A framework for developing .NET distributed systems

van Boven, Dennis

Award date:
2003

TU Eindhoven / Imtech ICT

MASTER'S THESIS

A Framework for Developing .NET Distributed Systems

by
Dennis van Boven

Eindhoven, July 2003

Acknowledgements

I wish to thank the following persons for their contribution to my graduation project:

dr. ir. G.J.P.M. Houben (TU/e) as being head of the examination board
dr. ir. A.T.M. Aerts (TU/e) as being member of the examination board
dr. N. Sidorova (TU/e) as being graduation supervisor from the TU/e
ir. M.J.A.M. van Gerwen (Imtech ICT) as being graduation supervisor from Imtech ICT

Further I would like to thank my Imtech colleagues for the nice period I had at Imtech and Imtech for providing a good graduation atmosphere and organizing several instructive excursions.

Finally I would like to express my appreciation to my family and my dear friend Ronald for their support and patience during my graduation period.

Summary

The graduation project for the Eindhoven University of Technology (TU/e) and the company "Imtech ICT iQuipware" focuses on developing a general framework for developing distributed systems with the recently introduced .NET Technology of Microsoft. Besides structuring the process of developing distributed systems, the goal of this framework is to divide the development of distributed systems into two independent parts: the development of system components and the development of the entire distributed system.

To develop a distributed system using this separation of concerns (SOC), experts in distributed systems would first design the distributed architecture (including component specification). Then the component programmers develop the components and/or of-the-shelf components are acquired. Finally the experts in distributed systems collect the necessary components and configure the properties of the distributed system. Main advantage of this approach is the introduction of a new independent specialization area that focuses on the distributed properties of a distributed system. Other advantages are the improved overview on the system, logging possibilities that make the maintenance of the system easier, error detection / fixing per separate development process, and the flexibility of easily adapting the system properties.

Based on a literature study the following properties of a distributed system are identified: dependability, load distribution, mutual exclusion, transactions, deadlock, communication, component activation / lifetime, localization, and security. The framework supporting the separation of concerns (called SOC-Framework) is an extra layer between components within a distributed system to handle these properties. The way in which the SOC-Framework handles the properties (i.e. the behavior of the entire system) can be configured with the use of XML-configuration files.

The SOC-Framework is implemented with the use of .NET Technology. The SOC-Framework uses .NET Remoting, the part of the .NET Technology that supports the development of distributed systems, on the lowest level to interact between components. The SOC-Framework is validated on a case study on a distributed application developed using the framework. This test application was developed according to the SOC-development process as described above. The test application revealed some problems when using the SOC-framework. These problems were fixed by extending the SOC-framework on several points.

The test application shows that it is technically feasible to establish a separation of concerns in the development of distributed systems, thereby concluding that the goal of the project is accomplished. An extra point of attention is the process of developing a distributed system using the SOC-Framework. The expert in distributed systems needs certain information about the components within the system for configuring several properties of the configuration. If the type of this component information can be standardized, all components providing information according to this standard can be used within distributed systems using the SOC-Framework. The idea of the SOC-Framework is a generic idea: therefore it is also possible to implement the SOC-Framework with other middleware than .NET Remoting (e.g. CORBA [Tan]).

Table of Contents

Glossary

.NET Remoting	Part of .NET Technology that supports the development of distributed systems
.NET Technology	Latest technology of Microsoft, providing a framework and a development environment
CAO	Client Activated Object
CFM	Configuration Manager
ClientProxy	Proxy for the client calling a remote component
Configuration Manager	Part of the outgoing manager that handles the configuration
Fault Tolerance Manager	Part of the outgoing manager that handles the execution of calls
FTM	Fault Tolerance Manager
Imtech	Imtech ICT Iquipware
Incoming Manager	Part of the SOC-framework that intercepts calls before they reach a remote component
LDM	Load Distribution Manager
Load Distribution Manager	Part of the outgoing manager that handles the load distribution
ManagerIn	Incoming Manager
ManagerOut	Outgoing Manager
MIExecuter	Part of ManagerIn that executes calls on the real object and manages the incoming properties
Outgoing Manager	Part of the SOC-framework that intercepts calls before they are sent to a remote component
PM	Proxy Manager
Proxy Manager	Part of the outgoing manager that stores and creates the remote references
RealObject	The real object on which a remote call is done
Remote Procedure Call	Mechanism to call a function on a remote object
RPC	Remote Procedure Call
SEA	Survey Entry Application
Serialization	Serialization is the process of changing data into a structure suitable for transportation over a remote border. At the other side of the remote border it must be deserialized into the original data
SOC	Separation of Concerns
Separation of Concerns	Separating the development of distributed systems in the development of the distributed system in general and the components
SOC-Framework	The framework supporting the separation of concerns
SRA	Survey Report Application
Stateful call	Call that depends on previous calls, or later calls are dependant on this call
Stateless call	Independant call, i.e. not dependant on previous calls, and no other calls are dependant on the stateless call
Survey .NET Project	Project within Imtech for developing, maintaining, filling in and reporting a survey
Survey Entry Application	Application within the Survey.NET Project to fill in a survey
Survey Report Application	Application within the Survey .NET Project to report a survey
TU/e	Eindhoven University of Technology

1. Introduction

The document describes a graduation project for the Eindhoven University of Technology (TU/e) and the company "Imtech ICT iQuipware". The introduction provides the abstract, an introduction to Imtech ICT iQuipware and introductions to the main issues of the project.

1.1 Abstract

The project focuses on developing a general framework for developing distributed systems with the recently introduced .NET Technology of Microsoft. Besides structuring the process of developing distributed systems, the goal of this framework is to divide the development of distributed systems into two independent parts: the development of system components and the development of the entire distributed system. This separation of concerns in the development of distributed systems is useful, since it introduces a new independent specialization area that focuses on the distributed properties of a distributed system. The framework is implemented with the use of .NET Technology and validated on a case study on a distributed application developed using the framework. This case study reveals the usability and correctness of the framework, and helps to make a proper conclusion about it.

1.2 Imtech ICT iQuipware

Imtech ICT iQuipware, a combination of the companies formerly known as Mountside Software Engineering and Turnkiek Technical Systems Eindhoven, is a separate business unit within Imtech ICT. Its primary focus is targeted at building software for industrial and technical environments. Examples of such environments are machine and production line control systems, medical systems, digital printing systems, production line vision, CD/DVD manufacturing equipment and motion control systems. They are CMM level 2 certified, which means that they develop software according to a quality process. They prefer to execute projects in-house, but there are also people externally working on projects. They are very Microsoft oriented, which results in a special interest in Microsoft .NET, the latest technology of Microsoft.

1.3 Distributed Systems

This paragraph gives an introduction to distributed systems.

Definition

Since distributed systems have many different facets, it is hard to define a distributed system with one single definition. A specific definition given by [Ver] is:

"A distributed system is a system composed of several computers which communicate through a computer network, hosting processes that use a common set of distributed protocols to assist the coherent execution of distributed activities."

A more loose definition is used for this project:

"A distributed system is a collection of inter-related cooperating components, residing on several different computers connected through a computer network".

The definition of a component is adapted from [Bal], since it uses components in the same context as here:

"A component is a reusable black/grey-box entity (a piece of code) with well-defined interface and specified behaviour which is intended to be combined with other components to form a software system (an application)".

Need for distributed systems

From [Wu] and [Ver] the following motivations for developing an application in a distributed way are extracted:

- Inherently distributed applications: when the problem has a decentralized nature, it is not natural for the locus of control or the state repository to be centralized. E.g. a manufacturing enterprise network performing concurrent engineering activities from remote locations.
- Performance: the parallelism of distributed systems reduces processing bottlenecks and provides improved all-around performance. E.g. a process is divided in several parallel processes running on several computers.
- Resource Sharing: Distributed systems can efficiently support information and resource sharing for users at different locations. E.g. a central database used by applications on different locations, sharing the central stored information.
- Flexibility and extensibility: Distributed systems are capable of incremental growth and have the added advantage of facilitating modification or extension of a system to adapt to a changing environment without disrupting its operations. E.g. a manufacturing enterprise network adds a new engineering activity on a remote location.
- Availability and fault tolerance: Distributed systems have the potential ability to continue operation in the presence of failures in the system. E.g. if in a manufacturing enterprise network one engineering activity fails, the other engineering activities are able to continue.

1.4 Microsoft .NET Platform

The .NET Platform has recently been introduced by Microsoft. It consists of a framework and a development environment. The .NET Framework is the infrastructure for the overall .NET Platform and consists of standard class libraries and a run-time environment called the common language runtime (CLR), which manages the execution of code and provides services that make the development process easier.

The .NET Platform is a new step in the evolution of software development. It combines the strengths of several different development environments / architectures and introduces new mechanisms for more rapid and reliable software development. Some main features of the .NET Platform are:

- Use of a garbage collector
- Use of Java-like platform independent "bytecode" (called Intermediate Language (IL))
- The development environment called Visual Studio .NET supports easy-to-build user interfaces (similar to e.g. Delphi and Visual Basic)
- Rich collection of standard classes
- Completely new security architecture, consisting of Code Based Security, Role Based Security, Runtime security facilities (cryptography, isolated storage, etc.)
- Improved Data Access Support, better integration with SQL Server
- Rich XML Support. Parsing, exporting and converting into data tables are standard features in .NET
- Improved support for developing Distributed Systems by providing a higher level mechanism for executing remote procedure calls

The last feature is part of the subject of research in this project.

1.5 Overview

The structure of this report is in line with the research oourse as described in paragraph 2.3, with an additional chapter containing the conclusion and the recommendations (Chapter 7).

2. Research Plan

This chapter describes the problem, the goals and the course to achieve these goals.

2.1 Problem Formulation

Imtech ICT iQuipware uses a lot of Microsoft development tools. The recently developed .NET technology of Microsoft is therefore of special interest for Imtech. .NET Remoting is the part of .NET that supports the development of distributed systems. Since Imtech works a lot with distributed systems, this part is of high importance. Developing distributed systems is a specific area of computer science and requires a lot of knowledge and experience from all developers of distributed systems. .NET Remoting contains standard building blocks that provide an easy way of building distributed systems. .NET Remoting is a step in the direction to separate the development of components and the development of distributed systems in general, but does not provide a full separation of concerns. Therefore in this situation all component programmers must still be aware of most of the distributed properties of a system. This is not desired for Imtech, since not all programmers are experts in distributed systems.

2.2 Goal

Establish a wider separation of concerns for developing distributed systems with .NET Remoting for typical Imtech applications.

2.3 Course

The following steps are meant to lead to the goal:

- Study the general properties of distributed systems (Chapter 3).
- Research the possibility to create a framework (built on .NET Remoting) that provides a maximum separation of concerns. This framework will be called the SOC-Framework (Separation Of Concerns) in the rest of the document (Chapter 4 and 5).
- Build the SOC-framework (providing a maximum separation of concerns feasible within the time available) (Chapter 5).
- Test the correctness and usability of the SOC-framework on a case study by developing a distributed application with the SOC-framework (Chapter 6).

3. Properties of Distributed Systems

This chapter studies the general properties of distributed systems. Being aware of these properties is necessary to build the SOC-framework, since the goal of the SOC-framework is to handle these properties. In paragraph 5.4 is discussed how these properties are handled by the SOC-framework.

3.1 Dependability

[Ver] states that a system is dependable if it exhibits a high probability of behaving according to its specification. There are two problems with this "definition": a complete and unambiguous specification is often not available and high is a relative concept. Therefore it gives the following definition of dependability:

"Dependability – the measure in which reliance can justifiably be placed on the service delivered by a system."

According to [Ver], [Wu] and [Tan], dependability consists of the following issues:

- Availability: the measure in which a system is ready to be used immediately. A highly available system is one that will most likely be working at a given instant in time.
- Reliability: the measure in which a system can run continuously without failure.
- Safety: the measure in which a system, upon failing, does so in a non-catastrophic manner.
- Maintainability: the measure in which a system can be easily restored upon failing.

[Wu] also mentions security, but security will be discussed here as a separate property of a distributed system. To build a dependable system it is needed to prevent system failures. In general, the process that leads to a system failure is the following: **system fault – system error – system failure.** A system fault can be:

- Failure of a component.
- Failure of communication between components.
- Server failure.
- A system design fault, i.e. components are working properly according to their own specification, but the interaction of the components does not result in the desired system specification.

A system error is the manifestation of a system fault. The system fails when this system error results in the system performing one of its functions incorrectly. There are several ways to prevent system failures:

- Removing system faults: Detecting faults and removing them before they have the chance of causing an error.
- Preventing system faults: Preventing the causes of errors by eliminating the conditions that make fault occurrence probable during system operation. For instance, using high quality components, components with internal redundancy, rigorous design techniques, etc.
- Tolerate system faults, but do not let errors result in a system failure.

Since removing and preventing faults cannot eliminate all faults, it is needed to tolerate faults and block the effect of the fault before it generates a failure. In such case, we say that the system is capable of providing correct service despite the occurrence of one or more faults or, in other words, that the system is fault-tolerant. Fault tolerant systems use error processing. Error processing has two facets:

- Error detection: the first step at avoiding failure. Detection can be performed by several mechanisms, depending on the type of error.
- Error recovery: there are three ways of recovering from an error.
    - Backward recovery: the system goes back to a previous state known as correct and resumes. Going back to a correct state is not as trivial as it may look, since other components (possibly outside the system) could be affected by the error. One common form of backward error recovery involves progressing by steps in the computation, and storing the system state at the end of each step. The saved state is called a checkpoint.
    - Forward recovery: the system proceeds forward to a state where correct provision of service can still be ensured. Forward recovery is often a necessity. In some cases there is not enough time to go back to a correct state and to restart from there. External actions that are impossible to undo also require some form of forward recovery when errors occur.
    - Error masking: the system state has enough redundancy that the correct service can be provided without any noticeable glitch. Errors can be automatically masked and never become visible, since there will always be a way of providing the correct result.

"The use of redundancy is fundamental to fault tolerance. In computer systems, redundancy can be applied in the space, time and value domains. Space redundancy consists of having several copies of the same component. Time redundancy consists in doing the same thing more than once, in the same or in different ways, until the desired effect is achieved. Value redundancy consists in adding extra information about the value of the data being stored or sent". [Ver]

According to [Ver], distributed fault tolerance provides the ground to tolerate following classes of faults:

- Hardware faults: since the results of the several replicas are consolidated such that a correct value is returned, despite the failure of hardware components
- Transient software faults: since they occur sporadically and are thus normally masked by redundant execution in different environments
- Disasters: since the geographic dispersion of nodes supported by distributed fault tolerance provides the basis to survive them
- Design faults: in the case of design faults, simple replication provides little help, since errors will systematically occur in all replicas. In consequence, what is needed is that replicas are designed diversely. Note that this is very expensive and it still not guarantees success, since different designers can make the same design faults.

3.2 Load Distribution

If for a component redundant components exist within a distributed system, calls on this component can be distributed over these redundant components. How calls are distributed is determined by a load distribution algorithm. There exist several load distribution algorithms. Specific algorithms will not be discussed here, but a general classification for load distribution algorithms will be given (adapted from [Wu]):

Static vs. Dynamic: In static load distribution, allocation of calls to components is done in advance before components are executing, while in dynamic load distribution no decision is made until components are executing in the system.

Centralized vs. Decentralized Control (within Dynamic): In decentralized control the work involved in making decisions is distributed among different components, while such decisions are assigned to one component in a centralized control.

Cooperative vs. Non-cooperative (within Decentralized Control): Dynamic load distribution mechanisms can be classified into ones that involve cooperation between distributed components (cooperative) and ones in which components make decisions independently (non-cooperative)

Non-adaptive vs. Adaptive: A non-adaptive load distribution algorithm uses only one load distribution policy (algorithm) and does not change its behaviour according to the feedback from the system. An adaptive load distribution algorithm adjusts its load distribution policy based on the feedback. Typically, an adaptive load distribution algorithm is a collection of many load distribution algorithms. The selection among these algorithms depends on various parameters of the system.

3.3 Mutual Exclusion

Two processes are mutual exclusive if they share a resource that may not be used by both processes at the same time, i.e. the two processes are mutually conflicting. Several algorithms exist to enforce mutual exclusion (see [Ver], [Wu], and [Tan]). These algorithms use critical regions within the processes itself, i.e. within the components containing these processes. Since this is not according to the SOC-principle, these algorithms will not be discussed here.

A SOC-algorithm for enforcing mutual exclusivity is active outside the component. Therefore the minimal scale of mutual exclusivity is on function level. A function can be mutual exclusive with itself (i.e. only one instance of this function may be active at a time) or can be mutual exclusive with other functions. Two functions within a component are mutual exclusive if they use a same exclusive resource within this component. Examples of exclusive resources are (shared) data, machines (e.g. printer) etc. If all functions are mutual exclusive, the entire component is mutual exclusive. A SOC-algorithm should enforce that no two mutual exclusive functions can be active at a time.

According to [Wu], a regular mutual exclusion algorithm should satisfy:

- Freedom from deadlock. Processes should not wait endlessly on each other. For a detailed description of deadlock, see paragraph 3.5.
- Freedom from starvation. Each request from a process should eventually be granted.
- Fairness. Requests should be granted based on certain fairness rules. Freedom from starvation and fairness are related with the latter being a stronger condition.

3.4 Transactions

This paragraph discusses transactions. First, transactions in general will be handled, and then specific details about transactions in distributed systems are discussed.

3.4.1 Transaction in General

From [Ver] the following definition for a transaction is extracted: "A transaction is a paradigm that allows arbitrary sequences of operations on data items to be transformed into atomic (i.e. indivisible) operations." The need for transactions can be explained with an example: consider a bank transfer that withdraws money from account A and deposits the same amount in account B. If this sequence is interrupted after withdrawing the money from account A by some reason (e.g. a failure), money has disappeared (the sum of account A and account B has changed).

To ensure integrity of the data, a system supporting transactions must satisfy the following properties of the transactions (adapted from [Sil]):

Atomicity: Either all operations of the transaction are successfully executed, or none is executed. A transaction commits when it successfully terminates; otherwise a transaction aborts.

Consistency: Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the data.

Isolation: Even though multiple transactions may execute concurrently, the system must guarantee the same result as if the transactions were executed serially. Thus, each transaction is unaware of other transactions executing concurrently in the system.

Durability: After a transaction completes successfully, the changes it has made to the data persist, even if there are system failures.

These properties are also known as the ACID properties of transactions. According to [Ver], there are two ways for achieving atomicity:

- Delay the effects of the transaction until the transactions commits (i.e. keep the intermediate results in a log, called the redo log)
- Operations may be allowed to execute immediately, as long as the system has a way to undo these effects (using an undo log), in the case the transaction aborts.

Isolation can be achieved by using lock-based or timestamp-based algorithms. Lock-based algorithms make resources for a certain period mutual exclusive (lock the resource), such that other transactions cannot change this resource. Timestamp-based protocols assign a timestamp to each transaction and use these timestamps to decide whether a resource may be changed by an action of a certain transaction. See [Wu] for more details.

Adapted from [Sil], durability can be guaranteed by ensuring that either:

- The actions carried out by the transaction have been executed before the transaction completes
- Information about the actions carried out by the transaction written to stable storage is sufficient to enable the system to reconstruct the actions when the system is restarted after failure

[Ver] states that "databases are the ideal field of application of transactions. In systems that perform external actions, also called real actions, enforcing transactional semantics is much more complex and sometimes impossible". Many of the current Database Management Systems (DBMS) have mechanisms that support the use of transactions.

3.4.2 Transactions in Distributed Systems

[Ver] gives a clear overview on distributed transactions: "Distributed transactions are simply transactions that access items on different nodes of a distributed system. Distributed transactions can be built using the mechanisms developed for centralized transaction. ... In fact, a distributed transaction can be described as a collection of several sub-transactions, each individual sub-transaction being initiated on each node visited by the distributed transaction. The local transactional mechanisms on each site will guarantee that each individual sub-transaction either executes completely or aborts".

A problem specific to distributed transactions is that executing an action of a transaction on a node is not reliable. Since an action can fail, this can disturb the atomicity property. If one action fails and all other actions succeed, the transaction does not atomically execute all actions. To deal with this problem, nodes participating in a distributed transaction need to execute an atomic commitment protocol. Such protocols ensure that all nodes on which a transaction executes agree on the final outcome of the execution. "Among the simplest and most widely used commit protocols is the two-phase commit protocol (2PC). An alternative is the three-phase commit protocol (3PC), which avoids certain disadvantages of the 2PC protocol, but adds to complexity and overhead" [Sil]. Both protocols start with a prepare phase. The node "executing" the transaction is the coordinator. It sends a prepare message to all participating nodes. If they all agree, the next phase is started. In 2PC, a commit message is sent to all participating nodes. If they all successfully commit their action, the transaction is completed. In 3PC, a precommit phase is inserted to inform all participants about the preliminary decision regarding to the fate of the transaction. See [Ver, Sil] for more details.

In case of a nested transaction, all recursive underlying transactions must execute its operations atomically. A failure of one node in the chain of nested transactions may not result in an inconsistent state of the underlying nodes. Therefore, the atomic commit protocols must also be executed nested, e.g. if a sub-transaction receives a prepare message, this preparation will among other things consist of preparation of all underlying transactions.

On a single database, the DBMS handles concurrent transactions on that database. To deal with distributed transactions, transactional systems exist. These are complex systems that guarantee the ACID properties for (distributed) transactions in an efficient way.

3.5 Deadlock

Deadlock is a system state in which two ore more processes are waiting for each other in a configuration from which no progress can be made. Deadlock is caused by the need to lock resources. Locking is used for concurrency control, i.e. when multiple processes share the same resources (e.g. with transactions).

Formally, a deadlock can arise if and only if the following four conditions hold simultaneously [Ver,Wu]:

- Mutual exclusion: no resource can be shared by more than one process at a time
- Hold and wait: at least one process waits for additional resources while holding non-sharable resources
- No pre-emption: a process that holds a resource is allowed to keep it until it is ready to release it
- Circular wait: there is a circular chain of n processes, where p_0 is waiting for a resource held by p_1, which in turn is waiting for a resource held by p_2, …, and p_{n-1} is waiting for a resource hold by p_0

Following strategies exist for handling deadlocks [Ver,Wu]:

- Deadlock prevention: eliminate one of above mentioned deadlock conditions
- Deadlock avoidance: making checks before a resource is given to a process, to make sure that the conditions for deadlock are not met
- Deadlock detection and resolution: allow deadlocks to occur. Detect deadlocks and break them.

Deadlock prevention is not a trivial task. Elimination of the deadlock conditions will be discussed now:

- Mutual exclusion could be eliminated by having only sharable resources, but this is too restrictive a condition for most applications (e.g., shared data structures are simply non-sharable)
- Hold-and-wait may be eliminated if processes just hold one resource at a time, but again this is too restrictive for most applications. Alternatively, one might force a process to request all the resources it needs a priori, but this may be very inefficient, since some resources are locked long before they are actually needed.
- Non-preemption may be eliminated by allowing pre-emption, i.e. by forcing a process to release its resources. Since a process that holds a resource is likely to have changed the state of this resource, a process that is forced to release a resource may be forced to rollback its actions on this resource.
- The circular-wait condition can be prevented by imposing a total order on all resources and forcing processes to acquire all resources by the same order. Just as holding all resources simultaneously a priori, this may be very inefficient, since the order defined for the resources may not be the order in which the process needs to access the shared resources.

"Deadlock avoidance consists of making checks before a resource is given to a process, to make sure that the conditions for deadlock are not met. Deadlock detection consists in automating the process of detecting deadlocks, usually by detecting existing circular-wait conditions. Deadlock resolution consists in automating the process of breaking the deadlock, usually by aborting one or several of the processes involved. Distributed algorithms for deadlock detection or avoidance are much harder than their centralized counterparts. The problem with distributed deadlock detection is that the algorithm needs to capture the global state of the computation, in a system where lock requests can be 'in transit', i.e., in messages exchanged among nodes." [Ver]

3.6 Communication

Principally there exist two communication schemes in distributed systems [Wu]:

- *Message passing* which allows processes to exchange messages. An interprocess communication facility basically provides two abstract operations: send (message) and receive (message)
- *Shared data* which shares data for communication in a distributed system with no shared memory

In paragraph 5.1 is explained why this project does not focus on shared data. Therefore message passing will be discussed now. Adapted from [Wu], following decisions has to be made when implementing a message passing scheme:

- One-to-one or one-to-many: one-to-one message passing is between two components while one-to-many message passing supports a broadcast facility
- Synchronous or asynchronous: with synchronous message passing, the sender is blocked until the receiver has accepted the message. In asynchronous message passing the sender does not wait for the receiver
- One-way or two-way communication: one-way communication represents one interaction between the sender and the receiver, while two-way communication indicates many interactions, back and forth, between the sender and the receiver
- Direct or indirect communication: the difference between direct and indirect communication is whether the sender sends messages to the receiver directly or indirectly through an intermediate object, usually called a mailbox or port

There are several message-passing models that are commonly used. Since .NET Remoting supports a specific message-passing model based on Remote Procedure Call (RPC), the RPC model will be discussed now:

"RPC is similar to the client/server model where the client requests a service and waits for the results form the server." [Wu] The aim of RPC is to enable programmers to call procedures on another component (server) as if calling a local procedure. The concept of RPC is simple. The client makes a call on the server by making a local call on a fake object that has the same interface as the server. This fake object is called a proxy (proxy-design-pattern [Bus,Gam]). This proxy communicates with a server proxy, which executes the call on the server. The reply is sent back to the client proxy and finally this reply is sent to the original caller. This process is depicted in figure 3.1:

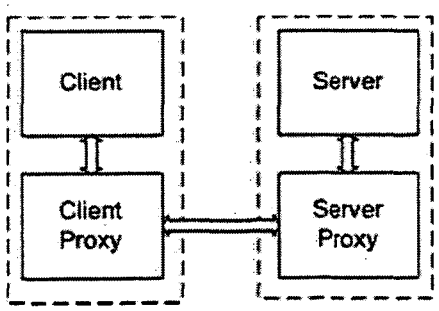


Figure 3.1 Remote Procedure Call (RPC)

In more detail, the following happens between the client and server proxy:

"The client proxy is responsible for sending a request to the server and waiting for a reply. To send the request, the proxy must first create the corresponding message. In order to do so, the stub converts the input parameters in a format suitable for being transmitted over the wire and recognizable by the server. This process is known as marshalling. After being formatted, the message is sent to the server through some session level protocol using the communication system. That protocol is responsible for retransmitting the request, waiting for replies, discarding duplicate or obsolete replies, etc. On the server side, the message follows a symmetric path. It is received and processed by the session protocol, which identifies the target service and delivers it to the relevant server proxy. The server proxy extracts the parameters from the message (this task is called unmarshaling) and does a local call to the real procedure that does the work. When this function returns, the server proxy creates a reply message and hands it to the session protocol in order to be returned to the client, following the same steps as before, now in the opposite direction. On the client side, the client proxy finally returns the original call to the client, with any relevant results". [Ver]

3.7 Component Activation / Lifetime

Remarkably the literature does not mention the activation and lifetime of a component within a distributed system. Since in my opinion this property cannot be left out, the theory behind the .NET way of component activation and lifetime is discussed here:

Components in a distributed system can be activated by the server or the client, service one or more clients and can have different lifetimes. The difference between server activation and client activation is that with client activation the client explicitly asks the server to create the remote component. With server activation, the remote component is created when the first call for this component is received at the server.

Following lists the possible combinations within .NET Remoting:

- Singleton:
  - Activated by server
  - Service multiple clients
  - Lifetime >= 1 call(s) (created on the first call)
- Single Call:
  - Activated by server
  - Service one single call
  - Lifetime = 1 call
- Client-Activated:
  - Activated by client
  - Service one single client
  - Lifetime >= 0 call(s)

A singleton component can share state information between calls of different components, where a client-activated object can only share state information between calls of one component. A single call object cannot share state information, since it only lives during 1 call.

There are several scenarios possible for determining the lifetime of singleton and client-activated objects:

- The object lives as long as there are clients for these objects
- The object lives for a pre-specified time (possibly extended when clients call the object)
- The object lives forever

3.8 Localization

Also localization is not mentioned in the literature. Localization is the way in which components localize each other. Globally there are two possibilities:

- Direct localization: one component knows how to localize the other component directly
- Indirect localization: a central broker coordinates the localization process

3.9 Security

Security is an important issue when a distributed system is residing in an open environment, i.e. an environment where it is possible that unauthorized parties can threaten your system. In the scope of this project, distributed systems will reside in a closed environment. Therefore security will only be discussed globally.

There are four types of security threats [Tan]:

- Interception: refers to the situation that an unauthorized party has gained access to a service or data. An example of this is when communication between two parties has been overheard by someone else.
- Interruption: refers to the situation in which services or data become unavailable, unusable, destroyed etc. (e.g. denial of service attacks)
- Modification: involve unauthorized changing of data or tampering with a service so that it no longer adheres to its original specifications (e.g. changing transmitted data).
- Fabrication: refers to the situation in which additional data or activities are generated that would normally not exist. For example, an intruder may attempt to add an entry into a password file or database.

Before it is possible to protect your system, it must be known which actions are allowed and which ones are prohibited in the system. This is described in the security policy. Once a security policy is available, the security mechanism can enforce this policy. Important security mechanisms are [Tan]:

- Encryption: transfers data into something that an attacker cannot understand (only the receiver can).
- Authentication: is used to verify the claimed identity of a user, client, server and so on.
- Authorization: after a client has been authenticated, it is necessary to check whether that client is authorized to perform the action requested.
- Auditing: auditing tools are used to trace which clients accessed what, and in which way.

4. Separation of Concerns in the Development of Distributed Systems

This chapter discusses the separation of concerns for developing distributed systems as earlier mentioned in the research plan (chapter 2) in more detail.

4.1 Separation of Concerns

In this project, "Separation of Concerns (SOC)" in the development of distributed systems is defined as:

"Separation of the process of developing distributed systems into two processes: the development of the system components and the development of the distributed system in general."

By applying the SOC-principle, developing distributed systems becomes a more structured process. Experts in distributed systems can focus then on the architecture of the system and the distributed properties of it, while component programmers can focus on the development of the components.

The course of developing a distributed system would then be as follows:

- Experts in distributed systems design the distributed architecture (including component specification)
- Component programmers develop the components and/or of-the-shelf components are acquired
- Experts in distributed systems collect the necessary components and configure the distributed system properties.

4.2 Need for Separation of Concerns

Developing distributed systems using the SOC-principle has the following advantages over developing distributed systems without using the SOC-principle:

- ***Specialization:*** In the SOC-approach each development process is carried out by its own experts. In the non-SOC-approach, each developer has to master both development processes. This results in a weaker system since not all developers are specialized in the process of developing distributed systems.
- ***Overview:*** The SOC-approach is more structured than the non-SOC-approach. Therefore the overview of the entire distributed system is better.
- ***Maintenance:*** By using the SOC-approach, it is possible to let the middleware that establishes the SOC-principle (i.e. the SOC-framework) log which components are not functioning correctly. For the distributed system expert this is very important information to maintain the system.
- ***Error detection / fixing:*** The separated approach makes it possible to run separate tests on components and the entire distributed system. Errors can be detected and fixed per separate development-process. Fixing errors in the development-process of distributed systems can now be done without changing the code of the components.
- ***Flexibility:*** In the separated approach it is easier to adapt the properties of the distributed system, since it is a separate part of the design.

The primary need for using the SOC-principle was mentioned in the problem formulation (paragraph 2.1), and is referred to as "Specialization" in the list above. The other advantages can also be seen as secondary needs.

4.3 Development of Distributed Systems

Using the SOC-principle, the development of distributed systems only focuses on the architecture of the system and the distributed properties of the system. The components specified in the architecture are assumed to be functionally acting according to their specifications.

The first part of the development of distributed systems is the architectural part, which specifies what components are needed in the system. Here distribution must already be kept in mind, since some architectural constructions are needed for specific distributed system properties (e.g. if for some component reliability is very important, several of these components could be placed in the systems architecture).

The second part of the development of distributed systems is the configuration of the properties (as mentioned in chapter 3) of the distributed system. The configuration of these properties will be named the "system configuration" in the rest of this document. Chapter 7 discusses the system configuration in more detail.

4.4 Development of Components

Using the SOC-principle, the development of components is independent of the development of the distributed part of the system. Therefore it is sufficient to specify a component by providing the interface of the component and the specifying the interface methods. The component can now be developed according to its specification, without concerning about the interaction with other components within the entire system. The distributed system expert needs certain information of the component for the distributed system configuration (the exact information can be found in the next chapter). Therefore, the component developer must provide this information. Since the needed information is not related to other components within the system, this is still according to the SOC-principle.

5. SOC-Framework

This chapter discusses the SOC-Framework. First the requirements and the properties of the framework are discussed. Then the architecture of the SOC-framework is described. This paragraph is followed by a paragraph about .NET Remoting. Then is discussed how the SOC-framework handles the properties of a distributed system. Following on this paragraph the implementation of the SOC-framework is dealt with and finally the configuration of the SOC-framework is discussed.

5.1 Requirements and Properties

Summarizing the previous chapters, the following requirements can be distinguished for the framework:

R1) The framework must support the development of distributed systems according to the SOC-principle
R2) The framework must be implemented in .NET
R3) The framework must be suitable to build distributed .NET applications with

From the requirements, the following properties of the framework can be adapted:

P1) Because of the SOC-principle, components in the distributed system are independent of the properties of the distributed system. Therefore they must be seen as "black boxes", containing only an interface
P2) The framework supports setting a system configuration (i.e. setting the properties of the distributed system)
P3) The framework manages the distributed properties of the system (i.e. supports the system to behave accordingly to its configuration)
P4) Since .NET Remoting uses calling for communication (see 5.3.3), calling is used for communication between components (R2)

5.2 Architecture

This paragraph describes the architecture of the SOC-framework. It discusses the incremental steps for reaching the final architecture.

5.2.1 Managing Distributed System Properties

Since the components in the distributed system are black boxes, and therefore cannot be changed, the framework must handle the properties of the distributed system. As a result, the framework must provide an extra layer that manages the properties of the distributed system using a certain configuration. This layer must manage the components to behave accordingly to their configuration and is therefore called the "manager". To make a component behave like its configuration, a manager must intercept and adapt the calls of a component. These calls can be incoming calls and outgoing calls. Since the management tasks of incoming and outgoing calls are not related, it is more natural to split the manager into an incoming and an outgoing manager. Because of the SOC-principle, a component may not make remote calls. Therefore the managers reside on the same server as the component. This also provides a fast and reliable communication. figure 5.1 depicts the manager layer in combination with two black box components.

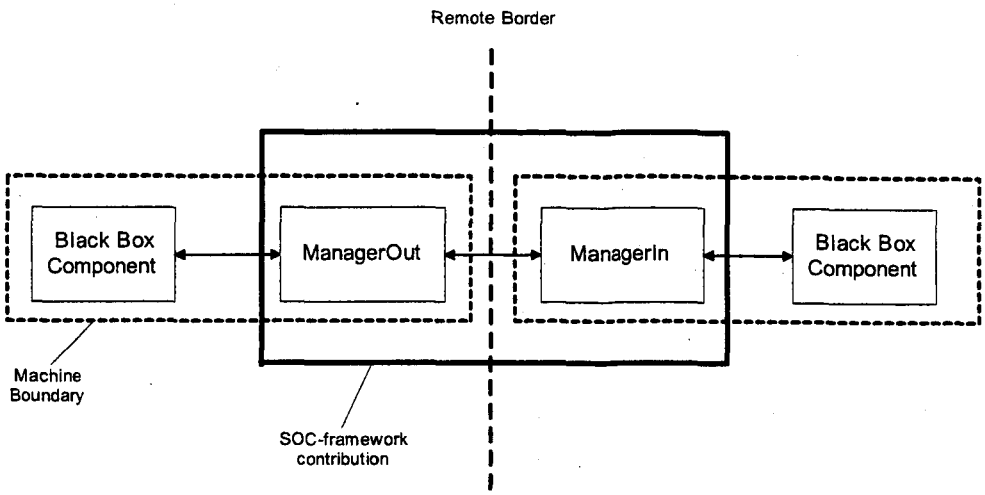


Figure 5.1: Using managers for managing distributed system properties

Several choices are possible for the number of outgoing managers on a server:

1. One central outgoing manager per server
2. One outgoing manager per component
3. One outgoing manager per instance of a component

Option 1 is the most centralized solution. It has the following advantages:

- Load Distribution: all outgoing calls pass this outgoing manager. Therefore it is easier to apply load distribution.

Disadvantages of option 1:

- Reliability: If the outgoing manager crashes, all components on that server are not able to execute outgoing calls.
- Each component must be aware of the location of the central manager.

Option 3 is the most decentralized solution. It has the following advantages:

- Most object oriented solution.

Disadvantages of option 3:

- Initialization time: for each new instance of a component, an outgoing manager must be instantiated.

Option 2 is a hybrid form of option 1 and option 3 and has a combination of (weaker) advantages and disadvantages of these options. In the architecture option 3 is chosen, since that results in the most natural and robust architecture. For the incoming manager a similar comparison can be made. Also here option 3 is chosen, with the same reason as for the outgoing manager. Since each instance of a component has its own incoming and outgoing manager, a distributed system will contain an equal amount of incoming and outgoing managers as instances of components. A common distributed system will consist of several components, and thus incoming and outgoing managers. Therefore it is desired that the framework provides standard managers which are as general as possible. The framework uses the most general solution for a

manager: a complete standard manager, i.e. a fixed manager that only provides a standard interface to be called. The configuration properties are read from a separate configuration medium.

5.2.2 Proxies

Making calls directly on the outgoing manager has the following disadvantages:

- For a component developer, it is not natural to make calls on a "manager". The developer should not be aware of implementation issues of the framework.
- It is not possible to apply type checking on the calls on the (complete standard) outgoing manager.

Therefore proxies (proxy-pattern [Bus,Gam]) are used for the called components. A proxy is placed between a component and its outgoing manager. The proxy looks to the developer like the remote component, but is a local component and actually only has the same interface. Calls on this proxy are via the managers forwarded to the real remote component. A proxy is an object transparent for the developer. Proxies are used as "normal" objects. This enables the developer to use remote objects as if they were local. figure 5.2 depicts the use of proxies:

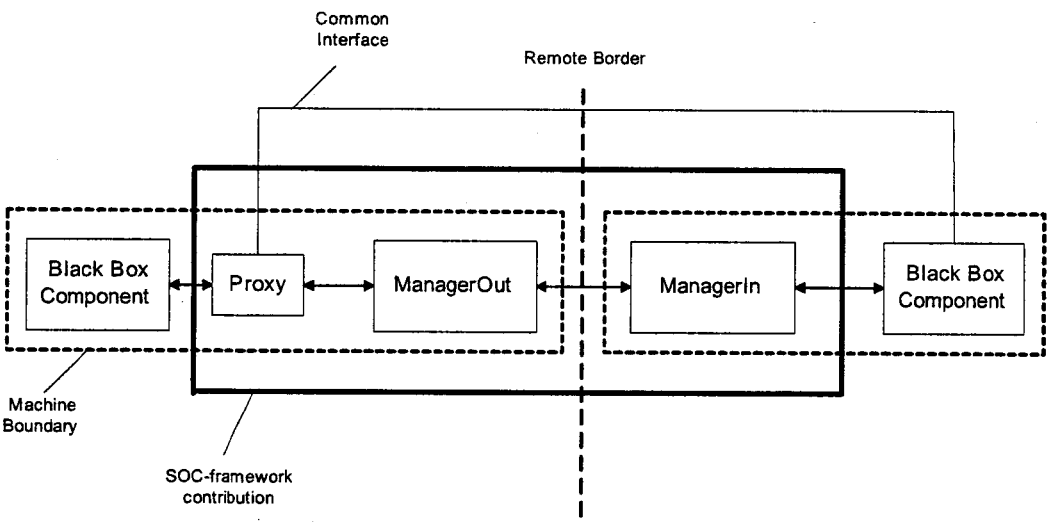


Figure 5.2: Using proxies to enable the component developer to make natural local calls

5.2.3 Remote Border

The interaction between the incoming and outgoing manager crosses the remote border. Therefore we need a mechanism that supports the interaction between two components over a remote border. In a distributed system using the SOC-framework, lowest level cross-border interaction is provided by .NET Remoting. Properties (as mentioned in chapter 3) that are needed on the lowest level to interact between two distributed components are: Communication, Component Activation / Lifetime and Localization. The architecture of the framework using .NET Remoting is depicted in figure 5.3:

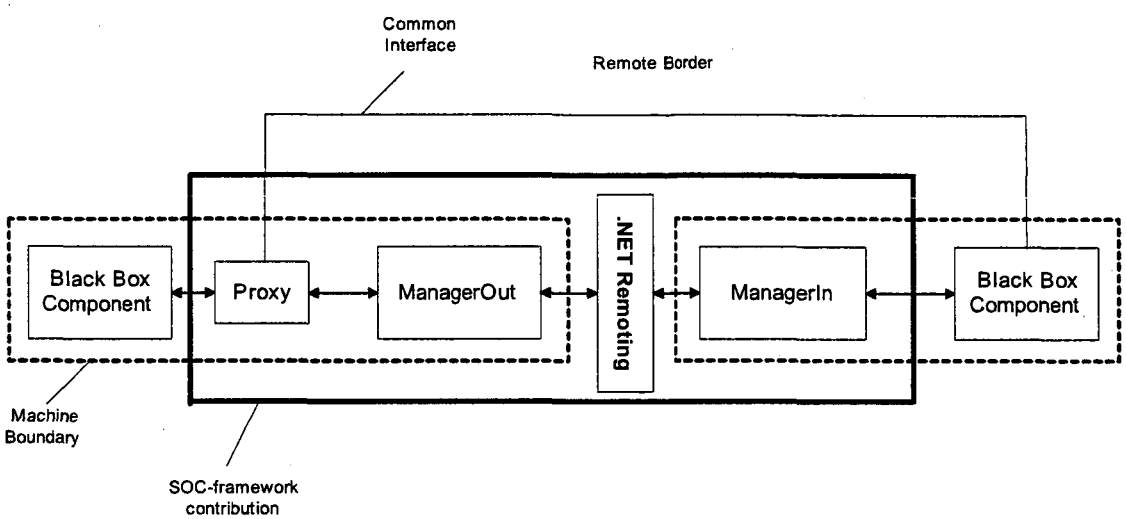


Figure 5.3 Architecture using .NET Remoting to cross the remote border

Details about the way in which .NET Remoting handles the interaction between two distributed components can be found in paragraph 5.3.

5.3 .NET Remoting

As discussed previously, .NET Remoting is used to provide the mechanisms for interaction between the incoming and outgoing manager within the SOC-framework. This paragraph discusses in general the interaction between two distributed components within .NET Remoting. First is discussed how to localize a remote component. Then the lifetime and activation of a remote component is discussed. Finally the communication between two remote components is dealt with.

5.3.1 Localization

.NET Remoting uses a direct localization mechanism. An example of a localization string is:

"tcp://computername:1234/endpoint"

where *"tcp"* is the protocol used for communication, *"computername"* is the name of the computer, *"1234"* is the port on which the computer is listening for incoming calls, and *"endpoint"* is an extra identifier for the remote component (it is possible to host multiple components on one channel).

5.3.2 Lifetime / Activation

.NET Remoting supports the three combinations of lifetime / activation as mentioned in paragraph 3.7 (Singleton, Single Call, Client Activated). Details about activation and lifetime are given below:

Activation

To activate a remote component in .NET Remoting, the client must know the interface of the remote object. The cleanest way to do this is to use shared interfaces. Disadvantage of this approach is that it cannot be directly used for client activated objects, since it is not possible to create instances of an interface. A factory (factory pattern, [Gam]) can be used to work around this problem, but then a function must be called to create the client activated object. Alternatives for using shared interfaces are shared base classes, shared assemblies or SOAPSUDS generated metadata. Using shared base classes has as disadvantage that a remote object can only inherit from one base class, since .NET does not support multiple inheritance. The use of shared assemblies is unnatural for distributed systems, since clients should not need to be aware of the implementation of the server object. SOAPSUDS is a .NET tool that can generate meta data of an assembly. Since this tool extracts the interface after the implementation of an assembly, this is not a structured way of developing distributed systems. The SOC-framework uses shared interfaces to activate remote objects. To activate a remote object using interfaces, a standard .NET class is used. This class provides a function that needs as parameters the type of the interface and the localization string. As result a proxy for the remote object is returned.

Lifetime

A remote component has an initial lifetime. This lifetime can be extended when a call is made on the component. The lifetime is represented by a "lease". If the lease expires, the .NET Remoting Lease Manager checks whether there are sponsors for this lease. A sponsor is an object that has been registered at the Lease Manager as sponsor for this lease. If there are sponsors for the lease, the lease is extended; otherwise the object is marked for garbage collection. Client-side sponsoring only works within a closed environment, since it does not work in the presence of firewalls. This is no problem for the SOC-framework, since it is used within a closed environment.

5.3.3 Communication

.NET Remoting uses the RPC-based communication mechanism as described in paragraph 3.6 for communication between two components. The client uses a proxy to make calls on the server. When a client calls a remote method on a proxy, the remoting infrastructure handles the calls, checks the type information, and sends the call over a channel to the server process. A listening channel picks up the request and forwards it to the server, which locates, creates (if necessary), and calls the requested object. The process is then reversed, as the server returns the response message via the server channel to the client channel. Finally, the proxy returns the result to the client object.

A channel is a transport mechanism that transports a message (e.g. a method call) to another component via an arbitrary protocol. A channel consists of a chain of message sinks. Each message sink processes a message. There are two message sinks within a channel which are mandatory: the formatter sink and the transport sink. Serialization is the process of changing data into a structure suitable for transportation over the remote border. At the other side of the remote border it must be deserialized into the original data. The formatter sink serializes a message to a transfer format. .NET provides two standard formatters (Binary and SOAP). A transport sink transfers the serialized message to a remote process. .NET provides two standard transport channels, supporting the TCP and HTTP protocol. It is also possible to have custom sinks. Custom sinks can be used for e.g. compression / encryption of the message contents.

The remoting architecture is depicted in figure 5.4:

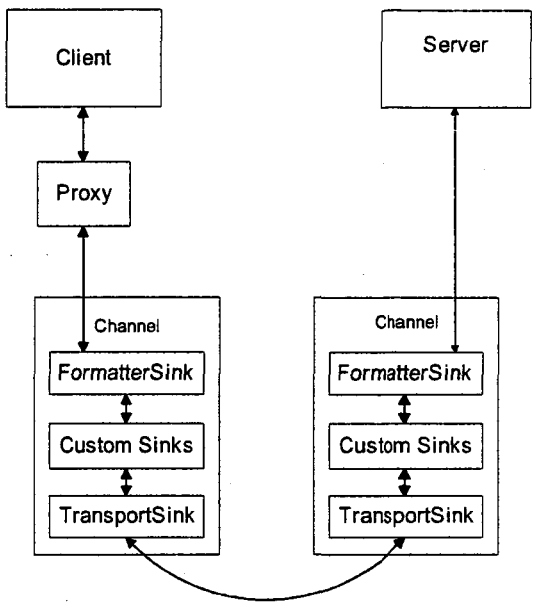


Figure 5.4 Remoting architecture for communication

5.4 Handling Properties

This paragraph discusses how the properties of a distributed system, as discussed in detail in chapter 3, are specifically handled by the SOC-framework.

5.4.1 Dependability

In the SOC-framework, no attention will be given to fault removal and fault prevention. These issues must be handled by application design and testing. The framework focuses on fault-tolerance.

Detection is the first step to avoid failure. The requesting component can then anticipate on the error. Specific failures in distributed systems are caused by non-reacting components. These components can crash or the communication channel may fail. Since the failed component cannot send an error message in these cases, these errors must be detected by time-outs. In the framework timeouts per external call can be set in the configuration. A proper time-out time must be deducted from the size of the call specified by the component developer. When making an external call, the outgoing manager reads the configuration and manages the timeout. If a call times out, an error is detected. The action to take on detection of an error is called error recovery and is discussed below.

Error recovery is very specific for each component / subsystem in a distributed system. Therefore it is difficult, if not impossible, for the framework to handle this only by managers, and not using the component itself. It is possible that the experts in distributed systems provide a "Recovery Agent". This is an object that can contain:

- A function to save and restore a correct state. Before calling a "dangerous" sequence of functions, the state must be saved (to enable backward recovery).
- A fixed checkpoint, i.e. point that contains a state that is always correct (to enable forward recovery).

On detection of an error the outgoing manager can consult the recovery manager and force a backward or forward recovery. But it is not always possible to simply store a save state, since changes could have affected other components. Requiring other components to store depending checkpoints is very difficult, since components can be accessed by multiple other components. In this case, using transactions is a better mechanism to solve the problem than using checkpoints, but cannot always solve the problem (e.g. in case of external actions that cannot be rolled back).

Error recovery based on checkpoints will not be implemented in the framework. One reason for this is mentioned above, another reason is mentioned by [Ver]: "if enough redundancy is added to the system, errors can be automatically masked and never become visible, because there will always be a way of providing the correct result".

Therefore the framework focuses on redundancy. The framework uses on time and space redundancy. It will use a form of active and passive replication as follows:

- If in a distributed system several redundant components are available, one call can be executed on several (same) components. In case of failure of a component, the other components are still able to give a result. The results can be compared and about the correct result can be "voted". This is also an improvement on the reliability of the result. It is also possible to take the first result as result. This is an improvement on performance, since the performance of the call is the performance of the quickest component. In the framework, the latter is implemented.
- If a call times out, this call can be executed again, possibly on a different (redundant) component.

When making calls on different redundant components, it is important to realize whether the call is stateful or stateless. This information is provided by the component developer, and is read from the configuration. A stateless call is an independent call, i.e. not dependant on other previous calls, and no other calls are dependant on the stateless call. If a call is stateless it does not matter on which locations the call is executed. If a call is stateful, it is dependant on previous calls, or later calls are dependant on the stateful call. A stateful call may only be executed on locations where all previous depending calls have been executed.

Each failure of an individual call on a component must be logged by the outgoing manager. Examples of items that can be logged are: Time, calling component, called component, called function, error message, etc. Logging is trivial and is currently not implemented in the SOC-framework. If an entire call is not able to execute successfully, even when using redundancy, it failed. The outgoing manager notifies the client proxy about the failure by throwing an exception. The client proxy can rethrow the exception or return an error value. This is still according to the SOC-principle, since a client proxy can be specified as possibly failing. The distributed system developer can specify the behavior on failure (e.g. throwing an exception) and the component developer can anticipate on the specified error.

5.4.2 Load Distribution

Load distribution can be applied when an external outgoing call is made. Therefore the outgoing manager is most suitable to deal with load distribution. The load distribution algorithm and the locations of a component are set in the configuration. When an outgoing call is made, the outgoing manager reads the load distribution configuration and selects the component on which the call will be executed accordingly to the specified algorithm. This is a decentralized and dynamic algorithm (dynamic since configuration can be changed during execution). Whether the algorithm is cooperative or non-cooperative, adaptive or non-adaptive depends on the implemented algorithms in the outgoing manager. Since this project is not specific about load distribution, only one algorithm is implemented in the SOC-framework. Note that for a cooperative algorithm, the incoming manager must also be adapted to support a cooperation-protocol.

5.4.3 Mutual Exclusion

The incoming manager is most suitable to deal with mutual exclusion, since all incoming calls pass the incoming manager. The incoming manager must be aware of the functions that are mutual exclusive. Which functions are mutual exclusive must be specified by the component developer. The mutual exclusive functions of a function can be set in the configuration. It is also possible to include "yourself", if a function may not be multi-threaded. A disadvantage of function-based locking is that blocking on code level is more efficient: not all code in a function need to be mutual exclusive to other functions. An advantage is the robustness and structural way of achieving mutual exclusiveness.

5.4.4 Transactions

Theoretically it is possible to deal with transactions within the SOC-framework. A separate transaction object can be provided by the distributed system architect, enabling the component programmer to execute the transactions provided by this transaction object. This transaction object consists of two parts: the local actions and the external actions (i.e. the subtransactions). The subtransactions are also contained in a transaction object. The commit protocol would then be executed in a nested way. E.g. the 2 phase commit protocol would be as follows:

Prepare:

- acquire locks for local actions
- store local actions
- prepare subtransactions

Commit:

- execute local actions
- release locks

Since all locks are acquired in advance, no rollbacks are needed. In case of deadlock, the transaction should be aborted. Deadlock can be detected using a timeout.

In practice this solution is hard to use. It has the following disadvantages:

- Decentralized approach: there is no central transaction coordinator, and therefore no central scheduling is possible. Central scheduling results in the most optimal schedule, since a central scheduler has the best possible overview of the system.
- It uses pessimistic locking: since all possible subtransactions need to be locked in advance, too much locking is necessary in case of "dependable" transactions, i.e. transactions whose precise actions can vary depending on intermediate results.

This transaction mechanism is not implemented within the SOC-framework. Transactions are a very complex field within computer science. Many people have been researching for many years on this particular subject, building huge transactional systems. Therefore it is not very realistic, in the scope of this project, to target at another transactional mechanism within the framework for this moment. Distributed system experts can provide transaction objects using standard available transaction mechanisms (e.g. Microsoft Transaction Server).

5.4.5 Deadlock

The SOC-framework itself does not prevent or avoid deadlock, since the SOC-framework has no control over the calls within the components (SOC-principle). To handle deadlock the SOC-framework provides a mechanism to detect and break deadlock, which is discussed later in this paragraph. Detecting and breaking deadlock solves the deadlock problem, but deadlock prevention is the most optimal solution, since in that case no time is wasted on detecting the deadlock and no components need to be restarted. Deadlock can be prevented by design. In the SOC-framework two types of application designers are distinguished. The component designer cannot resolve deadlock, since this would not be according to the SOC-principle. Therefore resolving deadlock is left to the distributed system designer.

Deadlock can occur when components need to be blocked. There are two cases where components can be blocked:

- Transactions: the Transaction Manager manages the transaction and the danger of deadlock.
- Mutual exclusion: Deadlock can occur if cyclic calls are possible

Figure 5.5 gives an example of such a deadlock:

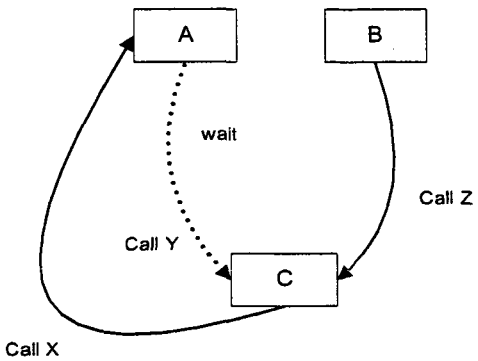


Figure 5.5

Call Y and Call Z are mutual exclusive. B starts with Call Z. Call Z contains among others Call X, which in turn contains Call Y. The cause of deadlock here is the presence of cyclic calls. If no cyclic calls are present, there cannot be deadlock due to mutual exclusive calls, since the circular wait condition does not hold.

Cyclic calls can be transformed into non-cyclic calls by splitting the component causing the cycle, as depicted in figure 5.6:

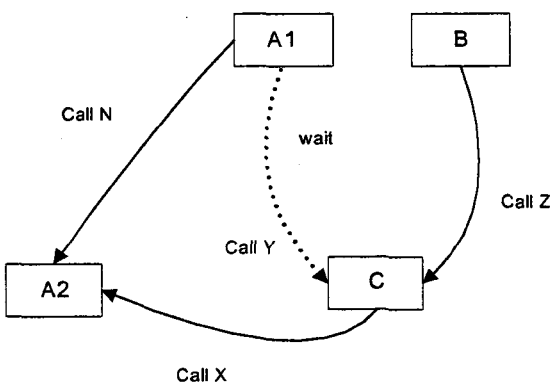


Figure 5.6

Comparing to figure 5.5, component A has been split into components A1 and A2. Apparently, A was not designed properly, and used a subcomponent that hierarchically did not belong on the level of A.

If cyclic calls cannot be avoided (e.g. when using callbacks), automatically the danger of deadlock is introduced. Deadlock can be detected by the incoming manager, since it handles all incoming calls for a component. Therefore it knows how long a call is already waiting. If the incoming manager uses a fair algorithm for executing calls, deadlock can be detected when a call is waiting for a very long time. This deadlock detection time can be specified in the configuration. To break the deadlock, the component can be

restarted. Components that have the danger of deadlock must then implement an interface supporting a function to restart the component. This mechanism to detect and break deadlock is currently not implemented within the SOC-framework.

5.4.6 Communication

When making calls between components within a distributed system using the SOC-framework, the incoming and outgoing manager will communicate with each other. This communication is done via the RPC-based communication mechanism provided by .NET Remoting, which is discussed in paragraph 5.3.3. From the viewpoint of the SOC-framework this is low level communication. Therefore the communication of two components within an application using the SOC-Framework is discussed. This communication uses:

- One-to-one communication and one-to-many: when using active replicas, one-to-many communication is established. Otherwise, one-to-one communication is used.
- Synchronous and asynchronous communication: component calls can be synchronous and asynchronous. The outgoing manager will be multi-threaded, and each call will reside within a separate thread. Within a thread, synchronous communication to another component will be used, i.e. this thread is blocked until it receives a reply.
- Two-way communication: after making a call on a remote component, a reply must be returned. Otherwise it can not be detected whether a call was successful or not. In case of 3-rd party components providing functions that do not return a value, the incoming manager returns a Boolean indicating whether the call was successful.
- Indirect communication: the incoming manager intercepts incoming calls before they are forwarded to the real component. Between the incoming and outgoing manager is communicated via direct communication.

5.4.7 Security

As stated in paragraph 3.9, security is important when distributed systems reside in an open environment. In this project, the focus is on distributed systems within a closed environment. Therefore security will not be an important issue within the framework. Nevertheless, it is worth mentioning that the .NET Platform provides a standard security mechanism. When hosting applications in Microsoft Internet Information Services (IIS), encryption, authentication, and authorization mechanisms are provided.

5.4.8 Component Lifetime / Activation

The SOC-framework supports the same combinations of component lifetime / activation as provided by .NET Remoting (Singleton, Single Call and Client Activated) and can be set in the configuration.

Regarding lifetime in the SOC-framework, a remote object lives as long as there is a client proxy for it. This is established by using the sponsoring mechanism of .NET Remoting (5.3.2). In the SOC-framework, the outgoing manager acts as sponsor for the incoming manager. Lifetime settings (initial lifetime, lifetime extension on call) for the remote objects can be set in the configuration.

5.4.9 Localization

The incoming manager is hosted as remote object via .NET Remoting, and therefore the SOC-framework uses the same mechanism to localize remote components as .NET Remoting (see 5.3.1). In the SOC-configuration, the localization settings are set in the SOC-framework configuration.

5.5 Implementation

This paragraph discusses the implementation of the SOC-framework. It starts with implementation issues of a general situation within an application developed according to the SOC-principle. Then the implementation of the individual components within the SOC-framework is discussed.

5.5.1 General View

figure 5.7 depicts the UML-model of a general situation within an application developed according to the SOC-principle.

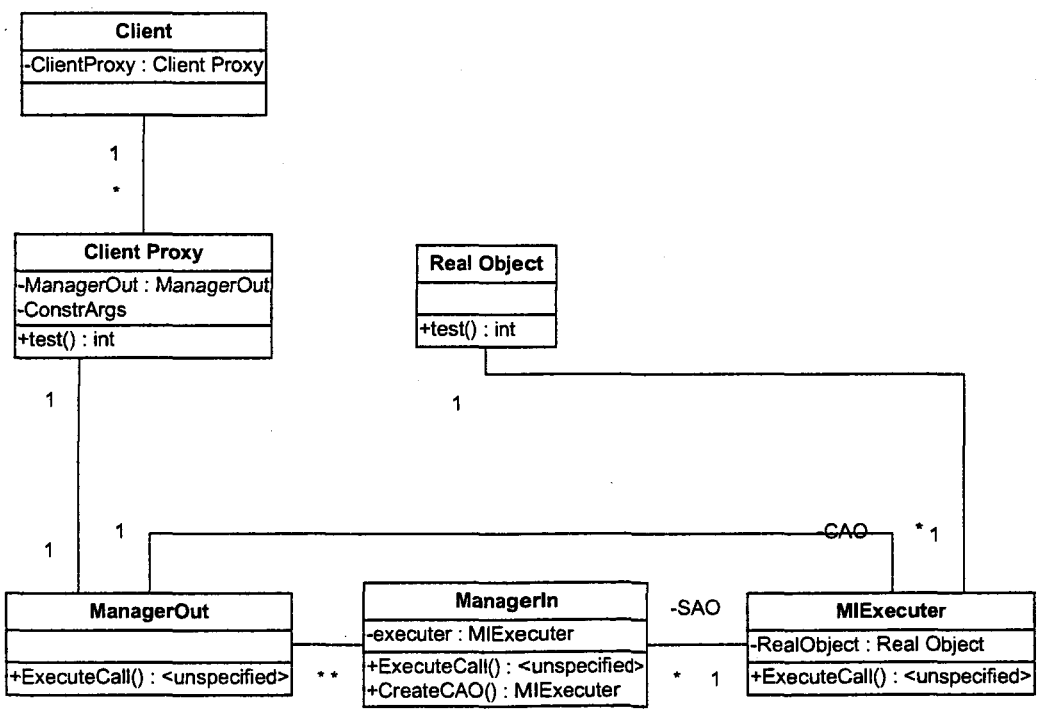


Figure 5.7: UML model of general situation within application developed using the SOC-framework

General

Client is the component that wants to make call *test* on the remote object *RealObject.* In a non-distributed environment, *Client* would directly approach *RealObject.* In a distributed environment using the SOC-framework, approaching *RealObject* needs four extra objects: *ClientProxy* is the proxy as discussed in paragraph 5.2.2, *ManagerOut* and *ManagerIn* are the managers as discussed in paragraph 5.2.1. *MIExecuter* is part of *ManagerIn* and takes care of the execution of the call on the real object.

Accessing the proxy

Client creates instances of *ClientProxy*, which is an object local to *Client*. *ClientProxy* is a proxy for *RealObject*, and therefore contains the same functions (i.e. they have the same interface, but since *RealObject* is a black box, it does not "implement" this interface explicitly). Here one function *test* is depicted. On construction, the constructor arguments (in case of a non-default constructor) are stored in *ClientProxy*.

Component Activation

RealObject is activated via *ManagerIn* and *MIExecuter*. *ManagerIn* itself is a Singleton remote object. If no instance of *ManagerIn* is available at the server, ManagerOut activates *ManagerIn* when making a call. On Creation of *ManagerIn,* an instance of *MIExecuter* is created.

The activation type of *RealObject* is read from the configuration. *RealObject* can be activated as:

- Singleton: *ManagerIn* itself is Singleton, therefore *ManagerIn* acts as pass-trough channel to *MIExecuter*. Since *MIExecuter* forwards the calls to *RealObject*, *RealObject* behaves as Singleton. The remote reference for *ManagerIn* is stored in the ManagerOut. *ManagerOut* is also sponsor for *ManagerIn*.
- Client-Activated Object (CAO): *ManagerIn* acts as factory for *RealObject* (need for factory is discussed in 5.3.2). It creates an instance of *MIExecuter* and returns it to *ManagerOut*. Now *MIExecuter* is only accessible through this specific instance of *ManagerOut*, and services one client which is in line with the definition of a CAO. *ManagerOut* sponsors *MIExecuter* and stores the remote reference.
- Single Call: actually a CAO *RealObject* is created, but no sponsor is registered and *ManagerOut* only makes one call on this object. This is established by not storing its reference.

Forwarding Calls

ClientProxy has one instance of *ManagerOut*, which provides function *ExecuteCall* that handles the distributed properties of the call and forwards the call to *ManagerIn* or *MIExecuter*. A call is passed trough the system as a string, an array containing the call parameters and possibly the constructor arguments (in case of a Client Activated Object). The constructor arguments are possibly later used to construct *RealObject* in *MIExecuter*, if *RealObject* is a Client Activated. At the end, *MIExecuter* executes the call on *RealObject*. Another approach for passing methods through the system is using the .NET *System.Reflection* namespace. This namespace provides a class *MethodInfo*, in which a method and its parameters for a specific type can be stored. Via the *Invoke* method of the *MethodInfo* class, this method can then be invoked on an object of that type. This approach ensures type safety, but is not usable within the SOC-framework, since *ClientProxy* does not know the interface of *RealObject*. Interfaces can not be used, since then the remote component must implement a specific interface. In that case, 3-rd party components can not be used if they do not implement that interface.

Hosting

ManagerIn is hosted on the remote computer as a windows service. Alternatives are hosting it as console application or hosting it in Internet Information Services. A console application has to be started manually and is visible on the screen. A windows service can start automatically and runs invisible. Internet Information Services needs to be installed on the computer and does not provide more useful features for this project than windows services.

5.5.2 ManagerOut

This part describes the architecture for the *ManagerOut*. figure 5.8 depicts the architecture:

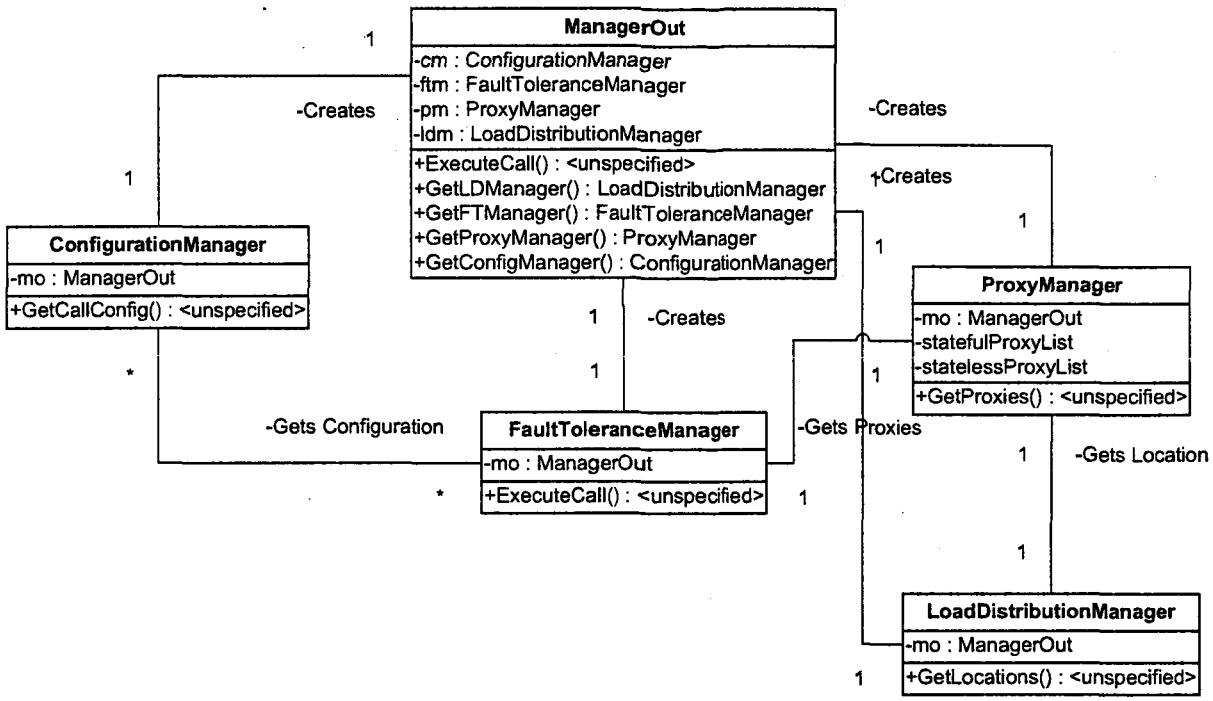


Figure 5.8 UML-diagram for the ManagerOut

The *ManagerOut* contains 4 submanagers: The *Configuration Manager* (CFM), *Fault Tolerance Manager* (FTM), *Load Distribution Manager* (LDM) and the *Proxy Manager* (PM). *ManagerOut* stores one instance of each manager. Via the *ManagerOut* the managers can communicate with each other.

When a call is executed on the *ManagerOut*, the call is passed to the *FTM*. figure 5.9 shows how a call is handled by the outgoing manager:

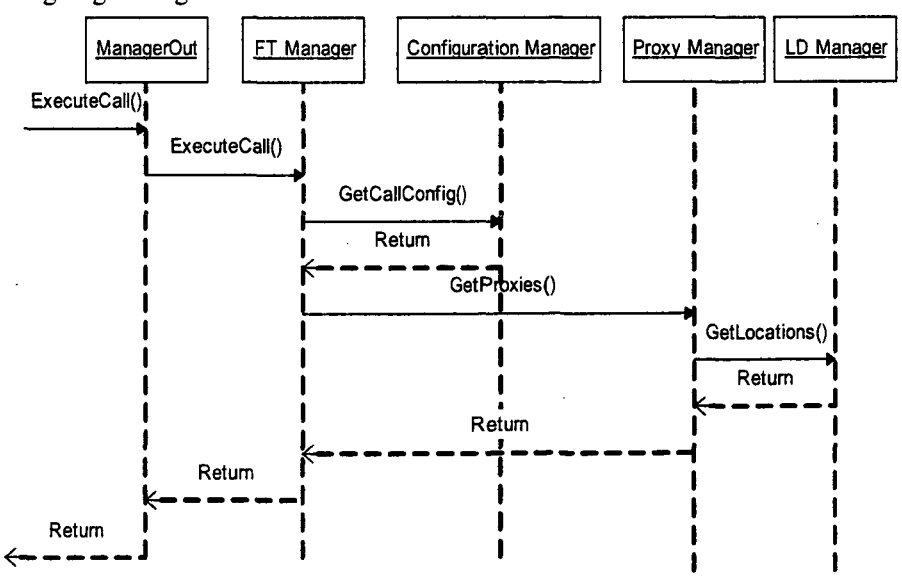


Figure 5.9 Sequence diagram of handling a call by the outgoing manager

The *FTM* first gets the configuration for the call. This configuration is among others passed as parameter in the next calls to the *PM* and *LDM*. The *FTM* gets the remote reference(s) (also called 'proxy(s)') on which to make the call. These proxies are provided by the *PM*. The *PM* contains a list of stateful proxies and a list of stateless proxies. If there are no proxies to make the call on, the locations to create new proxies are provided by the *LDM*, which returns the location(s) for the objects on which to make the call according to the configured load distribution algorithm. The *FTM* executes the call(s) in parallel. If at least one object returns a result, the call is returned to the ManagerOut. If no object returns a result within the configured timeout-time, this iteration of calls failed. Depending on the configuration it is possible to execute this call on other objects, or execute this call on the same object(s) again.

5.5.2.1 ManagerOut

The *ManagerOut* creates and stores the four submanagers. On creation it registers a channel. Via the ManagerOut, the four submanagers communicate with each other.

5.5.2.2 Fault Tolerance Manager (FTM)

One call can be executed on different locations, i.e. on different proxies. The *FTM* gets the proxies for a call from the *PM*. The *FTM* executes calls on these proxies in parallel in separate threads. Since .NET does not support the execution of functions with parameters in a thread, a thread-wrapper class is used. This wrapper is a self-created class that provides a constructor with the necessary parameters and a function to start the thread.

Thread-safe ArrayList storing the return values

Before the threads are started, an ArrayList (a standard .NET class representing a list) to store the return values of the threads is created. This list is given as parameter to each thread and the threads adds it result to the list when it is finished. Since the list is used by multiple threads, it must be thread-safe. .NET provides the possibility to acquire a thread-safe wrapper for an ArrayList. It must be noted that this thread-safeness is not valid when enumerating through an ArrayList. To guarantee thread-safeness in this occasion, the ArrayList must be locked (lock mechanism is provided by .NET).

Fault Tolerance mechanism

The *FTM* contains a fault tolerance mechanism which can be configured in the configuration file. To explain this mechanism, the following terms are first defined:

Iteration: execution of calls on a subset of the available proxies

Session: set of iterations

In the configuration, a maximum number of active replicas can be configured. Within one iteration the number of calls may not exceed this number. If there are more replicas available, it is possible to retry the call in another iteration if the previous iteration failed. The number of possible retrials can be configured in the property *MaxNrOfActiveReplicaRetrials*. If all possible replicas have been tried and failed, or the *MaxNrOfActiveReplicaRetrials* is reached, this session has failed. Depending on the configuration property *NrOfRetrialsOnFailure*, it is possible to retry the session. In case of passive replicas, an iteration only consists of one replica. The number of possible retrials is configured in the property *MaxNrOfPassiveReplicaRetrials* and the rest is the same as above. figure 5.10 depicts the above discussed Fault Tolerance mechanism.

Session											
Iteration			*Iteration*			*Iteration*			*Iteration*		
Call	Call	Call	Call	Call	Call	Call	Call	Call	Call	Call	Call

Figure 5.10 Fault Tolerance mechanism: using sessions, iterations and calls

In definition:

MaxNrOfActiveReplicas: the maximum number of parallel called objects within one iteration

MaxNrOfActiveReplicaRetrials: the maximum number of iterations within a session

MaxNrOfPassiveReplicaRetrials: the maximum number of iterations within a session. Since one iteration always consists of one call, this can be read as the maximum number of sequential calls within a session.

NrOfRetrialsOnFailure: maximum number of sessions

If an iteration was not successful, the location of the proxies used are added to a list. This list contains locations that are already used within a session. The LDM does not return locations from this list.

Stateful Calls

Note that in case of a stateful call, all these properties only make sense if no stateful proxies exist. In that case new proxies must be created, and that will be done according to the configuration. If stateful proxies exist, a stateful call must be executed on all stateful proxies since this call depends on previous stateful calls and later calls depend on this call. Two stateful calls may not execute in parallel, since it is not known in advance whether the "parallel" called proxies are successful.

Failed Calls

To detect whether an individual call has failed, the configured timeout time is used. If no result is returned within the configured timeout, the iteration has failed. When a stateless call iteration fails, all used proxies are removed from the list containing the stateless proxies. In case of a stateful call iteration, this iteration consists of calls on all stateful proxies. These proxies may not be removed from the list of stateful proxies, since this list would be empty in that case. Other sessions must be tried to obtain a result. If all sessions fail, i.e. the complete call fails, the stateless as well as the stateful proxies are removed from the proxy lists and an exception is thrown.

Successful Call

If a call is successful, it is still needed to remove the non-successful proxies. Removing these failed proxies is done in a separate thread. Since it is not necessary that the timeout has been expired for the successful call, the removal-thread must first wait until the timeout for all calls has expired. When a stateful call is made, it is possible that the timeout for the possibly failed previous stateful calls has not been expired. Since for a stateful call only proxies may be used on which all stateful calls have been successfully executed, the proxies that did not return a result on the previous stateful call must be moved to the stateless proxy list. Therefore the list of used proxies and the list of successful proxies from the last stateful call are always stored.

5.5.2.3 Proxy Manager (PM)

The *PM* stores two proxy lists: the statefulProxyList and the statelessProxyList.

Proxy p ε statefulProxyList ≡ ≥ 1 stateful call has been executed on p and no stateful calls on p have failed
Proxy p ε statelessProxyList ≡ no stateful call has been executed on p or a stateful call on p has failed

Both lists are used in a multi-threaded environment, and therefore the same thread-safe ArrayList mechanism is used as described in the fault tolerance manager. The proxy lists store a proxy and the location. The location is stored to check whether a proxy already exists for a location.

Getting Proxies

The *PM* provides a function to get a list of proxies for the *FTM* given a call configuration, constructor arguments and a list of forbidden locations. If the call for which the proxies must be returned is stateless, there are no restrictions on what proxies to return. The locations of the proxies are picked by the LDM. It is checked which locations already have a proxy in the stateful and stateless proxylist. For the locations that do not have a proxy yet, proxies are created. If the activation type of the call differs from *SingleCall*, it is added to the stateless proxylist. If the call for which the proxies must be returned is stateful, there are two cases:

1) The statefulproxylist is empty. New proxies must be created for the given locations and added to the statefulproxylist. If stateless proxies already exist for one of the locations, these proxies must be moved to the statefulproxylist.

2) The statefulproxylist is not empty, this call must be executed on all items from this list, since this call can depend on earlier (stateful) calls.

Creating Proxies

Given a list of locations, the activation type of the called object and the constructor arguments (for a CAO) the *PM* generates proxies for the given locations. When the activation type is Singleton, the proxy is a proxy for the *ManagerIn*. When the activation type is CAO or Singlecall, the *ManagerIn* acts as factory and a proxy for the *MIExecuter* is acquired. For a CAO and Singleton proxy, the *PM* is also directly registered as sponsor for the remote object.

5.5.2.4 Load Distribution Manager (LDM)

The load distribution manager contains a function that for a given method returns the list of locations for executing the call. Currently one LD-algorithm is implemented: Round Robin.

5.5.2.5 Configuration Manager (CFM)

The CFM reads the XML configuration file in a standard .NET class for storing data (*DataSet*). The configuration file is checked on the proper format by a schema, which is included in the assembly as a resource. The configuration is stored in a struct and is passed as result when the configuration is requested by another manager. The CFM uses a standard .NET file system watcher to detect whether the configuration file has been changed. If the configuration file is changed, the CFM re-reads the configuration.

5.5.3 ManagerIn

This part describes the architecture for the *ManagerIn*. Figure 5.11 depicts the architecture:

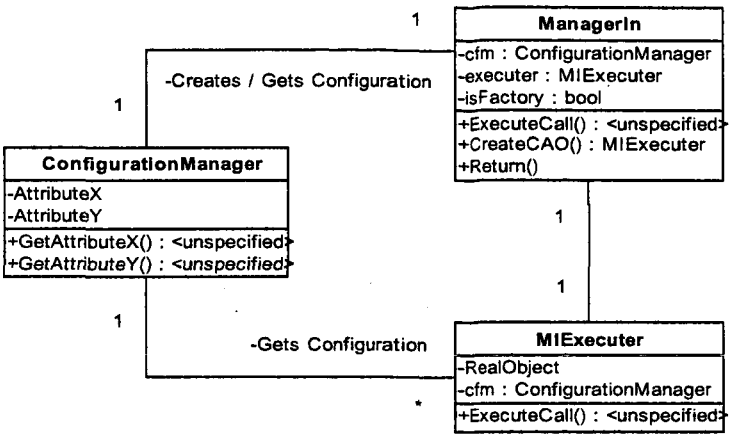


Figure 5.11 UML-diagram for the ManagerIn

The *ManagerIn* has two roles and therefore there exist also two types of architectures. The role the ManagerIn has is stored in a Boolean and is read from the *CFM* when the *ManagerIn* is constructed.

If the *ManagerIn* acts as factory, it does not contain its own *MIExecuter* and it is not possible to forward calls to the *RealObject*. It is only possible to act as factory and create and return a *MIExecuter*. The *ManagerOut* that asked for the CAO makes directly calls on the *MIExecuter*. If the ManagerIn does not act as factory, it has its own *MIExecuter*, and forwards calls to it.

Figure 5.12 shows the sequence diagram of a call in two different roles:

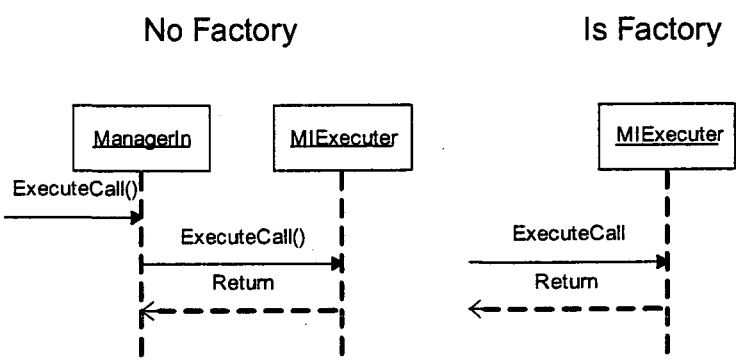


Figure 5.12: Sequence diagram for the two roles of the incoming manager

On construction, the MIExecuter dynamically loads the real object on which to make the calls. The information needed for this is the AssemblyName and the AssemblyType of the real object. This information is read from the configuration.

Mutual Exclusiveness

The function *ExecuteCall* makes the call on the real object. But before execution, this function checks whether no mutual exclusive functions are active. The mechanism used for dealing with mutual exclusive functions is the following:

Initially, the call and the call's thread are added to a central list. If there are calls higher in the list which are mutual exclusive with this call, its thread is suspended. There is one moment on which threads can be resumed: when another call has finished, it is removed from the central list and non-mutual exclusive threads are resumed. This is done by checking whether the calls in the central list are still blocked or not. If not, its thread is resumed. This algorithm is fair, since it first releases the highest item in the call list.

There are three possibilities for a call being mutual exclusive:

- The entire object is mutual exclusive, i.e. only one function may be executed at a time. In this case, only the highest call (i.e. with index = 0) may be executed
- A function is "selfmutex", i.e. only one instance of this function may be executed at a time. In this case must be checked whether for a call a higher call exists which represents the same function
- The function is mutual exclusive with another function. In this case must be checked whether for a call a higher call exists which represents a mutual exclusive function.

5.5.4 ClientProxy

The *ClientProxy* is not a standard component available within the framework. The distributed system expert must provide a *ClientProxy* for each remote component to be accessed within the distributed system. In the constructor, the constructor arguments are stored in a global variable and an instance of the *ManagerOut* is created. The client proxies must contain all functions provided by the remote component for which it is a proxy. The implementation of these functions consists of forwarding the call to the *ManagerOut* and returning its result to the component that called the *ClientProxy*.

As long as the client proxy is still in the scope of the component that holds the instance of the client proxy, its *ManagerOut* provides sponsoring for the remote components used by this instance. When the *ClientProxy* is garbage collected, its *ManagerOut* will also be garbage collected, and automatically the sponsor for the remote components used by the garbage collected *ClientProxy* is removed. In this way the lifetime behavior as mentioned in 5.4.8 is established.

5.6 Configuration

As mentioned in paragraph 5.1, the SOC-framework must be configurable. The configuration of the SOC-framework will be discussed in this chapter.

5.6.1 Locus of Configuration

A configuration of a distributed system can be managed on different levels:

- Per function
- Per component
- Per server

Per function:
Each function of a component will have its own configuration manager. This will result in a lot of configuration managers, and is probably a too detailed way. It will loose the overview.

Per component:

Each component will have its own configuration manager. This is the most natural way for an object-oriented architecture and is implemented in the SOC-framework.

Per server:

Each server will have its own configuration manager. A central configuration will not make a better overview, since this file will be very large when running several components on one server. Since one component does not need to know the configuration of another component, it does not have an added value when creating a central configuration file.

5.6.2 Configuration Files

The configuration can be set in XML-configuration files. The SOC-framework uses server object and client object configuration files. The .NET Remoting Framework already uses XML configuration files for storing the configuration of the remote objects. These configuration files store:

- Server / Client properties
- Channel properties
- Lifetime properties

These are properties for how to localize remote objects, how they are activated and how long they stay alive. Since these are not all the properties we need for the architecture, it must be possible to configure extra properties. Examples of server and client configuration files can be found in Appendix B.

5.6.3 Configuration Properties

The SOC-framework can be configured on the following properties:

Server Object:

Name: Name of the hosted object.

ActivationType: How is this remote object activated (CAO, SINGLETON, SINGLECALL)

Channel: Type of channel through which is communicated with clients (TCP, HTTP)

Port: Port on which remote object is hosted

Endpoint: End point on which remote object is hosted

Lifetime - InitialLeaseTime: Initial time that the remote object is alive (in seconds, 0 = forever)

Lifetime - RenewOnCallTime: Time that a lease is extended when a call is received (in seconds)

RealObject - AssemblyName: Name of the assembly of the real object (without .dll extension)

RealObject- AssemblyType: Full type of the real object (including Namespace)

Mutex - All: Bool indicating whether all functions are mutual exclusive

Mutex - GroupFunctions: Functions within this group are mutual exclusive

Mutex - SelfMutexFunctions: List of functions from which only one may be active at a time (i.e. they are mutual exclusive with them selves)

Client Object:

Name: Name of the remote object.

Channels: Channels used to access remote object(s). In case of replication, multiple channels may be necessary

ActivationType: How the remote object is activated

SponsorRenewalTime: Time to extend the lease of the remote object if a sponsor request is received (in seconds)

Locations: The locations on which calls on the remote object can be done

Call - Name: Name of the call

Call - LDAlgorithm: The Load Distribution Algorithm used to pick a location

Call - PollTime: Sleeptime between two checks on whether a function returned a result (in seconds)

Call - State: State of call (STATEFUL, STATELESS)

Call - ReplicaPolicy: Replica policy of the call (ACTIVEREPLICA, PASSIVEREPLICA, NOREPLICA)

Call - TimeOut: Time until this call times out if no result is returned within this time (in milliseconds)

Call - MaxNrOfActiveReplicas: Maximum number of active replicas

Call - MaxNrOfActiveReplicaRetrials: Maximum number of active replica retrials

Call - MaxNrOfPassiveReplicaRetrials: Maxium number of passive replica retrials

Call - NrOfRetrialsOnFailure: Number of retrials on failure

6. Test Results

This chapter discusses the test results. The SOC-framework has been tested by developing a distributed application with it. First this distributed test application is discussed. The test application revealed some problems, for which an extension on the SOC-Framework was needed. Following on this paragraph this extension is discussed. Then the validation of the SOC-Framework is discussed and finally the performance is handled.

6.1 Test Application

The application is part of the Survey .NET Project within Imtech. This project has been started up in 2002 to get experience with the Microsoft .NET technology. The project consists of several applications, which together form a suite to support the course of developing, maintaining, filling in, and reporting a survey. Figure 6.1 depicts the overview of the Survey .NET Project.

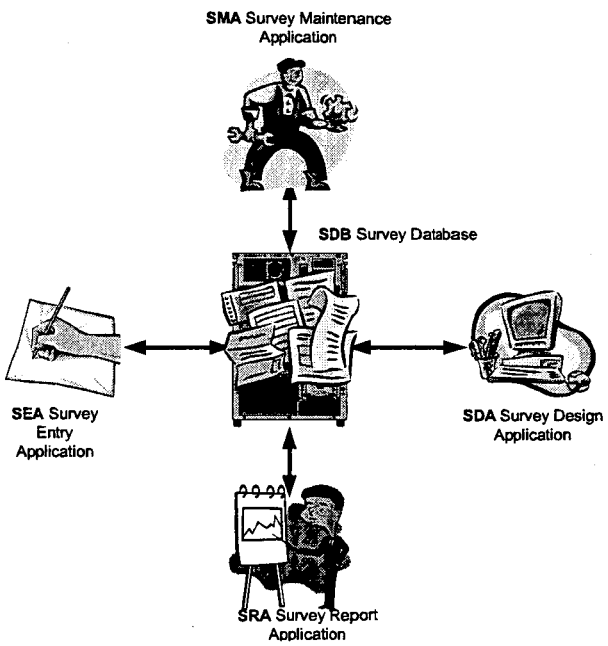


Figure 6.1 Overview of the Imtech Survey .NET Project

The application used to test the SOC-framework is the Survey Report Application (SRA). The customer requirements for the SRA can be found in appendix C. Main requirements for the SRA are:

- It must be possible to show the results of a survey
- It must be possible to print the results of a survey
- It must be possible to export the results of a survey to XML
- It must be possible to store and load a predefined view on the results of a survey (called report layout)

The database model for a survey and its results were already designed for the Survey Entry Application (SEA, the application to fill in a survey) and can be found in [Sch]. The SRA is a windows forms application, which reads the survey data from a SQL-Server database.

Developing the SRA

The SRA has been developed accordingly to the process of developing a distributed system as described in paragraph 4.1:

Step 1

The first step was to design the distributed architecture and specify the components within the architecture. The specification of the components has been done by specifying the functions that a component must provide within the detailed architecture UML diagram. The detailed architecture can be found in appendix A. A global view of the architecture is depicted in figure 6.2.

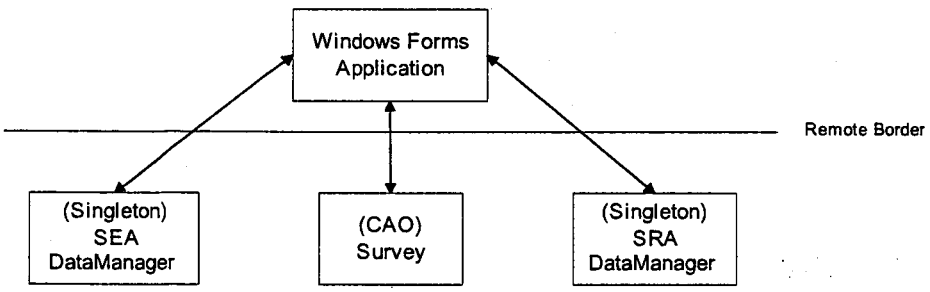


Figure 6.2 Global architecture of SRA

The architecture is globally split up into 4 components:

- The client side (windows forms / user interface)
- The Singleton server side SEA Data Manager (manages the data of the SEA, i.e. the results of a survey)
- The Singleton server side SRA Data Manager (manages the data of the SRA, i.e. the report layouts of a survey)
- The Client Activated server side Survey Representation, using SEA Data as well as SRA Data

The remote border is between the client side component and the three server side components.

Step 2

Second step of the process was to acquire the components. All components were implemented by the component developer. After implementing the separate components, these components and the entire application were locally tested. These tests were successful.

Step 3

As the third step, the components were put in a distributed environment and the system configuration was set.

While developing and running the test application, the following problems were discovered:

Problem:

In the application instances of standard .NET Framework classes should be returned by value. The distributed system expert researched in advance which .NET Framework classes were suitable for this and specified these classes for some functions as result object. There were classes that were marked as serializable by the documentation, but were not serialized in practice. Reason for this was that they also inherited from MarshalByRefObject, resulting in .NET Remoting to create remote references for them.

Solution:

Use a custom class that describes the .NET class. An example of a class that had above problem was the Font class. By introducing a custom serializable DFont class, describing the properties needed to construct a Font, the font could be returned by value.

Problem:

a) When a remote component returns an object as remote reference, this object is out of the scope of SOC-framework. The system does work, but the distributed properties for the returned remote object are not handled anymore by the SOC-framework.

b) When a remote component returns an non standard .NET object as remote reference, this object must inherit from a specific .NET class (MarshalByRefObject) to enable remote access to it. In this way 3-rd party components returning objects that do not inherit from MarshalByRefObject can not be used remotely.

Solution 1: Always return serializable objects as results. In this way result objects are forced to execute locally, which is not the idea behind a distributed system. Moreover, not all objects can be serialized (if they contain non-serializable attributes) and 3-rd party components can return objects that are not serializable. Therefore this is not an acceptable solution.

Solution 2: Put the result object in the scope of the framework by using a proxy for the result, and use the incoming and outgoing manager as discussed previously to handle the distributed properties. This solution requires the SOC-framework to be extended. Details of this solution and the SOC-framework extensions can be found in the next paragraph.

6.2 Extending the SOC-Framework

As described in the previous paragraph, the SOC-Framework as described previously does not deal correctly with objects that are returned as result. This paragraph discusses the architecture and implementation issues for returning result objects within the SOC-framework.

6.2.1 Architecture

When a call is executed on the remote black box component, the result is returned and finally reaches the black box component that initiated the remote call. There are two ways in which the result can be returned:

- By value
- By reference

Returning result objects by value

If the result is returned by value, it must be serialized over the remote border. The object is now locally copied, and has no remote issues anymore. Therefore the SOC-framework is not further necessary to use this result object. Using a serialized copy is depicted in figure 6.3:

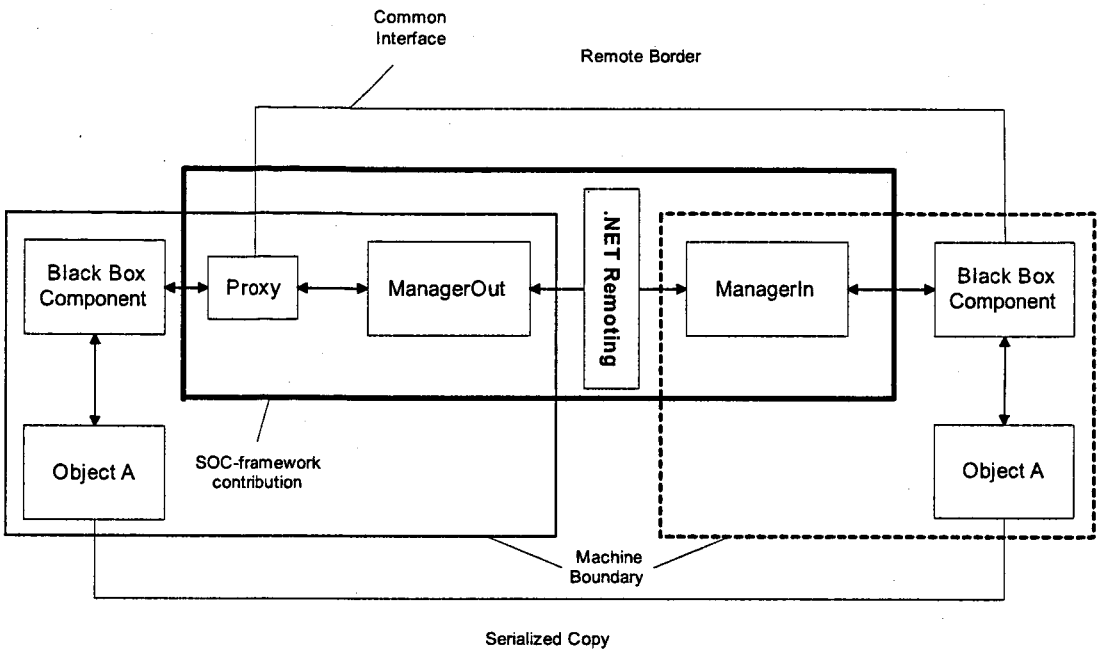


Figure 6.3 Using a serialized copy of a result object

Disadvantages of this approach are:

- No remoting is used anymore. Operations on the object are locally executed, which is not the idea of a distributed system. Furthermore, it decreases the possibility to work object-oriented, since it is a copy of the original object.
- The result object must be transported over the remote border, which is often a time expensive operation.
- The class definition of the result object must be available at the local component side.

If the result object is an instance of a standard .NET framework class, the class definitions are available at both sides of the remote border, since they both use the .NET framework. Classes that are marked as serializable are suitable to be automatically serialized by .NET Remoting. Custom build classes within .NET can be made serializable by marking them serializable.

Returning result objects by reference

If the result is returned by reference, the result object is a remote object. The result object can now be seen as a new black box component within the distributed system, and the SOC-framework is needed to handle its distributed properties. This is depicted in figure 6.4:

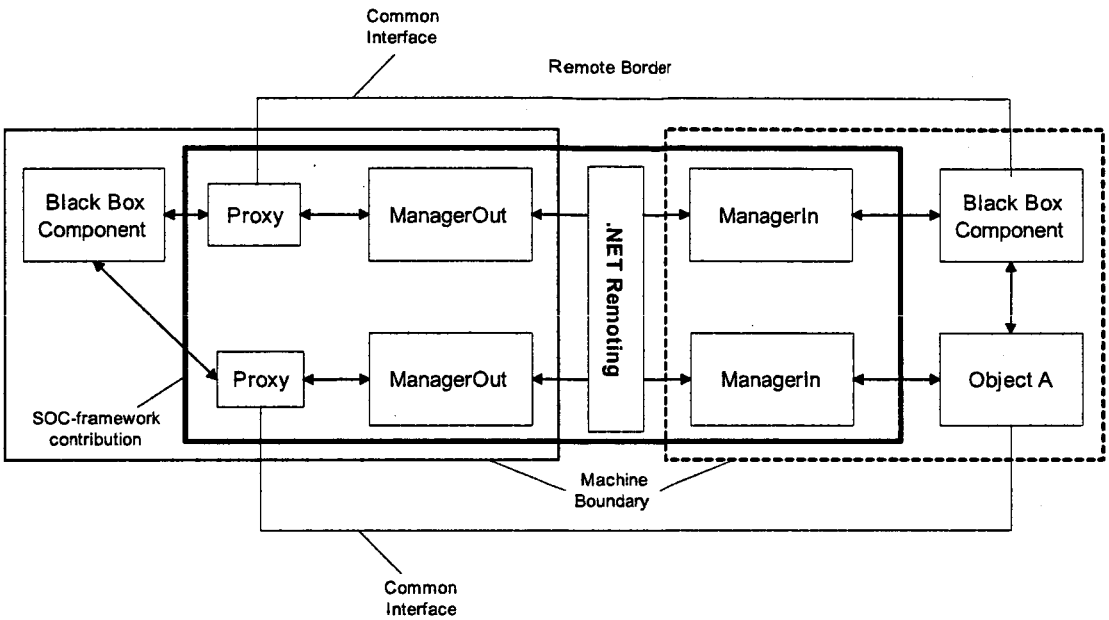


Figure 6.4 Accessing the result object remote using the SOC-framework

Result objects can recursively produce other result objects. At the end, there will often be a result that is serialized over the remote border, since this result is then locally used (e.g. an integer / string / standard .NET framework class).

It is also possible to pass a proxy as a result object. In that case the proxy and its outgoing manager must be serialized to the remote component requesting the proxy. In this way references to remote components can be distributed through the distributed system.

If several redundant components are available for a component, it is possible to have several redundant result objects. This situation is depicted in figure 6.5.

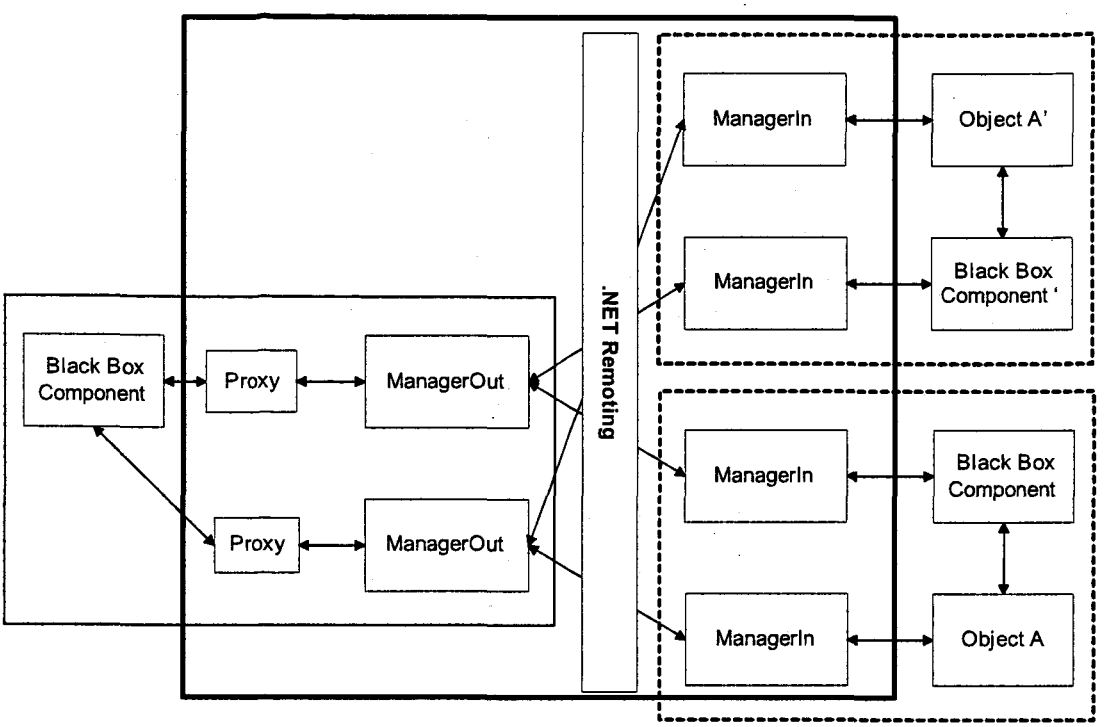


Figure 6.5 Architecture in the presence of redundant components

In this case the outgoing manager of the proxy for the result object must store the remote references for the result object.

6.2.2 Implementation

To implement this extension, following extensions within the implementation SOC-framework are needed.

Different call types: By Value or By Reference

A call can return an object by value or by reference. In case of a by value call, the incoming manager returns the serializable object. The outgoing manager only needs to wait for the first available result since the by value call returns a serialized object. If a call is a by reference call, the incoming manager creates an instance of *MIExecuter* (see 5.5.3) with as *RealObject* (see 5.5.3) the result. Since *MIExecuter* inherits from MarshalByRefObject, the result is now remotely accessible. The outgoing manager waits for the possibly several results and returns a list of a specified number (read from configuration) of remote references to the incoming managers for the result objects.

Proxies

Depending on the type of call, a proxy can now receive a serialized object or a list of remote references from the outgoing manager. In the first case, the serialized object can be directly returned to the component requesting the call. In the second case the proxy has to create a new proxy for the results. This new proxy contains a constructor taking as arguments a list of remote references. On construction it creates an outgoing manager and sets the remote references in the stateful proxy list of it. The outgoing manager also sponsors the remote references. The outgoing manager may not create new remote references to the object, since this object is acquired by result, and not by creation. Therefore the new activation type *Result* is introduced, which can be read from the configuration.

Making Proxy class and outgoing manager serializable

To pass a proxy as a result object, the proxy and the outgoing manager must be serializable. A proxy only contains an outgoing manager that must be serialized. Therefore it is sufficient to mark the proxy class as serializable. Now .NET automatically serializes and deserializes the proxy class and all its members. The outgoing manager contains more members to serialize and serialization an deserialization are not standard here. Therefore a special serialization and deserialization function are implemented (.NET uses these special functions for serialization and deserialization if a specific interface *(ISerialization)* is implemented). On deserialization of the outgoing manager, a channel is registered to enable communication. From the members of the outgoing manager, the configuration manager (*CFM*, see 5.5.2.3) and the proxy manager (*PM*, see 5.5.2.5) need special serialization and deserialization functions. On deserialization the *CFM* reads the configuration of the component requesting the proxy. This is necessary, since the configuration of a proxy can differ per component it belongs to. The *PM* must serialize and deserialize its lists of proxies. To serialize a remote reference, this reference can be converted to a serializable representation by marshalling it in .NET (the serialized representation is called *ObjRef*). When deserializing the *ObjRef*, the *ObjRef* can be unmarshalled to a remote reference. Remarkably was that .NET automatically unmarshalled the *ObjRef* on deserialization.

Pilot version

The implementation of the extensions as discussed above was successfully made. Since there was limited time to design and implement these extensions, the implementation can not be seen as optimal. This pilot version shows that the architectural issues as mentioned in 6.2.1 are technically feasible. Parts of the test application were adapted to the extended SOC-Framework and worked correctly.

6.3 Validation of SOC-Framework

This paragraph describes the validation of the SOC-Framework. The first two paragraphs of this chapter validated that the SOC-Framework was suitable to build a distributed system with. This paragraph further validates each property of distributed system mentioned in chapter 3. For this validation the non-extended version of the SOC-Framework is used, since the extended version is a pilot version. The difference between the two versions of the SOC-Framework only lies in the way in which is dealt with results. The core functionality of the SOC-framework is the same in both versions, i.e. dealing with the distributed properties between components. Therefore the validation of the non-extended version is also valid for the extended version of the framework. Now the validation of each property is discussed:

Dependability:

Test Case: Remote components are hosted on two different computers. The maximum numbers of replicas is configured on 2. The application is started.

Action: Shut down computer one.

Result: Distributed application remains up and running.

Action: Restart computer one, shut down computer two.

Result: Stateful calls fail. Stateless calls succeed.

Conclusion: This validation is successful.

Load Distribution:

Test Case: Remote components are hosted two computers. The maximum numbers of replicas is configured on 1. The load distribution algorithm is configured on "Round Robin". The application is started.

Action: Perform action that requires a stateless call twice.
Result: Call is executed exactly once on both computers.

Conclusion: This validation is successful.

Mutual Exclusion:

Remote components are hosted on one computer. Two instances of the application are started.

Test Case 1: A specific function is configured to be mutual exclusive with another function.

Action: Let both applications perform an action so that both functions are called.
Result: Calls are executed sequentially (Check the log)

Test Case 2: A specific function is configured to be self mutual exclusive.

Action: Let both applications perform an action requiring a call to this function.
Result: Calls are executed sequentially (Check the log)

Test Case 3: A specific component is configured to be completely mutual exlusive

Action: Let both applications perform an action so that two functions on that component are called.
Result: Calls are executed sequentially (Check the log)

Conclusion: This validation is successful.

Transactions:

Transactions are not supported by the SOC-framework, and are therefore not validated.

Deadlock:

Deadlock is currently not implemented within the SOC-framework, and is therefore not validated.

Communication:

Communication is tested by previous test cases, where remote function calls were done.

Security:

Since supporting security was not part of the goal of the project, the SOC-Framework does not support security. Note that .NET in combination with Internet Information Services provides standard support for security.

Component Activation:
Remote components are hosted on one computer. Two instances of the application are started.

Test Case 1: A specific component is configured as Client Activated Object.

Action: Perform an action in both applications resulting in the creation of the component.
Result: There are two instances of this component created (see the log)

Test Case 2: A specific component is configured as Singleton.

Action: Perform an action in both applications resulting in making calls on the component.
Result: There is only one instance of this component created (see the log)

Test Case 3: A specific component is configured as Single Call.

Action: Perform an action in both applications resulting in making calls on the component.
Result: For each call, an instance of the component is created. At the end, no instance of the component exists anymore.

Conclusion: This validation is successful.

Component Lifetime:
Test Case: Remote components are hosted on one computer. The application is started. The component is configured as Client Activated or Singleton.

Action: Perform an action in the application resulting in the creation of the component.
Result: As long as the local client proxy is in the scope of the local application, the remote object is not destroyed.

Localization:
Localization is tested by previous test cases, where remote components were created and thus localized.

6.4 Performance

On the one hand, the SOC-Framework decreases the performance, since it is an extra layer between the components within a distributed system. On the other hand, the SOC-framework can also increase performance, since there are mechanisms that have a positive influence on the performance, e.g. the mechanisms providing redundancy and load distribution. It is very difficult to measure the performance win / loss of the SOC-Framework when using the test application, since it is only a small distributed system. In that case statistics about the average call time (in general or per call) are not very reliable. To be able to reliably measure the performance of the SOC-Framework, the performance of the SOC-Framework should be measured in a large heavily loaded distributed environment and be compared to a version of this heavily loaded distributed system without using the SOC-framework. Since the test application is not suitable to measure the performance, no performance measurements are made for the SOC-Framework. When executing the test application in combination with the SOC-Framework, the general impression of the performance was acceptable.

7. Conclusion

This chapter first describes the conclusion of the project. Then several recommendations are given.

7.1 Conclusion

This paragraph provides the conclusion for this project.

Accomplishing the goal

When comparing the results of the project to the goal of the project (paragraph 2.2) can be concluded that the goal has successfully been accomplished. The separation of concerns provided by .NET Remoting has been extended in many ways: a separation of concerns has been established for all properties needed for distributed Imtech applications. Table 7.1 lists per property whether it is supported by .NET and whether it is supported by the SOC-Framework in combination with .NET.

Property	.NET	SOC-Framework + .NET
Dependability	-	+
Load Distribution	-	+
Mutual Exclusion	-	+
Transactions	+ **	+ **
Deadlock	-	+ *
Communication	+	+
Component Activation / Lifetime	+	+
Localization	+	+

Table 7.1 Support for distributed properties for distributed Imtech applications

* Currently not in the implementation of the SOC-framework

** By using standard transactional mechanisms

By developing the test application with the SOC-Framework according to the SOC-development process (paragraph 4.1) has been showed that establishing a separation of concerns was technically feasible. It also revealed that building distributed systems with the SOC-Framework works very well. There are many advantages of developing distributed systems using the SOC-principle (see paragraph 4.2) and they are all very useful in practice.

Process of developing distributed system

We have shown that establishing a SOC is technically feasible, and now the process of developing a distributed system using the SOC-principle needs to be worked out in more detail:

- There are no changes needed in the development of a distributed architecture, but the specification of the components within a SOC-developed distributed system needs an extension: the distributed system expert needs to specify for the component developer which components are actually executed remotely. Although components use local proxies to reach eventually a remote component, the actual call is done

over the remote border. Therefore this call can take more time and always has the possibility to fail (if all redundant components fail).

- When developing a component, extra information about the component is needed for the distributed system architect (e.g. which calls are stateful). This information must be specified by the component developer.
- When configuring the system, the distributed system expert uses among others the information provided by the component developer.

SOC-Framework Distributed Applications

Applications that are especially suitable for using the SOC-Framework are large heavily used distributed applications, developed in large project teams. For these applications the strength of the SOC-Framework can be used optimally: advantages as overview, structured way of developing, error detection, logging/maintenance and flexibility are very useful here. Obviously, for building very small distributed applications in small development teams the SOC-Framework can also be used. But then the advantages of using the SOC-Framework are less, and in that case it must be considered whether the advantages of using the SOC-Framework are more important than the possible disadvantages (e.g. performance).

SOC-Framework in general

When taking this project into a more general view, it is interesting to know how generic the idea of the SOC-framework is. When looking at the architecture of the SOC-Framework (paragraph 5.2) it can be seen that all building blocks of the architecture are generic, except for the *.NET Remoting* building block. This is the part of the architecture that deals with the interaction between components over the remote border. By replacing this building block by a generic *Remote Border Interaction* block, the entire architecture has become completely generic. An example of another *Remote Border Interaction* block than .NET Remoting is CORBA (Common Object Broker Architecture) [Tan].

7.2 Recommendations

This paragraph provides the recommendations for this project.

Management Application

Applications using the SOC-Framework have several configuration files, components, managers and client proxies. To make it easier for the distributed system expert to create and maintain a distributed application, an application managing all items of the distributed system would be very useful. This application could store the versions of the components, automatically generate client proxies for the components, store configuration files, and provide an easier user interface to configure the properties of a system than directly adapting the configuration files.

Establish a consortium

The information about a component needed by the distributed system expert must be provided by the component developer. Since components can be used within different systems, it is very useful to standardize the component information. If there is such a standard, all components providing this standard information can be used with a SOC-framework implementation using that standard. To realize such a standard, a consortium could be established. Another task for this consortium could be to build and provide different standard implementations of the SOC-Framework, e.g. a .NET version, a CORBA version etc..

8. References

[Bal] Dusan Balek – *Connectors in Software Architectures*, Charles University, Faculty of Mathematics and Physics Prague, 2002

[Bus] Frank Buschman et al. – *Pattern-Oriented Software Architecture Volume 1*, Wiley, 1996

[Gam] Erich Gamma et al. – *Design Patterns*, Addison Wesley, 1995

[Sch] Werner Schram – *Stageverslag Survey .NET*, Mountside Software Engineering, 2002

[Sil] Abraham Silberschatz et al. – *Database System Concepts*, McGraw-Hill, 1997

[Tan] Andrew Tanenbaum et al. – *Distributed Systems*, Prentice Hall 2002

[Ver] Paolo Verissimo, Luis Rodrigues – *Distributed Systems For System Architects*, Kluwer Academic Pusblishers, 2001

[Wu] Jie Wu – *Distributed Systems Design*, CRC Press, 1999

Appendix A: Survey .NET Application

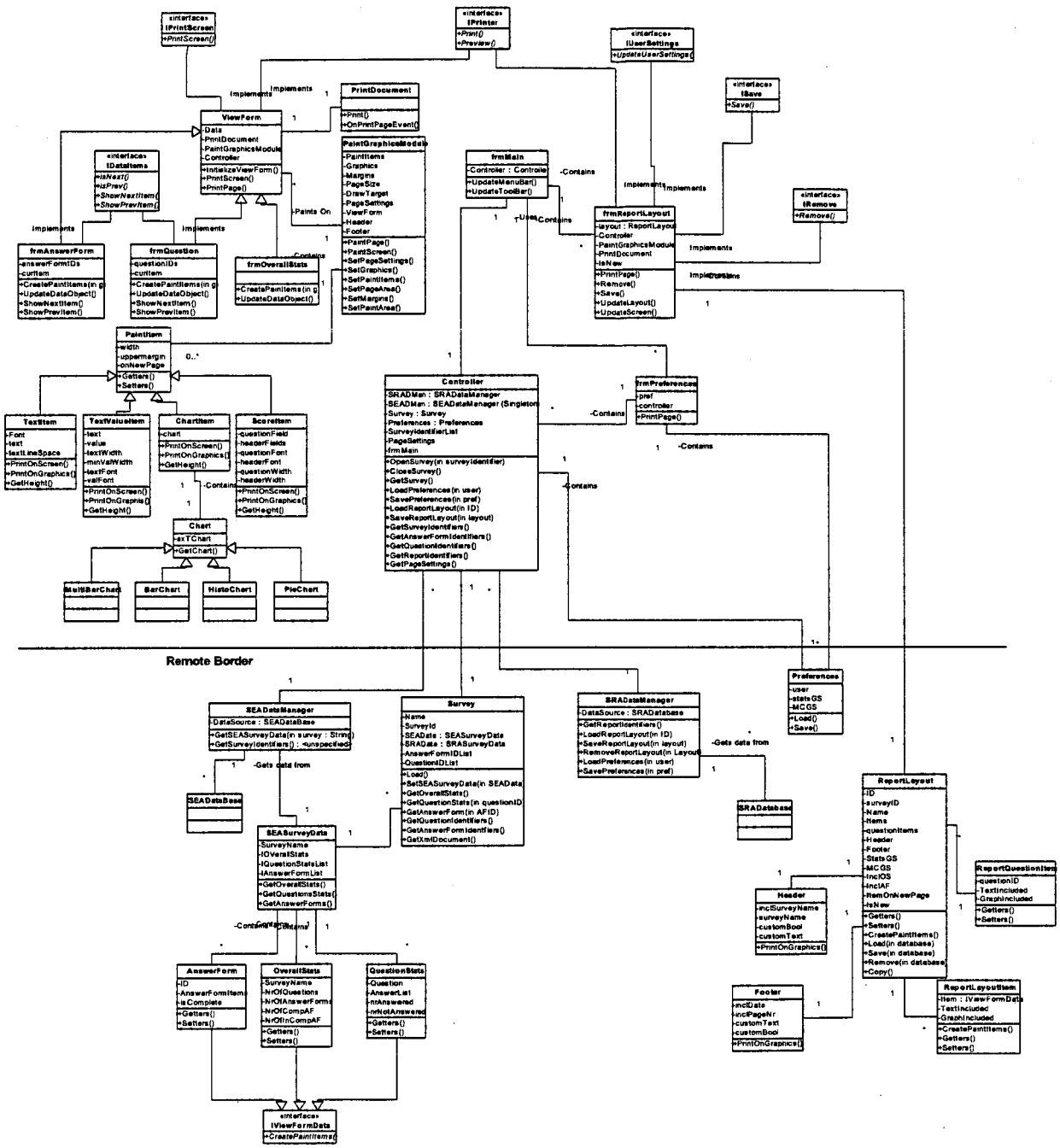


Figure A1 UML model for the SRA

Survey Data

The following data must be extracted from the database:

Overall Statistics

- Number of Questions
- Number of Answerforms
- Number of Complete answerforms (all available required questions must be filled in)
- Number of Incomplete answerforms (not complete)

The number of answered and unanswered questions is not showed, since this number is hard to define in a relevant manner. If an answerform is partially filled in, it is not defined what the number of remaining questions is, since it is possible to have different paths through the questions (in case of MC_Questions).

Statistics per Question

- Number of Questions answered (for which the answer was not empty)
- Number of Questions not answered (for which the answers was empty)
- Answers to the question
- Aggregated results of the question

Answerform:

- Questions and answers of this answerform

This data is loaded in the following data structure:

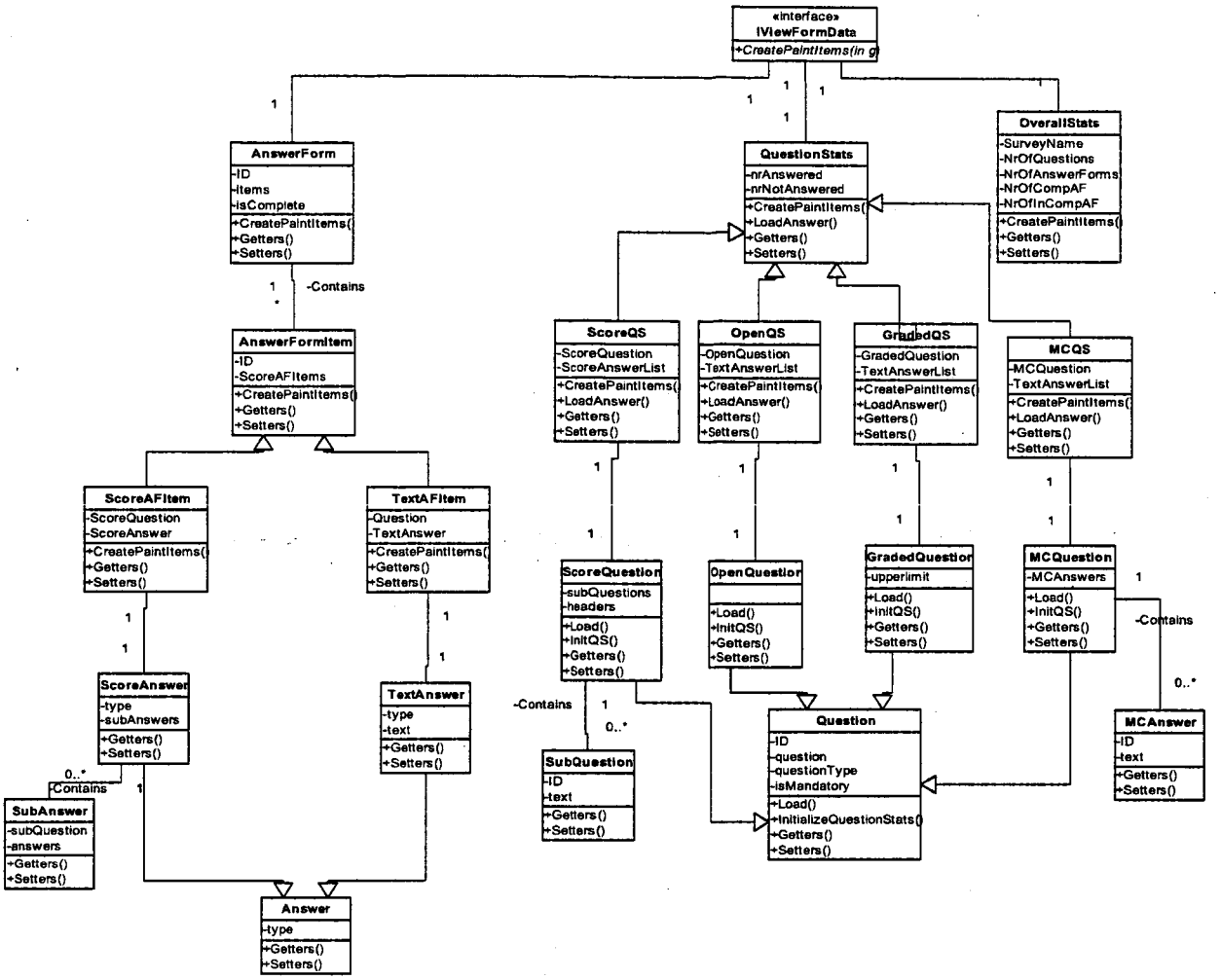


Figure A2 Data structure of survey data

The 3 categories of data, are each put in a separate data structure (AnswerForm, QuestionStats and OverallStats).

An ***AnswerForm*** object consists of:

- ID: unique identifier of the answerform
- A Boolean indicating whether the answerform is completely filled in (isComplete)
- AnswerForm Items. There are two kind of items: ScoreAFItem and a TextAFItem. The scoreAFItem represents the question and answer of a scorequestion. A TextAFItem represents a question and answer that are only text.

A ***QuestionStats*** object contains two integers indicating the number of nonempty and empty answers for this question. There are four types of questionstats:

ScoreQS: represents the statistics for a score question
OpenQS: represents the statistics for an open question
GradedQS: represents the statistics for a graded question
MCQS: represents the statistics for a multiple choice question

They all contain the proper type of question (e.g. a ScoreQS contains a ScoreQuestion) and a list of answers of the proper type. The data structure "Question" is discussed later.

The ***OverallStats*** object contains:

- Name of the survey
- Number of questions for this survey
- Number of answerforms filled in for this survey
- Number of complete answerforms
- Number of incomplete answerforms

A ***Question*** consists of an ID, the "real" question (in text), and a Boolean indicating whether the question is mandatory. There are four types of questions:

ScoreQuestion: a score question contains subquestions and headers. Answers to a scorequestion must be given in a 2-dimensional array. The subquestions are on the vertical ax and the headers on the horizontal ax. Per subquestion per header a number is filled in. The sum per header of all subquestions, must be a specific number.

OpenQuestion: An open question.
GradedQuestion: A question in which a grade must be assigned. This grade is limited to the "upperLimit".
MCQuestion: A question which has multiple predefined answers. The possible answers are stored in list MCAnswers.

An ***Answer*** is a pure abstract class. There are 2 types of answers:

ScoreAnswer: A score answer represents an answer to a score question. It contains the subanswers of this score answer. A subanswer contains the answer on a subQuestion, i.e. the results filled in under the headers of this subquestion.

TextAnswer: A textanswer only contains the answer text.

This data structure must be loaded from the Survey Entry Application database. To get the proper data, it is necessary to inspect each answer form, and walk through the questions in the same way as the participant. This is the only way to be certain that no questions are taken into account which where not in the "path" of a participant. A path of a participant cannot be predicted, since multiple choice questions can force a jump to different other questions, depending on the given answer. It is also possible that some answers on the database are not part of any path when somebody used the "back" button and changed his multiple choice answer. Since it is not efficient to walk through all answerform paths for each data item, all data items are loaded in one loop through all answerforms.

This part described purely the data to be stored. The following part describes the functions defined on these data objects.

The QuestionStats object has an abstract function "LoadAnswer". This function loads the answer, using parameter answerFormID to get the appropriate answers for the question of this object.

Many objects contain a function "CreatePaintItems". This function creates the paintitems for the object (a paintitem is defined in ...), i.e. it creates the graphical representation of the object.

The GUI

The GUI has one main MDI-form, "frmMain". The GUI has the Windows look-and-feel, providing a standard menu bar and toolbar. To enable/disable buttons/menus, the following robust architecture is used:

There is a set of buttons/menuitems which are disabled/enabled depending on the current selected form. frmMain contains the function updateMenuBar and updateToolBar. Each button/menuitem is associated with a specific interface. If the active form implements the interface, the button/menuitem is enabled, otherwise it is disabled. Therefore it is needed to call the update functions when a form gets the focus. There are also forms that do not activate any button/menuitem (such as the printpreviewdialog). Usually these are standard forms and it is not desired to create extra forms to only implement an extra "onFocus" eventhandler. Therefore the update functions on the non-standard forms are not only called on focus, but also when they lost focus.

Showing Data

The forms that must show data from a survey are ViewForms. Class ViewForm is an abstract class. There are three classes that inherit from ViewForm:

frmOverallStats: to show overall statistics
frmQuestion: to show statistics for a question
frmAnswerForm: to show an answerform

Base class ViewForm contains among others functions PrintPage and PrintScreen, which prints the data of the form on a page or on the screen. The data is obtained from the controller. frmQuestion and frmAnswerform contain a list of dataidentifiers (set in the constructor) and they hold the current position in this list for the current data. frmOverallStats does not need a list, since it only has one data object to show.

For printing, class "PaintGraphicsModule" is used. This class prints paintItems which are created and set by the ViewForm. These paintItems represent the data of such a ViewForm (e.g. in case of a frmQuestion, this data is QuestionStats data). PaintItem is an abstract class, from which several classes inherit. These subclasses are:

TextItem: An item containing text
TextValueItem: An item containing text, a specified with of the text and a value
ScoreItem: An item containing a question field, and header fields
ChartItem: An item containing a chart. Chart is an abstract class, derived classes construct the appropriate chart in the constructor, therefore not needing other functions than the base class.

Note: the width of an item is not set when it is created as paintitem by a ViewForm, since at that time it is not known what the width is. Therefore the width is set in "drawNextItem", just before it is printed.

Each subclass must implement the functions PrintOnScreen, PrintOnGraphics and GetHeight. The printfunctions has as parameter the position to print on and the target to print on. The PaintGraphicsModule is the global coordinator, giving "orders" to the item to print on a specific position and target. Function GetHeight is used to to update the paint position and to check in advance whether the item will fit on a page.

The PaintGraphicsModule contains the global settings, among which the papersize, margins etc. The meaning of the variables used is depicted in the next figure:

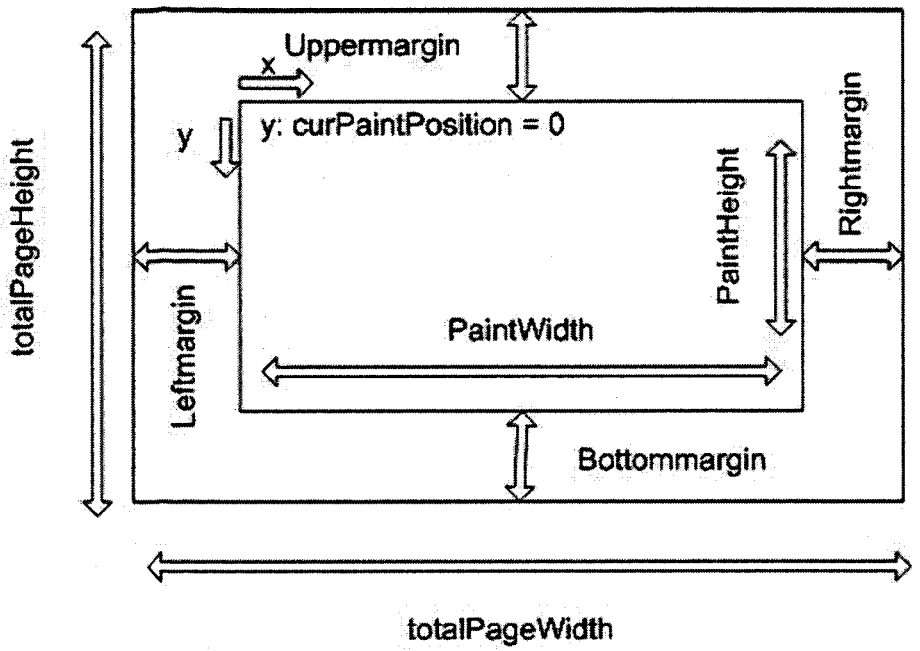


Figure A3 print overview

Important variable is the current paint position. This variable contains the y-position on the paintable area (i.e. the pagearea **without** the margins).

When printing on the screen, there is no need to check whether an item fits on the page (since there is only one page, and scrollbars are used when it does not fit on the visible part of the screen). But when printing on the printer, it is checked whether the item fits before it is printed.

Note: there was a problem with printing in the .NET Framework. The preview did not gave the same result as the printer output. Reason for this was that the preview did not take into account the hard printer margins. In version 1.1 of the .NET Framework an extra property is available to specify whether the origin must be on the hard printer margins or really on (0,0).

Storing SRA Database

The data that must be stored for the Survey Report Application is stored on the same database as the data for the Survey Entry Application. Main reasons for this where simplicity and referential integrity between the surveyID field of a report layout and the ID field of a survey.

The SRA data to be stored are Report Layouts and Preferences. These are stored in the following tables:

tblReportLayout

ID	Unique identifier for this table
surveyID	ID of the survey to which this report layout belongs
name	Name of this report layout
statsGS	Graphical style of statistics
MCGS	Graphical style of multiple choice questions
inclOS	Bit indicating whether overall statistics are included in this report layout
inclAF	Bit indicating whether answer forms are included in this report layout
itemOnNewPage	Bit indicating whether each item must be on a new page
headerID	ID of the header of this report layout
footerID	ID of the footer of this report layout

tblHeader

ID	Unique identifier for this table
inclSurveyName	Bit indicating whether the survey name must be included
customBool	Bit indicating whether custom text must be included
customText	The custom text

tblFooter

ID	Unique identifier for this table
inclDate	Bit indicating whether date must be included
inclPageNumber	Bit indicating whether page numbers must be included
customBool	Bit indicating whether custom text must be included
customText	The custom text

tblPreferences

username	Name of the user for these preferences
statsGS	Graph Style for depicting statistics
MCGS	Graph style fore depicting Multiple Choice Questions

Following depicts the tables and their relations:

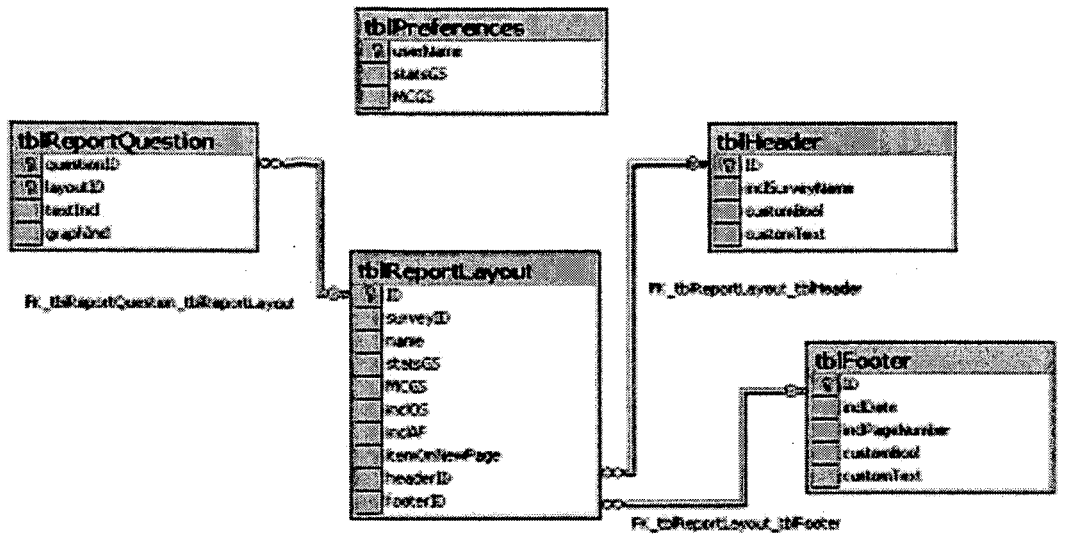


Figure A4 SRA Database structure

Appendix B: Configuration Examples

This appendix describes the configuration of the framework. The framework reads the configuration from a configuration file. Since the framework exists of a "server-side" and a "client-side" part, there are two kinds of configuration files. Both are xml-files, which structure is described by an XML-Schema.

An example of a server side configuration is:

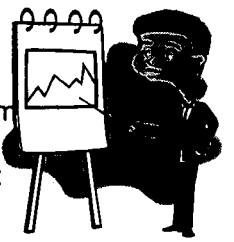
```
<?xml version="1.0" encoding="utf-8" ?>
<ConfigurationIn xmlns="ConfigurationIn.xsd">
     <RemoteObject>
          <Name>CompX</Name>
          <ActivationTypç>CAO</ActivationTypç>
          <Channel>TCP</Channel>
          <Port>1234</Port>
          <Endpoint>CompX</Endpoint>
          <Lifetime>
               <InitialLeaseTime>10</InitialLeaseTime>
               <RenewOnCallTime>3</RenewOnCallTime>
          </Lifetime>
          <RealObject>
               <AssemblyName>CompX</AssemblyName>
               <AssemblyType>NS__CompX.CompX</AssemblyType>
          </RealObject>
          <Mutex>
               <All>false</All>
               <Group>
                    <Function>test3</Function>
                    <Function>test4</Function>
               </Group>
               <Group>
                    <Function>test</Function>
                    <Function>test2</Function>
                    <Function>function2</Function>
               </Group>
               <SelfMutex>
                    <Func>test4</Func>
                    <Func>test3</Func>
               </SelfMutex>
          </Mutex>
     </RemoteObject>
</ConfigurationIn>
```

An example of a configuration is the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<ConfigurationOut xmlns="ConfigurationOut.xsd">
     <RemoteObject>
          <Name>CompX</Name>
          <Channels>
               <Channel>TCP</Channel>
               <Channel>HTTP</Channel>
          </Channels>
          <ActivationType>CAO</ActivationType>
          <SponsorRenewalTime>5</SponsorRenewalTime>
          <Locations>
               <Location>tcp://localhost:1234/CompX</Location>
          </Locations>
          <Call>
               <Name>test</Name>
               <LDAlgorithm>ROUNDROBIN</LDAlgorithm>
               <PollTime>6</PollTime>
               <State>STATEFUL</State>
               <ReplicaPolicy>ACTIVEREPLICA</ReplicaPolicy>
               <TimeOut>30000</TimeOut>
               <MaxNrOfActiveReplicas>4</MaxNrOfActiveReplicas>
               <MaxNrOfActiveReplicaRetrials>1</MaxNrOfActiveReplicaRetrials>
               <MaxNrOfPassiveReplicaRetri_ls>1</MaxNrOfPassiveReplicaRetri_ls>
               <NrOfRetrialsOnFailure>1</NrOfRetrialsOnFailure>
          </Call>
     </RemoteObject>
     <RemoteObject>
          <Name>CompY</Name>
          <Channels>
               <Channel>TCP</Channel>
               <Channel>HTTP</Channel>
          </Channels>
          <ActivationType>SINGLETON</ActivationType>
          <SponsorRenewalTime>5000</SponsorRenewalTime>
          <Locations>
               <Location>tcp://localhost:1234/CompZ</Location>
               <Location>tcp://localhost:1235/CompY</Location>
          </Locations>
          <Call>
               <Name>test</Name>
               <LDAlgorithm>ROUNDROBIN</LDAlgorithm>
               <PollTime>5</PollTime>
               <State>STATELESS</State>
               <ReplicaPolicy>ACTIVEREPLICA</ReplicaPolicy>
               <TimeOut>30000</TimeOut>
               <MaxNrOfActiveReplicas>2</MaxNrOfActiveReplicas>
               <MaxNrOfActiveReplicaRetrials>1</MaxNrOfActiveReplicaRetrials>
               <MaxNrOfPassiveReplicaRetri_ls>1</MaxNrOfPassiveReplicaRetri_ls>
               <NrOfRetrialsOnFailure>1</NrOfRetrialsOnFailure>
          </Call>
     </RemoteObject>
</ConfigurationOut>
```

Appendix C: SRA Requirements

This chapter lists the requirements for the Survey Report Application (SRA).



General Functionality

[CR-SRA-1] The SRA shall be able to show the results of one survey at a tim

[CR-SRA-2] The SRA shall be able to show the following results per survey:

- [2a] complete answer sheet
- [2b] data per question
- [2c] overall data

[CR-SRA-3] Depending on the type of results, the survey is presented in a textual and/or in a graphical way.

Detailed Functionality

[CR-SRA-4] Textual presentation of a survey must enable the user to view:

- [4a] overall statistics (detailed in [CR-SRA-6])
- [4b] statistics per question (detailed in [CR-SRA-7])
- [4c] answers (in case of open questions)
- [4d] complete answer sheet

[CR-SRA-5] Graphical presentation of a survey shall enable the user to view:

- [5a] overall statistics (detailed in [CR-SRA-6])
- [5b] statistics per question (detailed in [CR-SRA-7])

[CR-SRA-6] Overall statistics of a survey provide:

- [6a] number of questions
- [6b] number of participants
- [6e] number of completely filled in answer sheets
- [6f] number of partly filled in answer sheets
- [6g] useful balances between above numbers

[CR-SRA-7] Statistics per question provide:

- [7a] number of participants who answered the question
- [7b] number of participants who did not answer the question
- [7c] aggregated results of the question
- [7d] useful balances between above numbers

[CR-SRA-8] It shall be possible to store preferences for viewing the results. Preferences are:

- [8a] which graphical presentation is preferred for presenting the result of a specific question type
- [8b] which graphical presentation is preferred for presenting the overall statistics of the survey
- [8c] which graphical presentation is preferred for presenting the statistics of a question

[CR-SRA-10] It shall be possible to view and print surveys in a pre-defined way per survey. This pre-defined way is called a report layout. The user shall be able to:

- [10a] design a report layout
- [10b] store a report layout
- [10c] load a report layout

[CR-SRA-11] A report is a document as specified in the report layout (see [CR-SRA-10]), with as contents the actual data from the database. The user shall be able to:

- [11a] print a report.
- [11b] preview a report.

[CR-SRA-12] In a report layout the user shall be able to:

- [12a] define header and footer
- [12b] select question results/statistics to be included
- [12c] select whether or not answer sheets must be included
- [12d] select whether or not overall statistics must be included

[CR-SRA-13] It shall be possible to print the contents of the data presented on the screen.

[CR-SRA-14] At all times it shall be clear how many answers are taken into account when reporting overviews, percentages, etcetera.

[CR-SRA-15] It shall be possible to export the results to XML.

Software

[CR-SRA-16] The SRA shall be developed using Visual Studio .NET (C#).

[CR-SRA-17] Remote objects shall be used to implement the application.

[CR-SRA-18] There are no strict performance requirements. Application functionality and acquiring .NET knowledge are more important than performance.

[CR-SRA-19] If supported by .NET, an update of the application must be possible without intervention of the user.

[CR-SRA-20] For a programmer it must be possible to build in reports for new question types easily.

User Interface

[CR-SRA-21] The user interface (UI) shall have the windows look and feel.

[CR-SRA-22] The SRA will support a minimal screen resolution of 600 x 800. The SRA shall be optimized for a 1024 x 768 screen resolution.

[CR-SRA-23] It must be possible to view statistics of multiple questions at the same time.

[CR-SRA-24] It must be possible to view multiple answer sheets at the same time.