the system view. A system view message is broadcast with the newly joined node as an exclusive addressee.

Once the newly joined node receives the message it uses the view to install the replicas. All nodes in the system add the newly joined node to their view as a secondary host of replica copies of all objects in the system. This — for each node — concludes the reconciliation on "join".

The system view message conveys also the two lists of degradation nodes and list of nodes that gracefully left the system. Both lists are adopted by

the newly joined node and the system state is changed accordingly whether the system seems to be in degradation or normal mode.

In turns, rejoining of faulty nodes may lead to full system recovery. The appropriate transition — to normal mode — is signaled on the state machine when this is the case.

Joining of a new node is also the case when the first node enters the system. If this is the case, no further steps are required since such a system is consistent, hosting no replicas; the system is reconciled immediately, transition to normal mode is signaled.

Leave This membership change is the case when a node has concluded its processing and notifies on its intentions to exit, i.e. it is said to have left gracefully.

The node no longer hosts any replica copies. For some objects this may be the primary copy. In that case a new primary has to be appointed in the system.

The remaining nodes respond by excluding the leaving node's replicas from their system view. By global consensus the first listed "stayed" node is selected to act as the new primary for all objects. Also, other algorithms that take load-balancing into account, round-robin for example, could be used for new primary replica selection.

If the system is in the degraded mode the gracefully leaving node is recorded in the corresponding list of nodes. Reconciliation is concluded and a transition to either normal mode or back to degraded mode is signaled, depending on whether there are any degradation nodes.

Disconnect A "disconnect" membership change is reported about a node that is terminated forcefully, i.e. due to failure.

This is handled similarly to "leave" as downed replicas have to be removed from the system view. Additionally, such a failure puts the system in degraded mode. A node that failed is said to be holding the system in degradation —

until recovery. Till then the node is stored in the corresponding list of nodes. This list is also updated as nodes are subjected to network changes.

Network "Network" membership changes reflect the changes on the network layer, that result in (i) separation and/or (ii) reunification of nodes.

The nodes respond by removing the replicas residing on nodes that become separated (similar to leave). If there are no new nodes reported by the membership change no further reconciliation is needed.

Any nodes that possibly joined the view have to be included in the system views by reconciliation. In preparation of further reconciliation each node publishes its system view:

- Hosted replicas, their locations, roles and their version lists.
- The list of nodes holding the system in degraded mode.
- The list of nodes that gracefully left while in degraded mode.

After receiving system view messages from all nodes, each node calls the reconciliation support to produce a resulting consistent system view. Also the list of nodes holding the system in degraded mode and the list of nodes that have gracefully left the system are reconciled by the reconciliation support.

After reconciliation the resulting system view is installed and a transition to either normal mode or back to degraded mode is signaled, depending on whether there are any remaining degradation nodes.

Reconciliation Reconciliation aims at creating a resulting consistent system view from a number of input system views. Input system views may differ in updates on replicas. These inconsistencies are detected and solved using the associated reconciliation strategy. In the resulting system view consistency is assured.

As input to the reconciliation process the following is given: (i) system views collected from all nodes and node listings of (ii) all nodes that form the reunified partition (iii) nodes that were in the same partition with the local node and (iv) nodes that are newly joined to the reunified partition from the local node's point of view, (v) and also the two lists of nodes holding

the system in degraded mode and the nodes that left the system gracefully during degradation.

The reconciliation strategy is required to produce a consistent system view and also to reconcile the nodes that are holding the system in degraded mode and the nodes that have left the system gracefully during degradation.

Chapter 4

Prototype Implementation, Validation and Evaluation

"A visual syntax? Can you show me an example?"
"Coming right up."

—Ted Chiang, Story of Your Life

In our approach for various applications to interact by sharing objects over interconnected domains we offer an implementation of a distributed dependency injection container that interacts with a middleware solution for high availability, fault tolerance and a high degree of reliability.

The distributed dependency injection container in this case represents a clean separation of concern between code related to operation in a middleware framework and code associated with the application logic.

Our implementation is based on .NET framework[1] using Windows[2] as the underlying operating system. The C#[3] programming language has been chosen for implementation[4].

This chapter contains code.

[1].NET Framework version 4.5.
[2]Windows Vista and later are supported by the .NET Framework version 4.5.
[3]C# language specification version 5.0.
[4]For development Visual Studio 2013 and 2015 were used on Windows 8.1, .NET 4.5 as the target framework.

4.1 Introduction

When using the off-the-shelf middleware, programmers are still exposed to many details associated with the middleware architecture. Having to explicitly (register channels and services, and activate objects by URI) deal with non-functional concerns related to issues such as distribution of objects.

Like, when using .NET Remoting to write a program that uses remote objects. This involves the steps from Figure 4.1 in writing both the client and the server program [14]. The server implements the remote objects and the client consumes the services offered by the remote objects on the server. Multiple client programs spread across the network all communicate with the server program hosted on a single node.

The example describes the most verbose usage, and serves its purpose of demonstrating the complexities of dealing with the distributed system services. There is an obvious need for further simplifications.

4.2 Distributed Dependency Injection Container

The distributed dependency injection container is implemented as an extension to the Unity[5] container. The object sharing capabilities are provided by a distributed object replication middleware (section 4.3).

A *builder strategy* is called during the container's activities as it builds up an object instance. A custom strategy is added at the stage where the container has completed its work creating and initializing the instance. The control of the instance is handed to the replication middleware. In its place a proxy-object instance is provided. This proxy object is returned to the calling application.

A *lifetime manager* caches a reference to the proxy object in case of further resolutions.

[5]The Unity Application Block (Unity) version 3.5.

(I) Building the server involves the following steps:

1. Add a reference to the `System.Runtime.Remoting.dll` assembly.
2. Implement a class for the remote object by deriving it from `MarshalByRefObject`.
3. Choose one of the provided channel implementations (TCP or HTTP), and register it using the `ChannelServices.RegisterChannel` method.
4. Register the class as a well-known object using the `RemotingConfiguration.RegisterWellKnownServiceType` method.
5. Keep the server alive waiting for client requests.

(II) Building the client involves the following steps:

1. Identify the remote server object. This involves acquiring the information on the following, to form the remote objects URL:
   - The name of the machine that is hosting the server application.
   - The type of channel the server is using to expose the object.
   - The port number where the server is listening for incoming requests.
   - The remote object's assigned URI.
2. Add a reference to the assembly containing the metadata for the remote type.
3. Add a reference to the `System.Runtime.Remoting.dll` assembly.
4. Register a channel object using the same channel type as the server.
5. Activate the remote object by calling the `Activator.GetObject` method and passing the appropriate URL, to retrieve a proxy to the remote object.
6. Cast the proxy to the correct type and start using it as if it were the actual object.

Figure 4.1: **Remoting with .NET.**

4.2.1 Building Distributed Applications

This section describes the use of a distributed dependency injection container in developing a distributed application. The steps in Figure 4.2 perform roughly the equivalence of the steps given for .NET Remoting in Figure 4.1.

The distributed dependency injection container includes a data storage. Storage capabilities are enabled through the Entity Framework[6] [5]. The underlying storage can be either a relational database or an in-memory storage.

[6]Entity Framework version 6.1.1.

1. Add a reference to a number of DeDiSys, Unity, Spread and various framework assemblies.
2. Implement a class for the object by deriving it from `MarshalByRefObject`.
3. Apply the `[Serializable]` attribute to class.
4. Instantiate and use the distributed dependency injection container to configure the type mappings.
5. Resolve the concrete object implementation from the container.
6. Start using it as if it were the actual object.

Figure 4.2: **Using distributed dependency injection container** for building distributed applications.

Using a relational database requires setting up a SQL[7] [17] server instance of a database. We prefer to avoid this by using an in-memory storage[8,9]. Use of ether a relation database or an in-memory storage must be configured in accompanied configuration file.

Running the distributed dependency injection container preliminarily requires Spread[10,11] [7] group communication toolkit configured and running. Configuration includes creating a configuration file listing IP addresses of all hosts participating in message exchange.

Listing 4.1 shows and existing interface and a class implementation, that is further made replicable — by deriving the class from `MarshalByRefObject` and applying the `[Serializable]` attribute. Additionally `[ReadMethod]` attribute is applied to a method that is known not to cause any changes to the object's state. The interface and the replicable object class are further involved in building a distributed application.

In Listing 4.2, a distributed dependency injection container is first con-

[7]Microsoft SQL Server 2016.
[8]Effort (Effort.EF6) version 1.1.5 of a lightweight in-process main memory database provider, `https://github.com/tamasflamich/effort/`
[9]Dependent on NMemory version 1.1.0 of a lightweight non-persistent in-memory relational database engine, `https://github.com/tamasflamich/nmemory/`
[10]Spread Toolkit version 3.17.3, `http://www.spread.org/`
[11]A custom build of Spread was used, that included a speed-up in response time on detection of network changes.

```
1  /// <summary>
2  /// An existing interface.</summary>
3  interface IExistingInterface
4  {
5      void Read();
6      void Write();
7  }
8
9  /// <summary>
10 /// An existing object implementation.</summary>
11 class ExistingObject : IExistingInterface
12 {
13     public void Read() { /* implementation */ }
14     public void Write() { /* implementation */ }
15 }
16
17 /// <summary>
18 /// An existing object implementation made replicable.</summary>
19 [Serializable]
20 class ReplicatedObject : MarshalByRefObject, IExistingInterface
21 {
22     [ReadMethod]
23     public void Read() { /* implementation */ }
24     public void Write() { /* implementation */ }
25 }
```

Listing 4.1: **An existing object implementation made replicable.**

```
1  class Application
2  {
3      static void Main()
4      {
5          using (var dc = new DeDiSysContainer())
6          {
7              dc.RegisterType<
8                  IGroupCommunicationService,
9                  SpreadGroupCommunicationService>(
10                 new InjectionConstructor());
11
12             dc.RegisterType<
13                 IExistingInterface,
14                 ReplicatedObject>(
15                 new DeDiSysLifetimeManager());
16
17             IExistingInterface obj = dc.Resolve<IExistingInterface>();
18
19             obj.Write();
20             obj.Read();
21         }
22     }
23 }
```

Listing 4.2: **A sample distributed application.**

figured (lines 7–10) to use Spread as the underlying implementation of a group communication and membership service. Then a type registration is made in lines 12–15, creating type mapping between an interface *IExistingInterface* and a concrete type named *ReplicableObject* that implements the interface. This specifies that when requested (in line 17) for an instance of the type *IExistingInterface* the container should return an instance of the *ReplicableObject*. An instance of a custom *DeDiSysLifetimeManager* class provided as the lifetime manager (line 15) indicates that the resolved instance of *ReplicableObject* class should be replicated between a group of connected distributed dependency injection containers.

4.3 Distributed Object Replication Middleware

The object replication middleware is requested by the distributed dependency injection container's builder strategy to provide replication of an object instance. For this the middleware offers the *ProvideReplication* method presented in Listing 4.3, that accepts the object *type*, *name* and the actual object *instance* constructed by the container. In return the middleware provides the proxy object that is to be returned to the client application by the distributed dependency injection container.

```
1 object ProvideReplication(Type type, string name, object instance);
```

Listing 4.3: **Distributed object replication middleware** method to provide replication of an object.

The object *type* is from the mapping created with the container (object *instance* is already of the concrete type introduced by the mapping). A human readable convenience *name* can be provided with such a mapping and an instance also resolved using the same name, as shown by Listing 4.4.

```
1 using (var dc = new DeDiSysContainer())
2 {
3     dc.RegisterType<
4         IExistingInterface,
5         ReplicatedObject>("objA",
6         new DeDiSysLifetimeManager());
7
8     IExistingInterface objA = dc.Resolve<IExistingInterface>("objA");
9 }
```

Listing 4.4: **Named type registration** between an interface and a concrete type.

4.3.1 Entities and their Identity

To start with, we provide implementation details on the two identifiers used throughout the system — namely object and node identifiers from the subsection 3.6.1 "Representation of Entities and their Identity".

Object identifier is a unique system-wide identifier represented by an *ObjectId* class. This is a wrapper class around a Globally Unique Identifier (GUID) that is used internally.

Node identifier is represented by a *NodeId* class. This class joins an IP address and a port number into a single identifier used to uniquely identify nodes.

A replica is identified by the object identifier joined by the node identifier. The role of the replica is provided by the *P4ReplicaRoles* enumeration type that enables the choice between *Primary* and *Secondary* roles.

4.3.2 Operation Modes and System States

The behavior of the system (subsection 3.6.3 "Three Major System States") is implemented in the *SystemMode* class. An instance of this class is shared by all middleware components to acquire information on the state of the system.

The transitions (listed below) between states are coordinated by the replication protocol as membership changes occur in the system and as outcomes of reconciliations of these changes.

`EnterDegradation`	Signals transition to degradation.
`EnterReconciliation`	Signals transition to reconciliation.
`LeaveDegradation`	Signals transition from degradation.
`LeaveReconciliation`	Signals transition from reconciliation.

4.3.3 .NET Remoting Extension

We extend the .NET framework by injecting custom proxies on both sides of invocation. Their main purpose is to intercept method calls originating from the logical object and then again before any call is placed on the replica object instance.

At the client side this gives us the opportunity to intercept, control and customize subsequent remoting behavior. At the server side we alter the default remoting behavior by publishing instead a proxy to control the actual replica object instance. This again gives us the opportunity to intercept, control and customize. On the server side we control the actual placing of the method call on the object instance. By gaining full control we are automatically provided with the ability to intercept also the invocation results after the execution on the server side and after the delivery back to the client side.

We provide our custom implementations of proxy objects by deriving our classes from `RealProxy`. This conveniently redirects all invocations through the proxies' `Invoke` method passing an `IMessage` object. Client and server proxy classes are named `DeDiSysClientProxy` and `DeDiSysServerProxy`. A single `ObjectInstance` property is exposed by the server proxy object.

A client-server pair of proxies is created for each replica instance hosted by a node. Both proxy objects are associated with a specific object identifier. The client proxy is returned to the application for client invocations and the server proxy is made available at a well-known URI to act on server invocations. Local to the server node, the URI is comprised of the object identifier and the nodes' port number.

As nodes enter the system each is given a unique node identifier. A

channel is registered on that node and bound to its identifier (IP address and port). The channel is responsible for delivering the invocations across the remoting boundary. We utilize the TCP protocol and use the binary formatter as the most efficient and compact way of transporting the serialized message stream across .NET Remoting boundaries. This is achieved by using `BinaryFormatter` to serialize the message object and `TcpChannel` for transport.

With a proper setup of interception points we can entirely rely on the invocation logic as provided by the .NET Remoting infrastructure — method-call-as-message through proxies, sinks and channels.

4.3.4 Invocation Service

The invocation service plugs-in directly into the .NET Remoting proxy invocation mechanism.

The application invokes an object in the standard remoting way, using a logical object reference. The following are action steps performed by the invocation service once it intercepts the call message:

1. On the client side the invocation service calls on interceptors registered with it to find out the destination replica object instance. The interceptor provided by the replication manager is expected to provide the destination URL enclosed in the messages' URI property.
2. Next, all registered channels are checked to determine whether they accept the given URL. The channel that can service the URL returns the first sink in the remoting chain.
3. With the destination URL enclosed under the URI property the message is released into the message sink.
4. In the standard remoting way the message is routed to the destination node and further to the server proxy controlling the target replica object instance.
5. At the destination node the invocation service calls the interceptors registered for the server side interception. The activation service is

expected to provide an object instance.

6. The operation is executed by applying the message to the instance.
7. After the operation is performed, the invocation services on the server side call on the interceptor. An update propagation interceptor is expected to propagate the updates of the executed operation.
8. After the control is returned to the client side invocation service it also issues a call to the registered interceptors.

The invocation is concluded by returning the execution results to the application. Figure 4.3 depicts the invocation process for the client side, and Figure 4.4 for the server side.

Interface The invocation service provides methods (Listing 4.5) for manipulating the object instance and registering interceptors.

```
1 void ConnectInstance(ObjectId objectId);
2 void DisconnectInstance(ObjectId objectId);
3 object PublishInstance(Type type, ObjectId objectId);
4 void RegisterInterceptor(IInterceptor interceptor);
```

Listing 4.5: **Invocation service interface.**

The `PublishInstance` method provides the proxy object to be returned to the client application as the logical object instance. Internally the service makes the object replica instance available for remote invocation within the system.

The logical object instance can be temporarily "disconnected" from the service's invocation mechanism in case the client application wishes to dispose to object instance. Using the `DisconnectInstance` and `ConnectInstance` methods the proxy object that stands in for the logical object instance can be "disconnected" and then "hooked-up" again to the service's invocation mechanisms if the client application once more resolves that object instance from the distributed dependency container on another occasion. The object replica instance is not "disconnected" (along with the logical object instance) — it remains available for remote invocations within the system.

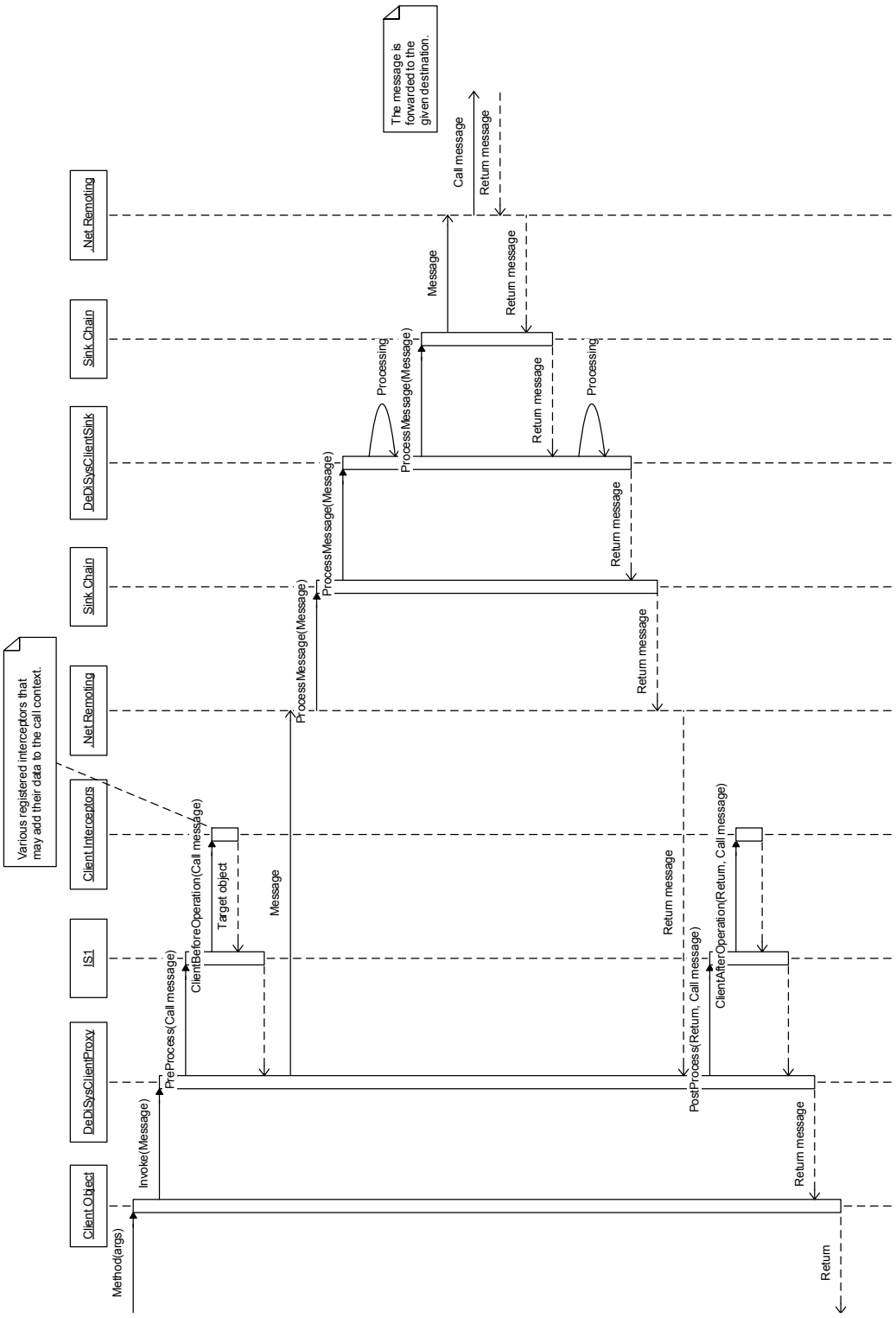

Figure 4.3: **Client-side invocation chain.**

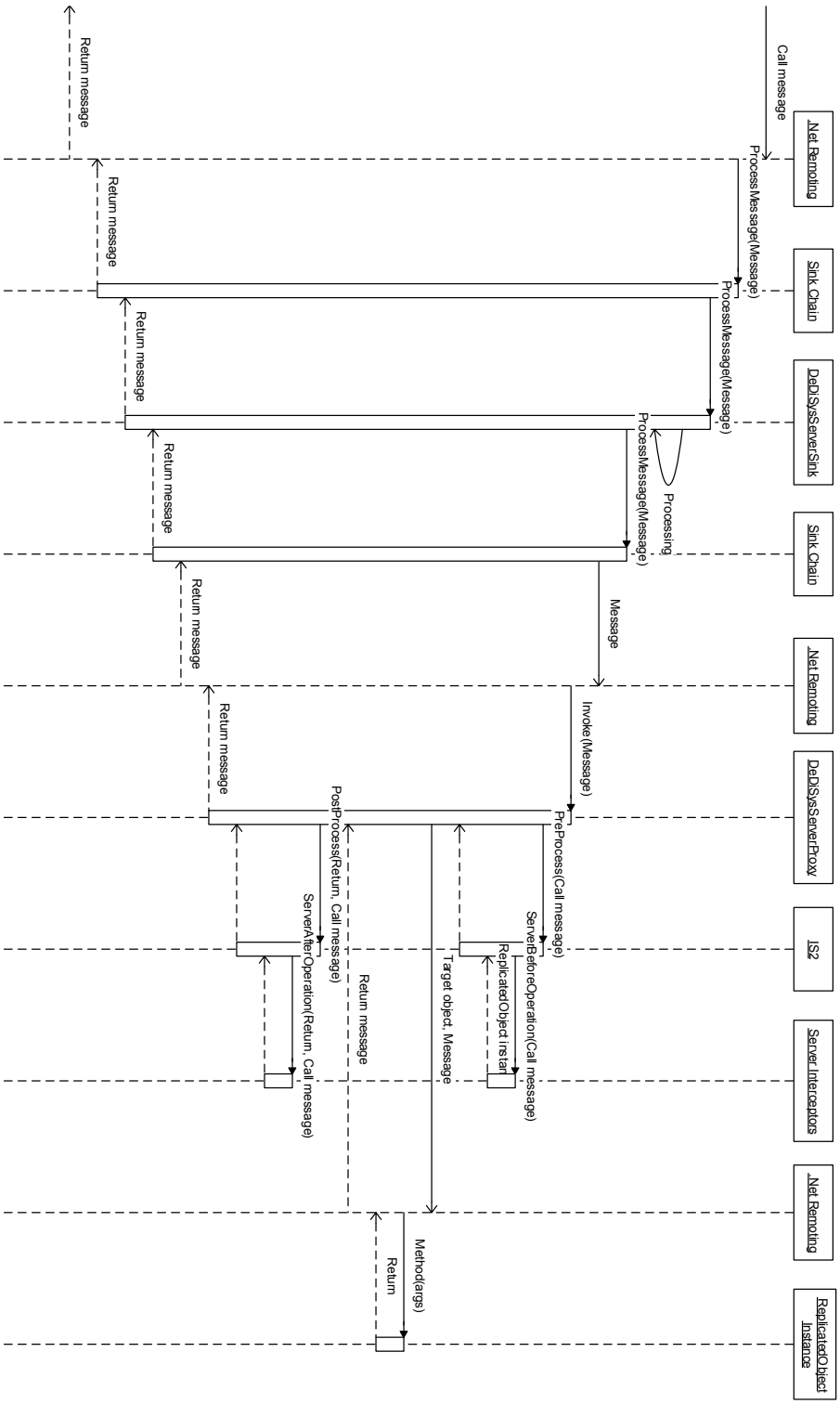

Figure 4.4: **Server-side invocation chain.**

`ConnectInstance` Connect the instance back to the service's invocation mechanism.

`DisconnectInstance` Temporarily disconnect the instance.

`PublishInstance` Publish the object replica instance and return the logical object instance.

`RegisterInterceptor` Interceptor registration.

Interceptors There are four types of interceptors: both client- and server-side interceptors for before and after operation interception. We define a base interface and derive interfaces for each of the interceptor types.

`IInterceptor` Base interceptor interface.

- `Priority` Priority of an interceptor within the group of interceptors (grouped by their type) and unique within that group.

`IClientBeforeOperation` Implemented by all interceptors for client-side interception before the operation is executed.

- `ClientBeforeOperation` Entry point for an interceptor to apply its own processing.

`IClientAfterOperation` Implemented by all interceptors for client-side interception after the operation is executed.

- `ClientAfterOperation` Entry point for an interceptor to apply its own processing.

`IServerBeforeOperation` Implemented by all interceptors for server-side interception before the operation is executed.

- `ServerBeforeOperation` Entry point for an interceptor to apply its own processing.

`IServerAfterOperation` Implemented by all interceptors for server-side interception after the operation is executed.

- `ServerAfterOperation` Entry point for an interceptor to apply its own processing.

Middleware components providing interceptors are required to assign them their designated priority and implement the entry point (Listing 4.6)

depending on the implementing type.

```
1  void ClientBeforeOperation(DeDiSysClientProxy callInitiator,
2                             IMethodCallMessage mcm);
3
4  void ClientAfterOperation(DeDiSysClientProxy callInitiator,
5                            IMethodReturnMessage mrm,
6                            IMethodCallMessage mcm);
7
8  void ServerBeforeOperation(DeDiSysServerProxy callInitiator,
9                             IMethodCallMessage mcm);
10
11 void ServerAfterOperation(DeDiSysServerProxy callInitiator,
12                            IMethodReturnMessage mrm,
13                            IMethodCallMessage mcm);
```

Listing 4.6: **Interceptor entry points.**

The four entry methods take the `IMethodCallMessage` object (representation of the original method call) as an argument. The message is provided to the interceptors and allows them to extract information regarding the object and the invoked method. The object identifier itself is embedded into the message by the invocation service. After-operation interceptors are additionally supplied with the `IMethodReturnMessage` object carrying the results of execution.

The registration process arranges interceptors into groups based on their type. Within a group of a specific type no two interceptors can have a same priority. The priority specifies the order in which the interceptors are called upon to provide their own processing.

4.3.5 Naming Service

The naming service is responsible for maintaining object identities. For each new object in the system it generates a new object identifier. This is achieved by using the `Bind` method to generate a new object identifier and associate (bind) it with the given type and name. The object identifier can be later resolved for the given type and name using the `Resolve` method (Listing 4.7).

`Bind` Generate a new object identifier to associate with the given type and name.

```
1 ObjectId Bind(Type type, string name);
2 ObjectId Resolve(Type type, string name);
```

Listing 4.7: **Naming service interface.**

Resolve Resolve (if existing) an object identifier for the given type and name.

The naming service is actually primarily queried to check whether the type and name are already known to the service. Only if not resolved successfully a new binding of such type is created with generation of a new object identifier. This is the case when replication of a new object is requested from the system. When a new object is in question, the instance is activated by the activation service, published by the invocation service and replication initiated by the replication manager.

4.3.6 Activation Service

The role of the activation service is to store and provide the actual replica object instances and logical object references.

In our case the activation service does not create a new object instance. An instance of an object is constructed by the container and passed to middleware to provide replication. This instance is along with its (object) identifier provided to the activation service for registration using the *RegisterInstance* method. The instance can be later retrieved using the *GetInstance* method (Listing 4.8).

Logical object reference, i.e. the proxy object is, after it is created by the invocation service, also registered with the activation service in a similar matter as the instance.

The activation service may be called to activate an instance of an object using the *ActivateInstance* method. The service responds by loading the latest version of the objects state from the system view.

```
1 MarshalByRefObject ActivateInstance(ObjectId objectId);
2 MarshalByRefObject GetInstance(ObjectId objectId);
3 object GetReference(ObjectId objectId);
4 void RegisterInstance(MarshalByRefObject instance, ObjectId objectId);
5 void RegisterReference(object reference, ObjectId objectId);
```

Listing 4.8: **Activation service interface.**

ActivateInstance Activate the instance associated with the specified identifier.

GetInstance Gets the object instance associated with the specified identifier.

GetReference Gets the object reference associated with the specified identifier.

RegisterInstance Registers the specified object instance with the service under the specified identifier.

RegisterReference Registers the specified object reference with the service under the specified identifier.

Provided Interceptors The activation service provides a server-side before operation interceptor that sets the *ObjectInstance* property on the *DeDiSysServerProxy* object.

4.3.7 Replication Manager

For managing replicas the replication manager provides the interface presented in Listing 4.9.

```
1 void AddReplica(ObjectId objectId, P4ReplicaRoles role, NodeId node);
2 void CreateReplica(ObjectId objectId);
3 void ExcludeReplicas(NodeId[] excludeNodes, NodeId newPrimary);
4 NodeId GetPrimary(ObjectId objectId);
5 NodeId GetReadCopy(ObjectId objectId);
6 Replica GetReplica(ObjectId objectId);
```

Listing 4.9: **Replication manager's interface.**

`AddReplica` Note that a replica copy of the specified identity and role is hosted by the specified node.

`CreateReplica` Creates a replica of the specified object instance on a local node. Suitable role is assigned to the replica as determined by the replication manager.

`ExcludeReplicas` Excludes all replicas hosted on specified nodes and if needed the new primary specifies the node where the new primary copy of the object is located.

`GetPrimary` For the specified object identity this returns the node hosting the primary replica.

`GetReadCopy` For the specified object identity this returns a node hosting a replica that may be used only for reading.

`GetReplica` Retrieves the replica with the specified identity.

The `ExcludeReplicas` method need further elaboration. The method is called when one or more nodes leave the group (either voluntary, due to fault or due to partition on the network level). Upon receiving the method call the replication manager is expected to update the system view accordingly. The nodes that are set to be excluded may either have hosted secondary or primary copies. The nodes that hosted a secondary can be simply removed from the system view. On the other hand, for each node that hosted a primary copy and set to be removed the provided new primary is promoted to that role.

Provided Interceptors Storing the information on the replica locations and their respective roles, the replication manager is also responsible to intervene when the application makes a reference to one of its logical objects. Behind the abstract view of this logical object there is a group of replica object instances. The replication manager is responsible to select the designate replica object instance to primarily process the method call.

The replication manager provides a client-side, before-operation interceptor for this purpose. The entry point for this interceptor type is the

`ClientBeforeOperation` method, that provides the method call message. The object identifier may be extracted from the method call message and the location of the target replica for that identifier is selected: Generally this is the primary copy that is requested from the replication manager using the `GetPrimary` method. However, read-only operations can also be serviced by secondary copies. Methods with read-only behavior are marked with the `ReadMethod` attribute. Whether this attribute is applied to the method can be determined by examining the method call message. When this it the case, `GetReadCopy` is called instead. Both `GetPrimary` and `GetReadCopy` return the target node identifier hosting the replica object instance with the designated role, either a primary or a secondary copy. Also note that `GetReadCopy` will return the node identifier hosting the primary copy if it is the only node in the system hosting the replica object instance.

From the selected target node identifier and the specified object identifier the TCP-specific destination URL is forged. The channel URI for the `TcpChannel` starts with the protocol specific `tcp://`, followed by the node's IP address and port number. To this, the object identifier and again the nodes' port number are appended.

Finally before concluding the interceptor processing the URL as embedded in the URI property of the method call message.

Further in the invocation chain the invocation service will provide the message sink that will forward the message to the given URL and the operation will indeed be executed on the target replica object instance.

Interactions with other Components The replication manager uses the group connection to notify of a replica creation by broadcasting the replica's system view in a message.

4.3.8 Replication Protocol

The replication protocol performs its specific update propagation and reconciliation activities.

There are no interfaces that are visible to other components. Update propagation is triggered through an interceptor which is registered with the invocation service. Interaction between replication protocol components in the system is performed using group communication messages. Reconciliation activities are triggered on membership changes that are reported by the group membership component. As these changes occur, the replication protocol coordinates the operation mode of the system. For reconciliation after nodes rejoin, the replication protocol relies on the reconciliation support.

Update Propagator The update propagator is the server-side after-operation interceptor provided by the replication protocol. This interceptor is placed in the invocation chain after the invocation of the operation on the replica object instance.

Once triggered, the update propagator receives both the method call and return messages. While the call message represents the operation that has already been executed on the replica object instance, the return message is the result of that execution. Both the method call and return messages are packed into a "communication" message that is broadcast using the group communication service.

The update propagator as an interceptor is called on all invocations. But the actual propagation of the update is skipped for read-only invocations.

Interactions with other Components When installing a system view, the replication protocol calls on the following components for each object instance contained in the system view being installed: (i) The activation service is called to activate the object instance (load the latest version of the object's state). (ii) Test whether the activation service can resolve the logical object reference for the object identifier. If the logical object reference is not known by the activation service, then (iii) the invocation service is called to install the object instance, which has been provided (activated) by the activation service. After installing the object instance, the invocation service returns (iv) the logical object reference that needs to be registered with the

activation service.

4.3.9 Group Membership and Communication Services

Group communication and membership services offer several interfaces that are for convenience listed in Table 4.1.

IGroupCommunicationService	Group communication service interface.
ICommunicationGroupFactory	Communication group factory interface.
ICommunicationGroup	Group communication channel interface.
IGroupConnection	Group connection interface.

Table 4.1: **Group communication and membership services' interfaces.**

The entry point to the group communication and group membership service is through the *IGroupCommunicationService* interface (Listing 4.10).

```
1 ICommunicationGroupFactory GetInstance(NodeId forNode);
```

Listing 4.10: **Group communication service interface.**

For the node instantiating the group communication service this interface provides the functionality for creating a factory instance of an (Listing 4.11) *ICommunicationGroupFactory* type. This is an interface for implementing factory classes that are used to create named groups through *GetGroup* method.

```
1 ICommunicationGroup GetGroup(string groupId);
2 ICommunicationGroup[] Groups { get; }
```

Listing 4.11: **Communication group factory interface.**

An instance of *ICommunicationGroup* type (Listing 4.12), retrieved from the factory, represents a named set of nodes.

Each node further requests a communication channel to the created group through *GetConnection*. This, an instance of *IGroupConnection* type

```
1 IGroupConnection GetConnection();
2 string GroupId { get; }
3 NodeId LocalNode { get; }
4 NodeId[] Members { get; }
5 event EventHandler<MembershipChangeEventArgs> OnMembershipChange;
```

Listing 4.12: **Group communication channel interface.**

(Listing 4.13), is used to set up a connection to the group and to communicate to group members. Connecting to the group using the *Connect* method also joins the node into the group.

```
1 void Connect();
2 bool Connected { get; }
3 void Disconnect();
4 ICommunicationGroup Group { get; }
5 NodeId LocalNode { get; }
6 void Multicast(Message message, bool sendSelf);
7 event EventHandler<MessageReceivedEventArgs> OnMessageReceived;
```

Listing 4.13: **Group connection interface.**

Group members communicate by multi-casting messages using the provided *Multicast* method.

Messages are sent to the group and received by each member over the (*IGroupConnection*) group communication channel. The recipient's group connection (*IGroupConnection*) further raises the notification to the registered event handlers of the *MessageReceivedEventArgs* class:

Message The message that has been received.
Sender The node that has sent the message.

Messages are represented by the *Message* class:

Object Object conveyed by the message.
Recipients List of nodes that also received the message.
Sender Node sending the message.
Type Message type.

The object contained in the message is (for communication) serialized into a data stream. For this, the object should be marked with the `[Serializable]` attribute.

The interface `ICommunicationGroup` also provides functionality for group membership changes and notifications. Notifications of membership change from the "old" to the "current" view are raised to the registered event handlers of the `MembershipChangeEventArgs` class:

`Cause` Cause of the membership change. Can be one of the following: `Join`, `Leave`, `Disconnect`, `Network`. Additionally `Transition` and `SelfLeave` causes are used by the service internally and are not reported to the handlers.

`Current` List of nodes that form the group view after the change.

`Left` List of nodes that are no longer reachable due to change — left the group view.

`New` List of nodes that are introduced into the group view by the change — are new.

`Old` List of nodes that formed the group view before the change.

`Stayed` List of nodes that were not subjected to membership change — from the local node's perception.

4.4 Application Scenario

For an application scenario we have developed a test bed application for validation and evaluation purposes and in order to be able to quickly and easily test the middleware as it was being developed. The demo is comprised of a test object that holds an integer value. The object is referred to as an integer object. Possible operations are addition, division, multiplication and subtraction. An operation is applied to the current value with a given integer. Also provided are methods for getting and setting the current object's value. The get operation is a non-modifying or read operation.

The interface `IIntegerOperations` for integer operations (Listing 4.14)

is implemented by a *IntegerClass* class. An integer application provides a command prompt user interface, allowing to select the designated operation and to input the desired amount. Although the application is very simple it is enough to test the functionality of the middleware. The application scenario is used for validation and evaluation in the following sections.

```
1  int Add(int amount, out int oldValue, out int newValue);
2
3  int Divide(int amount, out int oldValue, out int newValue);
4
5  int Multiply(int amount, out int oldValue, out int newValue);
6
7  int Subtract(int amount, out int oldValue, out int newValue);
8
9  int Set(int value, out int oldValue, out int newValue);
10
11 int Get();
```

Listing 4.14: **Integer operations interface.**

Another application scenario is presented in [30].

4.5 Validation

We provide testing and validation of the prototype implementation of a (distributed) system provided by the distributed dependency injection container. The system is tested (i) during normal operations under fault-free conditions and (ii) during degraded operations in presence of failures and (iii) in re-establishment of re-unified system after recovery from failures.

For our test system (Figure 4.5) we provide a network of four nodes. A test application using distributed dependency injection container and making use of distributed object replication middleware is executed on each node.

The network setup makes it possible to split it up into two partitions each containing two nodes. This is for fault simulation in order to test the behavior of the system in the presence of failures. In order to simulate network partitions a network cable between switches is removed. A computer is switched off in order to simulate a crash failure.

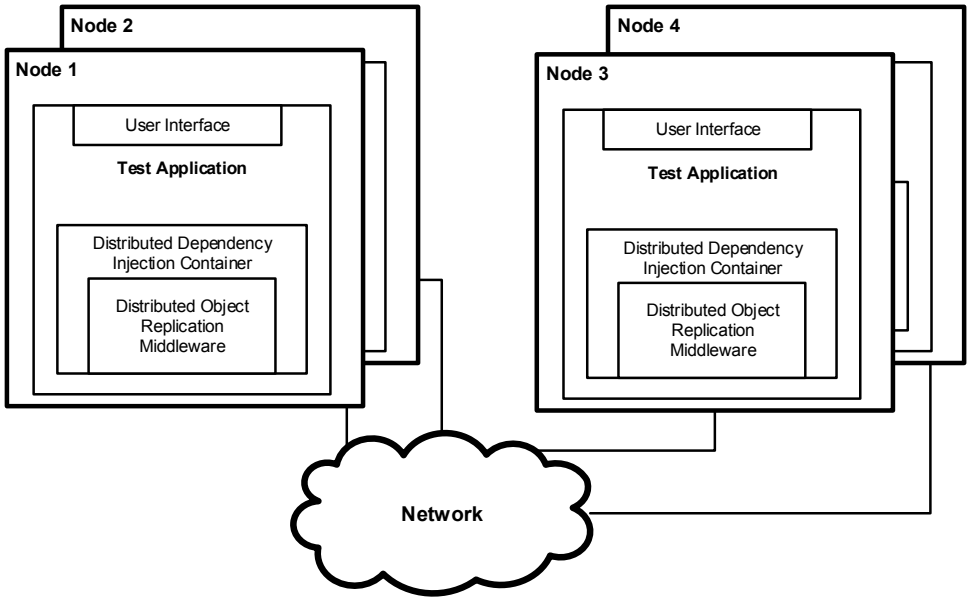

Figure 4.5: **Test environment.**

4.5.1 Test Cases

The test cases are devised to demonstrate the replication support, increased availability and the fault-tolerant capabilities of our distributed system. Faults are introduced in order to be detected by the system and to further demonstrate the increased availability as the system continues its normal operations. The recovery from failure is demonstrated by detection and reconciliation for re-establishment of the re-unified system.

Test Case 1: Replication During Normal Operations The main objective of this test case is to show the correct functioning of the replication support features under normal, fault-free conditions.

The test applications are started on a well-working network. The test application user interface is used to manipulate the test object.

The following behavior of the test system with respect to replication can be observed: (i) Primary copies of test objects are modified according to

the user inputs. (ii) Object modifications (updates) are propagated to the secondary copies throughout the test system.

Test Case 2: Detection of Degraded Situation The objective of this test case is to demonstrate the ability of the system to change its behavior from normal situation to degraded situation.

The test applications are started on a well-working network.

a) Upon simulation of a node — preliminary hosting the primary replica — crash the following behavior can be observed: (i) The group membership service on all remaining nodes initiates "view change" notifications informing about the drop out of one node. (ii) A new primary replica is nominated for the test object.
b) Upon simulation of a network split the following behavior can be observed: (i) The group membership service on all nodes initiates "view change" notifications informing about the "parting" of nodes on other network partitions. (ii) In each partition a new primary replica is agreed and nominated for the test object.

Test Case 3: Availability Enhancements in Degraded Situation This test case demonstrates that the system is able to enhance application availability in case of system degradation, such as network split or node failures.

The following behavior of the test system with respect to availability can be observed: (i) The degraded system from the previous test case ("Test Case 2: Detection of Degraded Situation") is able to operate. (ii) The test application user interface is used to even further manipulate the test object in both cases (a) with a single faulty node and in (b) both partitions separately.

Test Case 4: Replication Support in Degraded Situation This test case shows the ability of the replication support even in case of system degradation, such as network split or node failures.

Upon the continuous usage of the degraded system as described in Test Case 3: Availability Enhancements in Degraded Situation the following behavior of the test system with respect to replication can be observed: (i) Newly nominated primary copies of test objects are modified according to the user inputs. (ii) Object modifications (updates) are propagated to the secondary copies only among the operational nodes or, due to network partitioning, nodes "visible" within each of the two partitions.

Test Case 5: Detection of Re-unified System The main objective of this test case is to show the ability of the system to initiate a reconciliation activity upon detection of a re-unified system.

This test case is executed subsequent to the execution of the Test Case 4: Replication Support in Degraded Situation. There the test system is left in one of the degraded situations needed as a starting point: either the system is partitioned in two partial networks, or one node is switched off. The test system has evolved independent of the failed node or the other partition.

The test system is repaired:

a) For simulated node crash the appropriate computer is switched back on and the test application is started up again.
b) For simulated network split the network is reconnected again.

The following behavior can be observed: (i) The group membership service on all nodes initiates "view change" notifications informing about the "newly visible" nodes. (ii) The test system enters reconciliation.

Test Case 6: Reconciliation The main objective of this test case is to demonstrate the system's reconciliation functionality.

This test case is the continuation of the series of previous test cases with continuous operations in degraded mode and the re-unification of the system that lead up to the reconciliation of the system.

After reconciliation of the system the following behavior of the test system with regard to unified system can be observed: (i) All the nodes adopt the same unified view of the system, with a single primary copy for each object

and all the remaining nodes as secondary copies. (ii) Normal operations of the test system are restored.

4.6 Evaluation

The purpose of evaluation is to measure the invocation latencies in normal mode. This is to evaluate the overhead added by replication middleware compared to .NET Remoting and local invocations. The evaluation of remaining fault tolerance capabilities in the form of reconciliation process is not of our interest.

We compare the latencies of (i) a read operation and (ii) a write operations, and also include (iii) a combination of both in 10:1 ratio for different number of nodes.

This page intentionally left blank.

Chapter 5

Conclusions and Further Work

Somewhere ages and ages hence:
Two roads diverged in a wood, and I —
I took the one less traveled by,
And that has made all the difference.

—Robert Frost

Our introduction offered some insights into object-oriented application development and exposed the wide-spread presence of distributed systems. But Chapter 1 is primarily concerned with the motivation and the contributions of this thesis along with its structure.

We entirely dedicated Chapter 2 to the overview of the technology that is the basis for this thesis. We presented the object technology and the reasoning behind object management with the use of dependency injection within the context of dependency injection containers — dependency injection as a creational design pattern and principles, which are provided by a practical dependency injection container tool. Also related to objects, we rationalize their fault-tolerant distribution through replication with the assistance of middleware.

Our main contributions were presented in Chapter 3. Firstly, a distributed dependency injection that extends the dependency injection principles across domain boundaries. Secondly, a distributed dependency injection container that eludes the confinement of a single application domain and connects with

other containers on behalf of the applications. Thirdly, a distributed object replication middleware for truly distributed objects that accounts for fault tolerance, high availability and a high degree of reliability through replication and relaxed consistency.

In Chapter 4 we presented our fourth and final contribution — a prototype implementation of our approach. We made use of Microsoft's .NET technology: a regular Unity dependency injection container was extended to a distributed dependency injection container and we have built our replication capabilities as an extension to .NET Remoting. We have used the distributed dependency injection container in building a distributed application and submitted it to validation to confirm the increased availability of the system and the fault-tolerance gains by our replication middleware.

5.1 Conclusions

As our contribution we conceive a new concept for applications to interact by distributed sharing of objects over multiple *interconnected domains* to manage their *dependencies* in the form of a *distributed dependency injection*. For the conclusion we provide a summary of what we gain by introducing distribution into dependency injection.

Dependency injection is primarily used for object composition and continues to grow in popularity as a dependency management tool. When dependencies are well managed, the code remains flexible, robust, and reusable — pretty much "solid". Opposed to that, consider maintaining a poor code that is hard to change, fragile, and non-reusable — and otherwise a.k.a "smells". Injection is a means-to-an-end of dependency management that through loose coupling simplifies design, implementation, testing and maintenance and allows the focus on core functionality of objects and of applications at hand. Containers that provide the mechanism for assembling new object instances and managing existing object instances are intuitive to work with.

On the other hand, distribution is the only means for applications to

collaborate by sharing resources. In the case of objects, the purpose of the distributed system is to coordinate their use, with the assistance of middleware to make the distribution as transparent as possible.

When it comes to distributed systems for every promise there's a peril. Establishing communication between objects that run in different processes and do not reside on the same computer is a complex task that requires an in-depth knowledge and experience. In a distributed system parts can fail independently of each other. It is impossible to guarantee consistency, availability, and partition tolerance (CAP theorem). By choosing availability, we give up consistency. One has to embrace failure and provide availability.

Generic middleware, like Microsoft .NET Remoting, can not provide satisfactory solutions. (i) Advertised as a distributed object middleware it does not provide replication capabilities and lacks fault-tolerance. (ii) Programmers are exposed to many details associated with the architecture.

We provide a middleware solution for truly distributed objects in the form of high availability and a high degree of reliability through replication and relaxed consistency. This provides failover capability for continuous availability. And we are back to the dependency injection as this is packaged as an intuitive container.

When using the distributed dependency injection container the programmer does not directly interact with our middleware solution. The container provides a clean separation of concerns.

If dependency injection is used for resolving dependencies then distributed dependency injection enables resolving dependencies over the "wire". Distributed dependency injection containers connect over the "wire" to do the "wiring".

5.2 Further Work

The concept of distributed dependency injection that is proposed by this thesis has an exploiting potential and can be further promoted. One crucial

aspect that remains open is the actual formalization of the concept. As is the case of general dependency injection, or common programming: an object generally states its dependencies in terms of objects — or even more commonly interfaces — it requires by listing them in the class constructor (Listing 5.1).

```
1 SomeClass(IInterfaceOne one, IInterfaceTwo two, ClassThree three)
```

Listing 5.1: **Constructor dependencies.**

The formulation of the distributed dependency injection would provide the means to differentiate the dependencies that are distributed across application domains. Possible approaches could involve (i) the use of attributes or (ii) an additional keyword added to the language specification. The example in Listing 5.2 shows a *Distributed* attribute applied to the *first* dependency and the *second* dependency marked by the proposed *dis* keyword. The use of attributes being the preferred approach as more feasible.

```
1 [Distributed("first")]
2 SomeClass(IInterfaceOne first, dis IInterfaceTwo second, ClassThree three)
```

Listing 5.2: **Distributed constructor dependencies.**

Regarding our approach the following are some technical insights that can be exploited. One major prospect is to avoid .NET Remoting all together. The separation point being the interception by the invocation service that plugs into .NET Remoting infrastructure. As an alternative Unity container already provides interception capabilities for dynamically inserting code. This is a very similar to another, also feasible alternative, true AOP [34] approach. Other approaches include custom (although cumbersome) invocation API through proxy or a command design pattern [27]. Remote object invocations, avoiding the .NET Remoting infrastructure, could be achieved by message passing through group communication mechanisms.

Within the scope of .NET Remoting there remain also events and asynchronous invocations to be considered.

Object lifetime management can be further improved. Infrastructure of the .NET Remoting provides various management techniques for the remote objects (like object lifetime leases). The activation service can be extended to respect object lifetime leases and swapping of objects between permanent storage and memory, and to include even further memory management techniques. Lifetime management can be considered also within the scope of Unity container. Primarily to differentiate whether the objects are singleton or transient.

Reconciliation capabilities of the system can be further improved and exploited. To enable reconciliation on the operation level, the proposed model of the version list (containing snapshots of objects' states after each invocation) could be extended to include: (i) the method call along with parameters, (ii) results of the execution and (iii) the present group members. The proposed system is missing also a full support of a transaction manager with distributed commit capabilities.

Security could be provided as .NET Remoting does not provide any security features itself — authentication and access control can be implemented for clients by using custom channels. For secure group communication Spread can be enhanced with security services.

Ctrl-S[1]*, Ctrl+Alt+Delete*[2]

[1]Contributed by Dr. Andrej Brodnik. He suggests saving your work before pressing Ctrl+Alt+Delete.

[2]Key combination to restart your PC. "Have you tried restarting it?", is often "suggested" by the technical support, as a last resort in an attempt to correct an unknown error. I know I have offered it as a solution on several occasions. I am just about set to undergo a more life's changing restart, for the better I hope: "What is life like after thesis?"

This page intentionally left blank.

List of Figures

This page intentionally left blank.

List of Tables

This page intentionally left blank.

Listings

This page intentionally left blank.

Bibliography

[1] CORBA specification. http://www.corba.org. Accessed: 2016-07-05.

[2] Distributed component object model (DCOM) remote protocol specification. https://msdn.microsoft.com/library/cc201989.aspx. Accessed: 2016-07-05.

[3] DeDiSys project description, Annex I - description of work, 04.06.2004. DeDiSys Consortium. http://www.dedisys.org/.

[4] Dependency injection != using a DI container. http://www.loosecouplings.com/2011/01/dependency-injection-using-di-container.html. Accessed: 2016-07-05.

[5] Entity framework. http://www.asp.net/entity-framework. Accessed: 2016-08-15.

[6] Java remote method invocation (RMI) specification. https://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmiTOC.html. Accessed: 2016-07-05.

[7] Yair Amir, Claudiu Danilov, and Jonathan Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proc. of The Int. Conf. on Dependable Systems and Networks*, pages 327–336. IEEE, 2000.

[8] Tom Barnaby. *Distributed .NET Programming in C#*. Apress, Berkely, CA, USA, 2002.

[9] Dominic Betts, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. *Dependency Injection with Unity*. Microsoft Patterns & Practices, 1st edition, 2013.

[10] Stefan Beyer, Mari-Carmen Bañuls, Pablo Galdámez, Johannes Osrael, and Francesc D. Muñoz-Escoí. Increasing availability in a replicated partitionable distributed object system. In *Proc. of the 4th Int. Symp. on Parallel and Distr. Processing and Appl. (ISPA'06)*, volume 4330 of *LNCS*. Springer, 2006.

[11] Stefan Beyer, Francesc D. Muñoz-Escoí, and Pablo Galdámez. CORBA replication support for fault-tolerance in a partitionable distributed system. In *Workshop Proc. of the 17th Int. Conf. on Database and Expert Systems Applications*. IEEE CS, 2006.

[12] Timothy Budd. *An Introduction to Object-oriented Programming*. Addison Wesley, Redwood City, CA, USA, 1991.

[13] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In Sape Mullender, editor, *Distributed systems*, chapter 8, pages 199–216. ACM Press, Addison-Wesley, 2nd edition, 1993.

[14] David Conger. *Remoting with C# and .NET: Remote Objects for Distributed Applications*. Wiley, 2003.

[15] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[16] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2), 1991.

[17] C. J. Date and Hugh Darwen. *A guide to the SQL standard : a user's guide to the standard database language SQL.* Addison-Wesley, 4th edition, 1997.

[18] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[19] Michael D. Ekstrand and Michael Ludwig. Dependency injection with static analysis and context-aware policy. *Journal of Object Technology*, 15(1), 2016.

[20] Wolfgang Emmerich. *Engineering Distributed Objects.* John Wiley & Sons, Chichester, UK, 1st edition, 2000.

[21] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2), 1985.

[22] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, Boston, MA, USA, 2002.

[23] Martin Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, 2004. Accessed: 2016-08-04.

[24] Martin Fowler. CodeSmell. `http://martinfowler.com/bliki/CodeSmell.html`, 2006. Accessed: 2016-08-03.

[25] Lorenz Froihofer (ed.). FTNS system model. Technical Report D2.2.1, DeDiSys Consortium, 2005.

[26] Lorenz Froihofer (ed.). FTNS comparison. Technical Report D4.1.1, DeDiSys Consortium, 2007.

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[28] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[29] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[30] Igor Habjan, Klemen Zagar, and Matej Sekoranja. Fault-tolerant EPICS directory service. In *Proc. of the 6th Int'l Workshop on Personal Computers and Particle Accelerator Controls*. Jefferson Lab, 2006.

[31] Igor Habjan (ed.). Software prototype and refined design. Technical Report D3.3.2, DeDiSys Consortium, 2007.

[32] Abdelsalam A. Helal, Bharat K. Bhargava, and Abdelsalam A. Heddaya. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[33] Michi Henning. The rise and fall of CORBA. *Commun. ACM*, 51(8): 52–57, August 2008.

[34] Gregor Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[35] Donald E. Knuth. Computer programming as an art. *Commun. ACM*, 17(12):667–673, December 1974.

[36] Hubert Kuenig (ed.). Software prototype and refined design and validation report. Technical Report D3.2.2, DeDiSys Consortium, 2007.

[37] Matthew MacDonald. *Microsoft .NET Distributed Applications: Integrating XML Web Services and .NET Remoting*. Microsoft Press, Redmond, WA, USA, 2003.

[38] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River, NJ, USA, 2002.

[39] Sandi Metz. *Practical Object-Oriented Design in Ruby: An Agile Primer*. Addison-Wesley Professional, 1st edition, 2012.

[40] Johannes Osrael, Lorenz Froihofer, Karl M. Goeschka, Stefan Beyer, Pablo Galdámez, and Francesc Muñoz. A system architecture for enhanced availability of tightly coupled distributed systems. In *Proc. of the 1st Int. Conf. on Availability, Reliability, and Security*. IEEE CS, 2006.

[41] Johannes Osrael, Lorenz Froihofer, Georg Stoifl, Lucas Weigl, Klemen Zagar, Igor Habjan, and Karl M. Goeschka. Using replication to build highly available .NET applications. In *Workshop Proc. of the 17th Int. Conf. on Database and Expert Systems Applications*. IEEE CS, 2006.

[42] Dhanji R. Prasanna. *Dependency Injection*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009.

[43] Hans P. Reiser, Michael J. Danel, and Franz J. Hauck. A flexible replication framework for scalable and reliable .NET services. In *Proc. of the IADIS Int. Conf. on Applied Computing*, volume 1, pages 161–169, 2005.

[44] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), 2005.

[45] Fred B. Schneider. What good are models and what models are good? In *Distributed Systems*, chapter 2. ACM Press/Addison-Wesley Publishing Co., 2nd edition, 1993.

[46] Mark Seemann. Dependency injection is loose coupling. `http://blog.ploeh.dk/2010/04/07/DependencyInjectionisLooseCoupling/`, 2010. Accessed: 2016-08-03.

[47] Thomas Seidmann. Distributed shared memory using the .NET framework. In *Proc. of the 3rd IEEE/ACM Int. Symp. on Cluster Computing and the Grid*, pages 457–462, 2003.

[48] Alexander Szep, Robert Smeikal, Martin Jandl, and Karl M. Goeschka. Dependable distributed systems. In *Proc. Companion 19th ACM SIGPLAN Int'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2004.

[49] Klemen Zagar (ed.). Software prototype and refined design and validation report. Technical Report D3.4.2, DeDiSys Consortium, 2007.

Glossary

.NET Framework A collection of programming support for software development and execution environment from Microsoft.

abstraction[1] A technique in problem solving in which details are grouped into a single common concept. This concept can then be viewed as a single entity, and inessential information ignored.

application An application program (application for short or clipped to app) is a computer program designed to perform a group of coordinated functions, tasks, or activities for the benefit of the user. (See *program*)

application domain A mechanism (similar to a process in an operating system) used within the Common Language Infrastructure (CLI) to isolate executed software applications from one another so that they do not affect each other. Each application domain has its own virtual address space which scopes the resources for the application domain using that address space. (See *application*, *program*)

class An extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods). (See *object*)

client A role played in a single remote invocation by the entity issuing the request or invoking an operation upon the server. May refer to an

[1] According to Budd [12].

issuing computer, process or program. (See *server*, *computer*, *process*, *program*)

computer (See *host*, *node*, *client*, *server*)

dependency injection A creational design pattern for object dependency management. (See *design pattern*)

design pattern A general reusable solution to a commonly occurring problem within a given context in software design.

distributed system A collection of components on autonomous hosts that interact via middleware so that they appear as an integrated facility.

encapsulation The technique of hiding information within a structure, such as the hiding of instance data within a class.

failover An automatic switching to a redundant copy upon a failure of the previously active original.

host A computer that executes components that form part of a distributed system.

identifier The property of objects that distinguishes them from other objects. A unique identifier is assigned to the object when it is created. Objects generally retain their identity throughout their lifetime. An object has just one identity but there may be many references to the object. Two objects are considered to be the same object based on having identical properties, even if they are not actually the same physical instance (structural equivalence). (See *object*, *reference*)

instance A concrete occurrence of any object, existing usually during the runtime of a computer program. Formally, "instance" is synonymous with "object" as they are each a particular value (realization), and these may be called an instance object; "instance" emphasizes the distinct

identity of the object. The creation of an instance is called instantiation. (See *object*, *identifier*)

middleware A layer between network operating system and applications that aims to resolve heterogeneity and distribution.

network partition Separation a group of replicas into two or more subgroups (called partitions). This separation can be caused by a link failure. The replicas can communicate within the partition but communication across partitions is impossible. (See *reunification*)

node A single computer or host — anything that has an IP address. (See *computer*)

object Objects consist of state and related behavior and are conceptually similar to real-world objects. Software objects are a particular instance of a class, which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. Referenced by an identifier, objects have a notion of "this" or "self". Object's procedures can access and often modify the data fields of the object with which they are associated (stateful objects). An object is what actually runs in the computer. (See *abstraction*, *class*, *instance*, *stateful object*)

object-oriented programming An application development methodology based on the concept of objects. Programs are designed by making them out of objects that interact with one another.

process A program in execution. (See *program*, *client*, *server*)

program A collection of instructions that performs a specific task when executed by a computer. In the scope of this thesis programs are considered more down the lines of an application. (See *application*, *process*)

reference A handle that a client has on an object. Used to refer (access and assign) to an object with a specific identity. However, multiple references can refer to the same object. Typically, references are isomorphic to memory addresses. In distributed computing, the reference may also include an embedded specification of the network protocols used to locate and access the referenced object, the way information is encoded or serialized. Thus, it is a complete specification for how to construct a proxy that will subsequently engage in a peer-to-peer interaction, and through which the local computer may gain access to data that is replicated. In this sense, it serves the same purpose as an identifier or address in memory. (See *object*, *identifier*)

replica A copy that remains synchronized with its original. (See *replication*)

replication The process of creating a replica and keeping in up-to-date with the original. (See *replica*)

reunification The process of merging two or more partitions on the network layer, i.e. communication between the partitions is re-established. (See *network partition*, *reconciliation*)

reconciliation The process of detecting and solving inconsistencies and conflicts caused by updates on replicas in different partitions that is necessary after reunification. (See *reunification*)

server A role played in a single remote invocation by the entity accepting the request from a client in order to respond appropriately. May refer to an listening computer, process or program. (See *client*, *computer*, *process*, *program*)

stateful object An object that maintains, changes and alters internal state over time and during the invocation of its methods, and can exhibit (different) behavior that varies based upon what the object has already done. (See *object*)

Also as a part of the glossary we provide Slovene explanations of the terms relevant to this thesis, most notably, the keywords.

Slovar izrazov

vrivanje odvisnosti (angl. *dependency injection, DI*) Pri objektno usmerjenem programiranju zasnova, ki ureja odvisnost objekta; če potrebuje npr. objekt pri uvodni nastavitvi drug objekt, se odvisnost zabeleži na osrednjem mestu in ga objektu uvodne nastavitve ni treba proizvesti ali poiskat.

objektno usmerjeno programiranje (angl. *object-oriented programming*) Razvoj programja, pri katerem so koncepti realnega sveta predstavljeni s pomočjo objektov in razredov. Rezultat je program, sestavljen iz množic med seboj sodelujočih objektov z enolično identiteto, ograjenimi (enkapsuliranimi) lastnostmi in operacijami, ki medsebojno komunicirajo s pošiljanjem sporočil, in razredov, znotraj katerih so primerki povezani z dedovanjem, polimorfizmom in dinamičnim povezovanjem.

ogrodje .NET (angl. *.NET Framework*) Microsoftovo okolje za razvoj spletnih storitev in drugih programskih komponent.

porazdelitev (angl. *distribution*) Razporeditev na fizično različne lokacije, ki so medsebojno povezane ali odvisne.

porazdeljen sistem (angl. *distributed system*) Sistem, porazdeljen med več medsebojno povezanimi računalniki.

replicirati (angl. *to replicate*) Ustvariti kopijo, repliko česa ali kaj ponoviti.

replika (angl. *replica*) Vsaka od vernih kopij, torej kopija usklajena z izvirnikom (eksaktna kopija).

replikacija (angl. *replication*) Proces večkratnega kopiranja, podvojevanja (razmnoževanje ali reprodukcija) ter ohranjanja enakosti med replikami; zaradi izboljšanja dostopnosti in zanesljivosti ter odpornosti proti napakam.

vmesna plast (angl. *middleware*) Programska oprema, ki povezuje posamične programe, programske komponente.