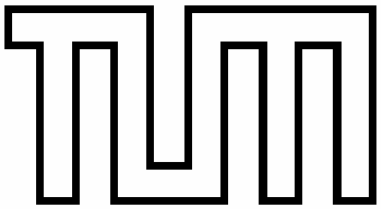
Distributed Object Computing
Caching and Prefetching

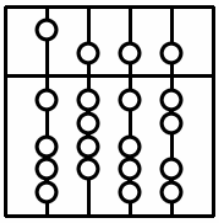
Christoph Vilsmeier

Institut für Informatik
Technische Universität München



Institut für Informatik

der Technischen
Universität München



Distributed Object Computing
Caching and Prefetching

Christoph Vilsmeier

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. H. M. Gerndt
Prüfer der Dissertation:	1.	Univ.-Prof. B. Brügge, Ph. D.
	2.	Univ.-Prof. A. Feldmann, Ph. D.

Die Dissertation wurde am 03.11.2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 07.07.2006 angenommen.

ABSTRACT

Distributed Object Computing eases the development of distributed applications by providing location transparency for remote method calls to distributed objects. Since remote method calls involve a network roundtrip, they are expensive compared to local method calls. Several existing approaches reduce the number of remote method calls but they require a high implementation effort.

The dissertation presents a novel approach that improves the performance of distributed applications with minimal implementation effort by caching and prefetching remote method result values. By caching of method result values, network roundtrips can be avoided for methods that are already in the cache. By prefetching method result values, the cache can be filled in advance, so that subsequent method calls do not need to be transferred over the network.

An abstract Distributed Object Computing Caching and Prefetching (DOC-CaP) framework was developed to provide caching and prefetching in the stub layer of distributed applications, thereby ensuring location transparency for client and server. An extension to the IDL language called XIDL was introduced to allow the specification of object interfaces and characteristics of the method implementations. The DOC-CaP IDL compiler generates the stub layer of any distributed application automatically from the XIDL definitions. Thus DOC-CaP allows developers to use the caching and prefetching approach without any manual additions to their code.

An instance of the DOC-CaP framework was implemented in CORBA and used to experimentally evaluate the caching and prefetching approach against a standard industry benchmark and a real-world industry application. Our evaluation results show that the DOC-CaP approach is applicable to a wide area of distributed applications and can improve the remote communication performance of distributed applications by factors of up to 60.

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor, Bernd Bruegge, for all the guidance and advice he has given me. He spent many long hours reading through the dissertation and discussing the work, even when he didn't have the time to do it. I also want to thank Anja Feldmann for reading preliminary versions of the dissertation and giving many fruitful hints.

I am deeply indebted to Peter Breitling for the long hours of discussion the potential benefits and pitfalls of caching methods. Guenther Walch gave me the opportunity to apply the theory of software development in real-world projects during my time at university. One of these projects will be presented in this dissertation.

I thank my friends for supporting me with a great private background and for their understanding when I had little time for them, especially in the last busy months of my work. Last not least, I would like to thank my parents for their support throughout the years.

OVERVIEW

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Contribution

Caching and prefetching have been used to increase the performance of computer memory systems, file systems, databases and the World Wide Web. This work extends well known caching and prefetching strategies to the field of Distributed Object Computing. The main contribution of this work is to show that caching and prefetching mechanisms can be used to improve the performance of distributed applications.

The promise of commercially available Distributed Object Computing products is to relieve programmers of distributed applications from network communication issues. Instead, the programmer can develop a given application like a local, in-process application. After design and implementation of the application, the subsystems of the application can be distributed across multiple computers and the network communication is handled by the Distributed Object Computing product. In this work we show that this promise is generally not met with today's Distributed Object Computing products. We demonstrate that serious performance problems arise and present current practice approaches that are used by application developers to speed up distributed application performance.

We then present a number of novel caching and prefetching mechanisms that increase the performance of distributed applications and show how it can be integrated with Distributed Object Computing products.

The mechanisms can be used to speed up not only newly developed applications, but also legacy systems. They do not require compiler knowledge or source code knowledge about the distributed application. In addition, an application programmer does not have to modify the source code: A single recompilation cycle of the network communication layer is sufficient for incorporating the caching and prefetching mechanisms into an existing distributed application. In contrast to prefetching approaches where the application itself has the responsibility to advise the prefetching subsystem about its future needs (also known as

'informed' prefetching in [01] and [02]), the required prefetching information is automatically gathered.

To validate the concept, we present three case studies along with performance measurements. First, we introduce a simple application ('Address Book'), which we use for illustration purposes throughout this work. Then we use a real-world application built according to the industry-standard object benchmark suite 'TPC-W benchmark'. Finally we present a distributed application used in an automobile company to show how our system can be used to speed up real-world applications.

1.2. Distributed Object Computing

Object-oriented programming is nowadays considered the method of choice for developing and maintaining complex software systems. With concepts like information hiding, interface inheritance and implementation inheritance, the object-oriented programming approach promises the development of more reusable and maintainable software systems.

With object-oriented programming, software systems are based on objects that communicate with each other via method calls. Figure 1 shows two objects, called 'Client' and 'Server', communicating with each other. Because the communicating objects are located in the same address space 'P1' in the same process, the method call is called 'local'.

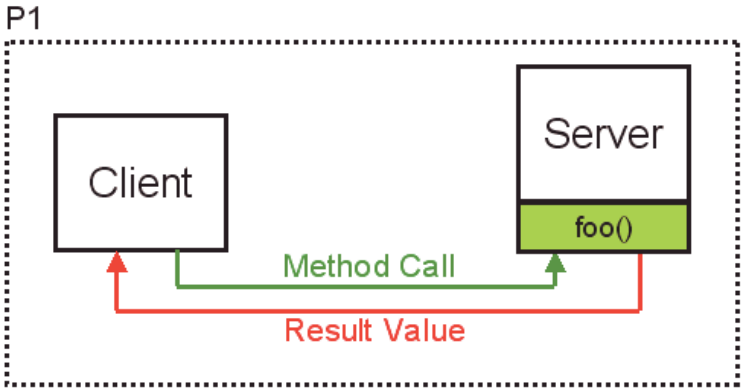


Figure 1: A Local Method Call

The `Client` object calls a method `foo()` that is provided by the `Server` object. The client has to know the signature of the called method, i.e. the method name, the list of parameter types and the type of the result value. Upon receipt of the method call, the server's method implementation is executed. After the method implementation has finished executing, a result value is returned to the client object. If the client is blocked while the method implementation

is executing, the method call is synchronous. If the method call returns immediately, and the client can go on while the method implementation in the server is executing, the method call is asynchronous. In both cases, the he internals of the server object's state and the details of its method implementations are unknown to the client. The collection of the method signatures exported by an object is called its 'interface'.

Figure 2 shows a sequence diagram of a synchronous local method call[1].

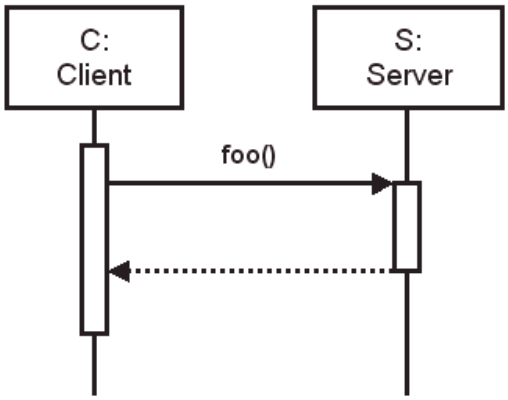


Figure 2: Sequence Diagram Of A Local Method Call

The `Client` object `C` calls a method `foo()` provided by the `Server` object `S`. If the method call is synchronous, the `Client C` has to wait while the method implementation in `S` is executing. After the method implementation is finished, the result value is returned and the thread of control is given back to `C`. If the method call is asynchronous, the method call returns immediately and `C` does not have to wait until the method implementation of `foo()` has finished. Instead, the thread of control is back at `C` immediately and the execution of `foo()` in `S` is executed in parallel to the execution of `C`. `C` can retrieve the result value later by polling the server or getting notified when `S` has finished executing the `foo()` method implementation.

It is important to note that high-level method calls cannot be executed by the computer hardware directly. Instead, a compiler translates the high-level source code into machine instructions that can be executed by the underlying hardware processor. The high-level programmer does not need to know the details of the compilation process or the machine instructions the compiler generates.

[1] UML Sequence Diagrams are used throughout this document to illustrate the communication of objects. For an introduction to UML, see [03].

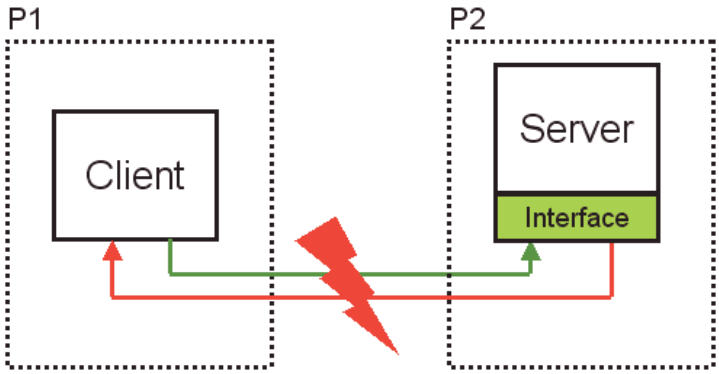


Figure 3: Remote Method Call

When objects are located in different address spaces 'P1' and 'P2', as shown in Figure 3, object communication via local method calls is not possible. Instead, the communication has to be done over a communication medium that connects `Client` and `Server`[2], for example a computer network or inter-process communication mechanisms like Named Pipes or Shared Memory (see [69] and [70]).

Modern computer networks are designed in a highly structured way. To reduce their design complexity, the networks are organized as a series of layers, each one built upon its predecessor. Decomposing a computer network system into layers has the advantage that each network subsystem can be replaced as long as it is ensured that the interface to its upper and lower layer stays the same. Two well-known layered network architectures are the seven-layer ISO-OSI reference architecture [05] and the four-layer TCP/IP reference architecture [04].

The OSI reference architecture model was developed by the International Organization for Standardization (ISO, see [06]). The model is called OSI (Open Systems Interconnection) Reference Model because it deals with connecting open systems - systems that allow for communication between heterogenous systems.

The OSI model has seven layers, based on the following design principles:

1. A layer should be created where a different level of abstraction is needed.
2. Each layer should perform a well-defined function.
3. The function of each layer should be chosen with respect to internationally standardized protocols.
4. The layer boundaries should be chosen to minimize the information flow across the interfaces.

[2] Here, we use the term 'Client' in the sense of caller and 'Server' in the sense of callee. It should not be confused with the term client/server architecture.

5. The number of layers should be large enough that distinct functions need not be realized together in the same layer, and small enough that the architecture does not become unwieldy.

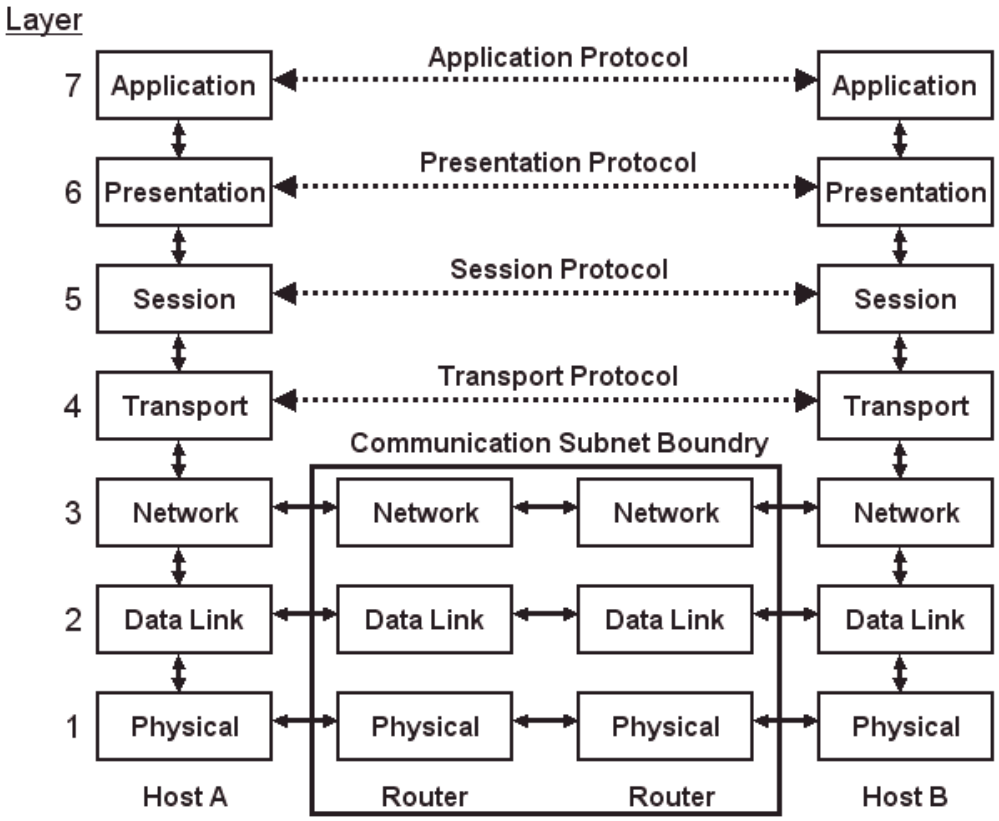


Figure 4: The ISO-OSI Reference Architecture

Figure 4 presents the OSI Reference Model. Seven layers are defined:

1. *Physical Layer*. The physical later is concerned with transmitting raw bits over a communication channel, e.g. a copper wire or a radio frequency connection.
2. *Data Link Layer*. The main task of the data link layer is to take an unreliable transmission facility and transform it into a line that hides transmission errors from the network layer. Commonly used techniques include framing, using checksums and acknowledge frames.
3. *Network Layer*. The main task of the network layer is routing data packets and making sure that they are transferred from the source to the destination.
4. *Transport Layer*. The basic function of the transport layer is to accept data from the session layer, split it up into smaller units if necessary, pass these to the network layer, and ensure that they arrive correctly at the destination.

5. *Session Layer*. A session might be opened and closed and might be used, e.g., to allow a user to log into a remote time-sharing system or to transfer a file between two machines.
6. *Presentation Layer*. Unlike all the lower layers, which are just moving bits reliably from one place to another, the presentation layer is concerned with the syntax and semantics of the information transmitted. It translates user-defined data into sequences of bytes and passes these bytes to the session layer.
7. *Application Layer*. The application layer defines the application protocol that is used by user applications to communicate with each other. Examples are file transfer protocols, electronic mail protocols and protocols for sending remote method calls.

Compared with the OSI architecture, the TCP/IP reference architecture leaves out OSI layers 5 and 6 (Session Layer and Presentation Layer) and combines OSI layers 1 and 2 (Physical Layer and Data Link Layer) into a single layer (Host-To-Network Layer). The TCP/IP model is shown in Figure 5.

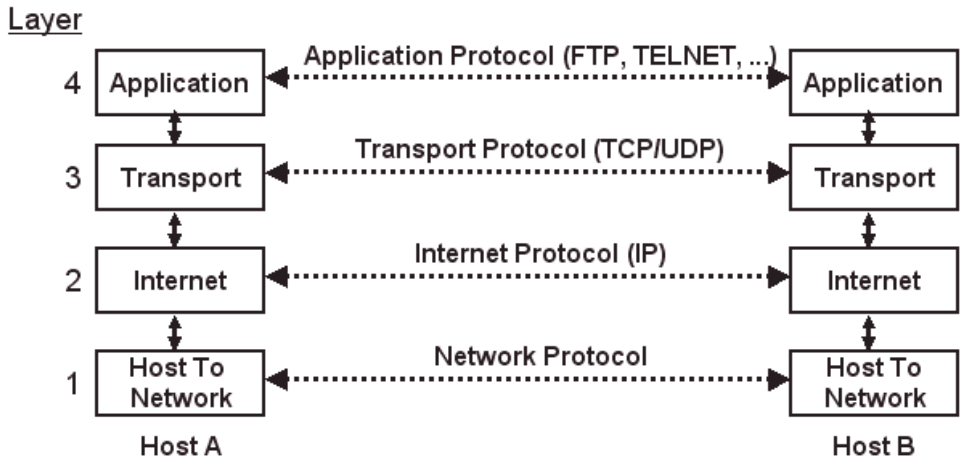


Figure 5: The TCP/IP Reference Architecture

The TCP/IP Reference Model defines four layers:

1. *Host-To-Network Layer*. The Host-to-Network layer provides the interfaces to the physical network. The layer only specifies that the host has to connect to the network using some protocol so it can send IP packets over it.
2. *Internet Layer*. The task of the internet layer is to inject packets into any network and have them travel independently to the destination. The layer defines IP (Internet Protocol) for its official packet format and protocol. Packet routing is one of the main tasks of this layer.
3. *Transport Layer*. The transport layer adds services to the internet layer that provide end-to-end communication to the application layer. The transport layer defines TCP (Transport Control Protocol) for reliable connection-based communication and UDP

(User Datagram Protocol) for unreliable connection-less communication as its protocols.

4. *Application Layer*. The original TCP/IP specification described a number of different applications that fit into the top layer of the protocol stack. These applications include Telnet, FTP, SMTP and DNS. Moreover, any network application that uses an implementation of a TCP/IP model for network communication, is located at the application layer.

The main difference between the OSI and the TCP/IP architectures is that TCP/IP is widely used and OSI is not. Implementations of the TCP/IP architecture are available for almost every operating system and computer platform.

The Berkeley Socket Interface, the de-facto standard for network programming provides an application-programming interface (API) that a developer can use to read and write data to and from TCP/IP communication networks. A socket is an abstraction of an application-layer communication endpoint that can be connected to another socket and used for reading and writing data. Using sockets, the process of building data packets, wrapping them in network frames, enforcing communication reliability, routing packets from source to destination, detecting data collisions and packet losses and so on, is completely hidden from the application. Figure 6 shows where client and server applications are located in the TCP/IP reference architecture.

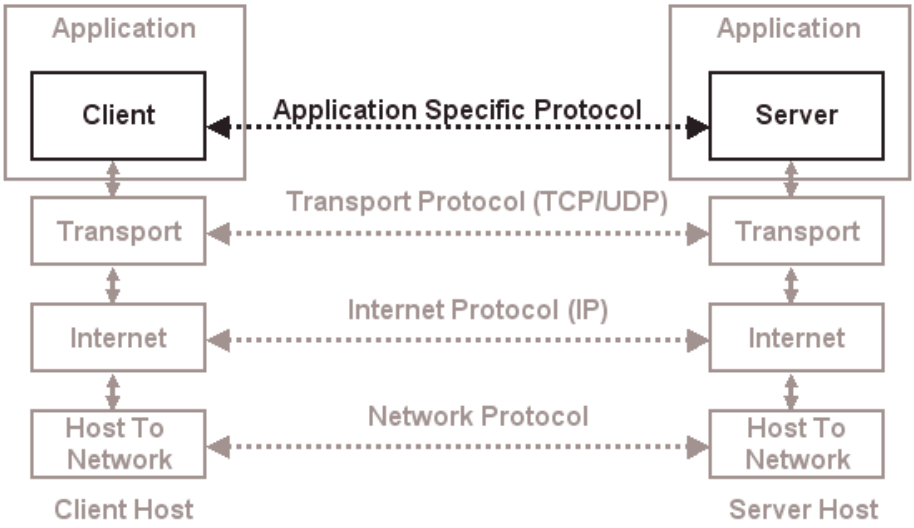


Figure 6: TCP/IP Client/Server Communication

But even with the availability of TCP/IP implementations and the Berkeley Socket Interface, realizing distributed data communication over a computer network is still difficult and error-prone. For example, the programmer of an application faces the following tasks:

1. Define a message-based application protocol for client/server communication. The client sends a service request message to the server, which executes the service and sends back a response message to the client.
2. Establish a socket connection between client and server object. If a client wishes to invoke a service implemented by a server, it has to know the network address of the server.
3. Generate a service request description of the service to be executed by the server, i.e. the service's name and parameter values.
4. Transform the service request description into an array of bytes and send it across the network, using a socket connection.
5. Wait for the response message and read it from the network. The response message arrives as a byte array and the result value has to be extracted.
6. Close the client-server connection if it is no longer needed.
7. Deal with error conditions like communication failures, network errors and unwanted server shutdowns.

Implementing client/server request/response communication by hand has several problems. The application programmer has to make sure that both the client and the server understand the same application protocol. The implementation of the network communication protocol in the client as well as server requires considerable development effort. When software requirements change, for example when a new service is added to an existing server, the application protocol has to be re-defined and the client and server software have to be changed accordingly, increasing software maintenance effort. If the communication medium changes, for example from socket-based communication to inter-process communication using shared memory, the communication protocol has to be re-implemented. The client/server application protocol must specify the details of the client/server network communication, like connection establishment, connection shutdown and data exchange formats. Application specific communication protocols may also lead to systems that are not interoperable, in the sense that a client program from vendor A cannot communicate with a server program from vendor B.

With the advent of the Remote Procedure Call (RPC) mechanism [07], a higher level of abstraction was introduced in client/server communication. The basic idea of the RPC mechanism is to replace message-based network programming with method call semantics.

Figure 7 illustrates how these additional abstractions are incorporated in the TCP/IP architecture. A *`Client Stub`* is located at the client side of a distributed application and acts as a proxy for the *`Server`* object. Analogous, a *`Server Stub`* is located at the server side and acts as a proxy for the *`Client`* object.

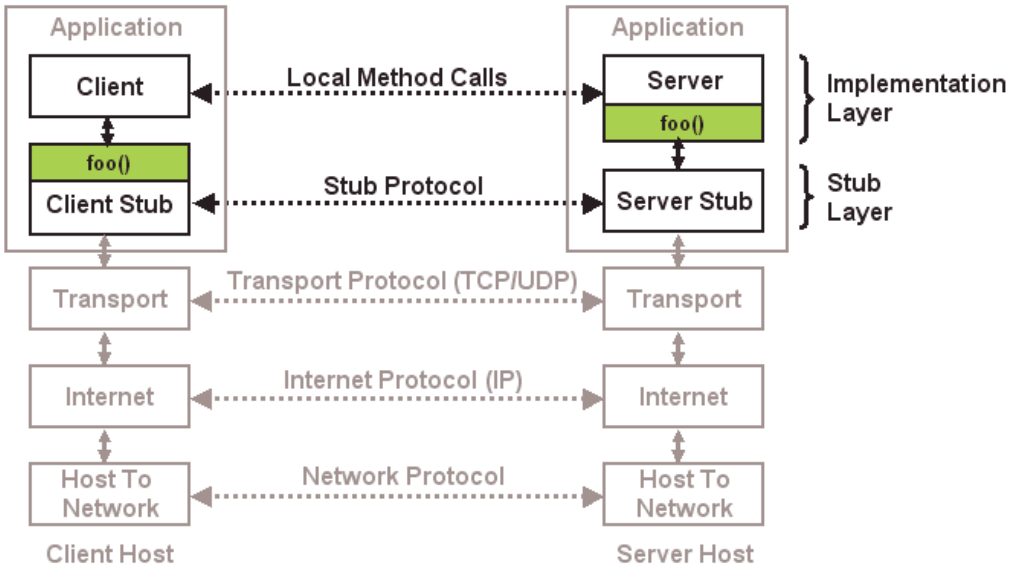


Figure 7: RPC Client/Server Communication

The TCP/IP Application Layer is replaced by an Implementation Layer and a Stub Layer. The *`Client Stub`* and the *`Server Stub`* are located in the Stub Layer and communicate with each other via the RPC Stub Protocol. *`Client`* and *`Server`* implementations are located in the Implementation Layer and communicate with each other via local method calls.

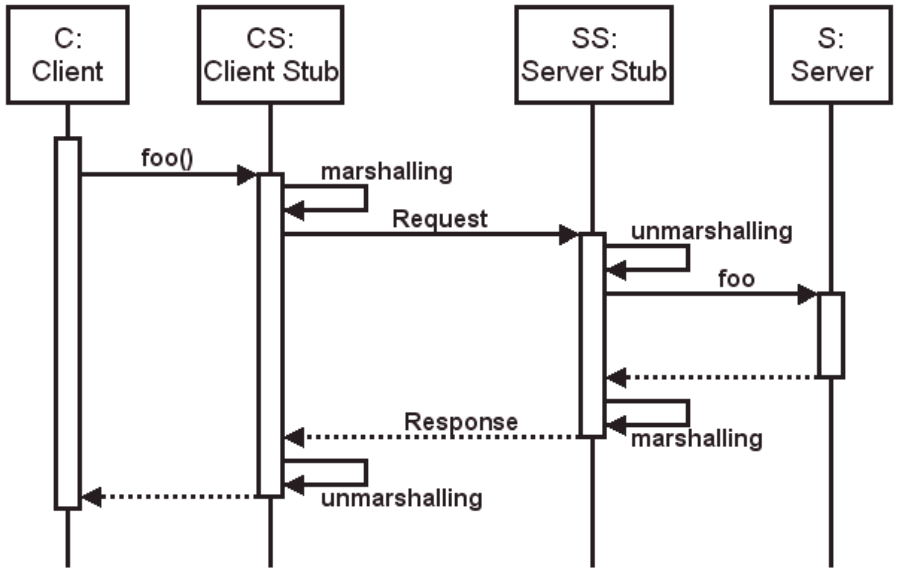


Figure 8: Sequence Diagram Of A Remote Method Call

Figure 8 presents a simplified sequence diagram of a remote method call and shows how the abstraction of local method calls is realized.

If a *Client* wants to invoke a method *foo()* provided by a *Server*, the *Client* calls the appropriate method (*foo()* in Figure 8) in the *Client Stub*. The *Client Stub* generates a request message that describes the remote method call (including the network address of the server, method name and parameter values). This request message is then converted to a data format that can be passed to the network transport layer. This conversion process is called 'marshalling'. The request then travels down the TCP/IP layers, which involves a number of calls to the TCP/IP library, operating system functions, network adapter device driver calls, etc. Upon receipt of the request packet, the *Server Stub* extracts the method call request information from the incoming byte stream and converts it back into a request message, the conversion process called 'unmarshalling'. Then the *Server Stub* calls the appropriate method implementation of the *Server*. After the method implementation has finished executing, the method result value is marshalled into a response message and sent back to the *Client Stub*. The *Client Stub* unmarshalls the response and returns the method result value to the *Client* object.

The *Client Stub* provides exactly the same methods as the *Server*. This way, the *Client* can issue method calls as if it is calling the *Server* directly via a local method call. The *Client Stub* handles the network communication transparently to the *Client*. At the server side, the *Server* is called through its interface, as if the method calls would originate directly from the *Client*. The *Server* cannot tell whether methods are invoked from the *Client* or a *Server Stub*. The fact that method calls and method result values are transferred over a network is transparent to both the *Client* and the *Server*.

The source code of the Stub Layer does not need to be provided by the application programmer. Instead, it can be generated automatically from an abstract description of the *Server* interface. We leave out an explanation of the stub generation for now, as we will explain it later in great detail.

The RPC mechanism supports remote application communication via method call semantics, but RPC does not provide object-oriented semantics like object interfaces, inheritance and object-oriented method calls. Thus, RPC is not directly suited for object-oriented systems, where remote object communication is desired[3]. In the next section we introduce the 'Distributed Object Computing' mechanism, which extends the functionality of RPC to support remote object communication.

[3] RPC can be seen as the C programming language [71], where an application provides only global functions, as opposed to the C++ programming language, which extends C by introducing object-oriented paradigms like object interfaces, inheritance and polymorphic methods.

Figure 9 presents a client/server system where the server application – in contrast to Figure 7 – hosts a number of *Server* objects that provide services to remote *Client* objects. For each *Server* object, there is a *Client Stub* object at the client side and a *Server Stub* object at the server side. If a *Client* wishes to invoke a method in a certain *Server* object, it has to call a method on the *Client Stub* that represents the method in the *Server* object. The *Client Stubs* communicate with the *Server Stubs* using a stub communication protocol. (In fact, many Distributed Object Computing implementations use a modified RPC-like protocol for data communication, for example Microsoft DCOM [08] and the very first implementations of CORBA [10])

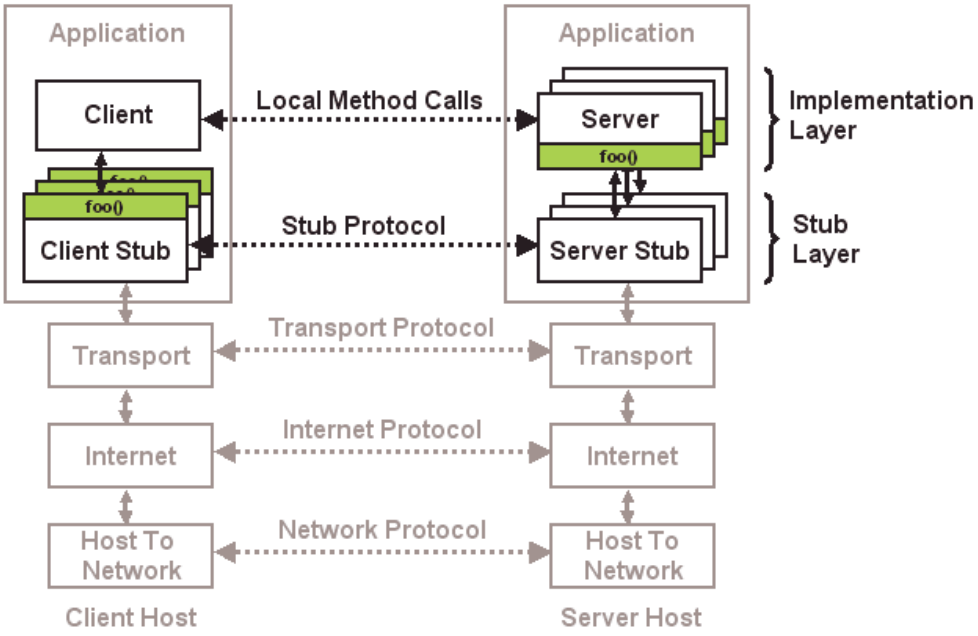


Figure 9: Distributed Object Computing

In the following section we develop an abstract framework for Distributed Object Computing systems. This framework provides the functionality for remote object communication, as shown in Figure 9. We will use the framework for illustrating purposes throughout the dissertation. Figure 10 presents a simplified class diagram for the Distributed Object Computing Framework. Note that the class diagram does not show *Client* and *Server* implementations that implement the functionality of a specific application domain. *Client* and *Server* implementations will be added and discussed later in this section, see Figure 11.

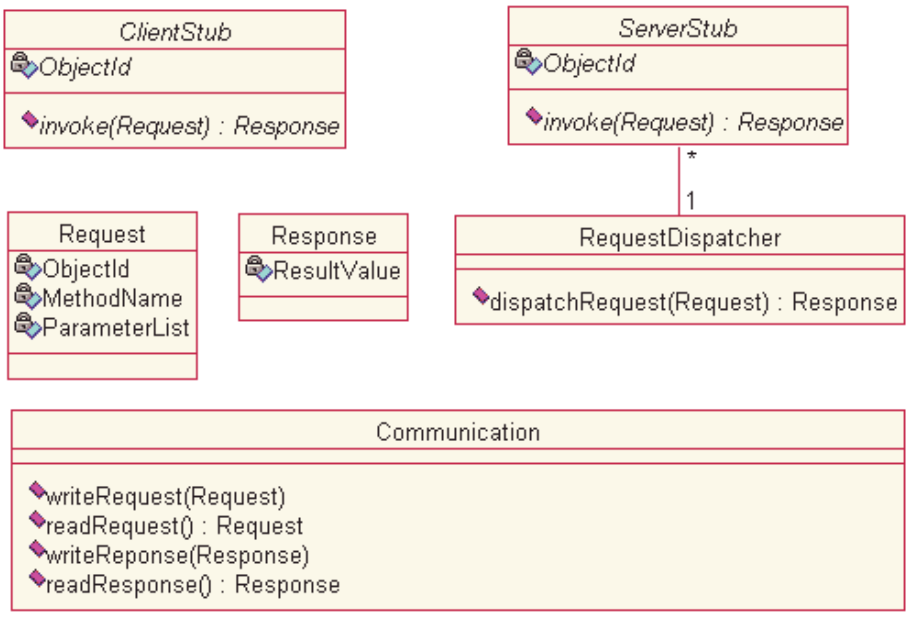


Figure 10: Distributed Object Computing Framework

The classes introduced in Figure 10 implement the functionality of sending and receiving *Request* and *Response* objects over a network.

Request is a container class that holds the information about a single remote method call: The attribute *ObjectId* is used to identify the *ServerStub* that should receive this *Request*. The attributes *MethodName* and *ParameterList* contain the information about the method to be invoked and its parameter values. For simplicity reasons the data types for the attributes *ObjectId*, *MethodName* and *ParameterList* are not specified. For now it is sufficient to assume that these attributes can be of any data type.

The *Response* class holds the information about the method result value of the method in its attribute *ResultValue*. Again, the data type is not specified.

ClientStub is the abstract base class for all client stub objects. It has an `ObjectId` that is used to uniquely identify the server side *ServerStub* that this *ClientStub* represents. The *invoke()* method is used to send a *Request* over the network and wait for the *Response*.

Communication provides the connection to the network layer, which can be a TCP/IP network for example. The method *writeRequest()* marshals a *Request* object into a data format that can be transferred over the network and writes it to the network transport layer. The method *readRequest()* reads data from the network and unmarshalls it into a

Request object. The methods *writeResponse()* and *readResponse()* are used for marshalling and unmarshalling *Response* objects.

RequestDispatcher is a class that every *ServerStub* must register with. The request dispatcher has a main loop that looks for *Request* objects from the network, via *Communication.readRequest()*. Upon receipt of a *Request*, the *Request* is dispatched to the *ServerStub* with the matching *ObjectId*.

ServerStub is a base class for all server side stubs. The field *ObjectId* is used for the identification of a *ServerStub*. The invoke() method is called by the *RequestDispatcher* to forward a Request to the *ServerStub*, which in turn calls the method implementation of the *Server* object (not shown in Figure 10). After executing the method implementation, the *ServerStub* creates a Response object, sets its *ResultValue* attribute and returns it to the *RequestDispatcher*.

After describing how method calls are transferred over a network in Distributed Object Computing, we introduce now an application sample that makes use of this functionality: Figure 11 presents a class diagram that contains a sample interface *Person* with a sample method *getName()*. The abstract base classes *ClientStub* and *ServerStub* are provided by the Distributed Object Computing Framework and are shown in Figure 10.

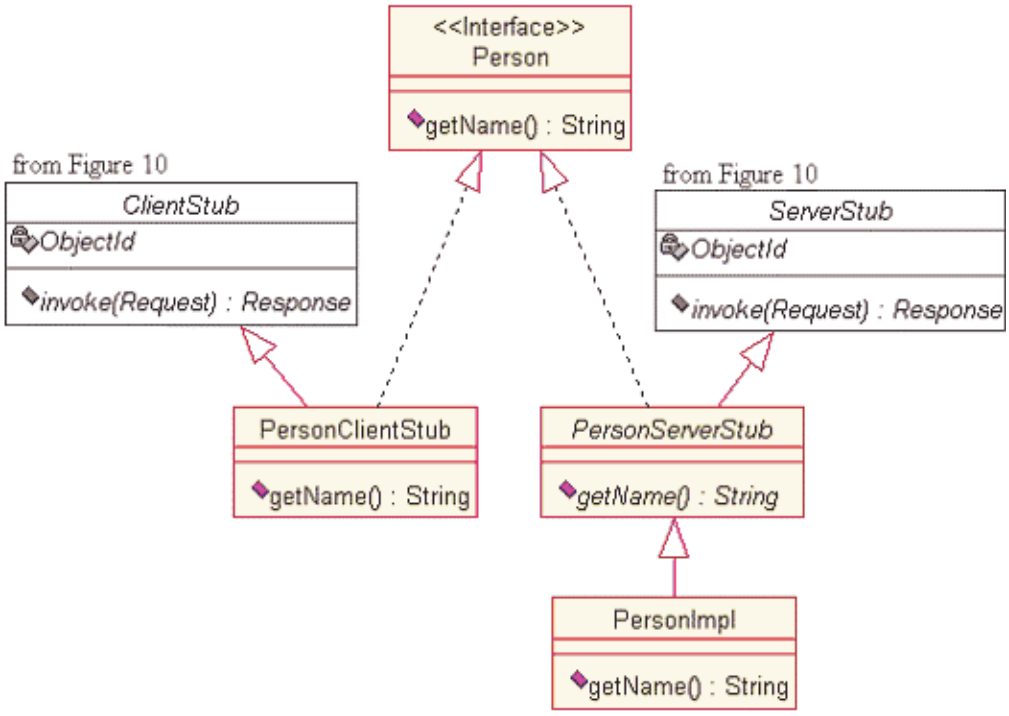


Figure 11: Sample Application Domain Classes

`Person` specifies the interface of all the methods provided by the `Server` object implementation. It was a single method `getName()` that returns a character string value. The interface `Person` is the specification between the `Server` object implementation and a `Client`. The `Server` object implementation must implement all methods that are specified by the interface. The client may call all methods specified by the interface.

`PersonServerStub` is the stub object that receives incoming requests (by extending `ServerStub` from Figure 10) and forwards them to the `Server` object implementation by calling the appropriate method, `getName()` in our example.

`PersonClientStub` is the client side counterpart of a `PersonServerStub` object. In our example, it implements the interface `Person`; thereby making sure that it provides the method `getName()`. It extends the class `ClientStub` in Figure 10, therefore inheriting the `invoke()` functionality.

`PersonImpl` is the server side `Server` object implementation. By convention, we append a trailing '`Impl`' to `Server` implementation classes. The programmer has to make sure that this class implements the interface `Person` and can be invoked via the `invoke()` method. This is done by making `PersonImpl` a subclass of `PersonServerStub`.

The dynamic model of the Distributed Object Computing Framework illustrates the interaction between the Distributed Object Computing framework objects and the sample application domain objects. Since *Client* and *Server* do not communicate via local method calls we present two sequence diagrams, one for the client side and one for the server side, shown in Figure 12 and Figure 13, respectively.

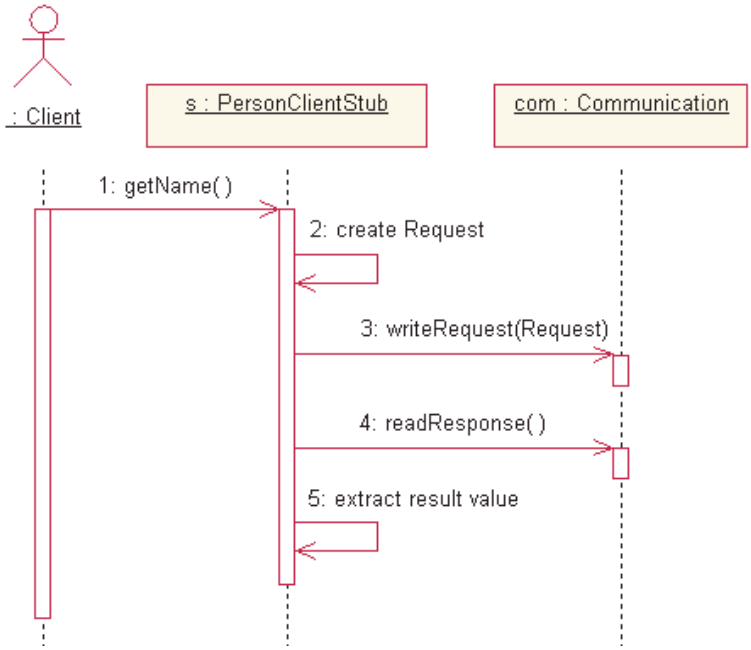


Figure 12: Sample Dynamic Model Of A Remote Method Call (Client Side)

Here, the *Client* calls *getName()* in the *Person* interface, which is realized by calling *getName()* in the *PersonClientStub* object that implements the *Person* interface and represents the server side *PersonServerStub*. After creating a *Request* object, the stub calls the *writeRequest()* method of a *Communication* object to send the *Request* over the network. Then it waits for the *Response* and reads it from the network, using the *readResponse()* method of the *Communication* object. After reading the *Response*, the method result value is extracted from the *Response* and returned to the *Client*. Note that waiting for the *Response* and reading it from the network is a synchronous operation, which means that the *PersonClientStub* is blocked while waiting for the *Response*[4].

[4] Note that, unless indicatd otherwise, remote method calls are synchronous, which means that the client is blocked until the method call has finished executing and the result value has been returned to the client.

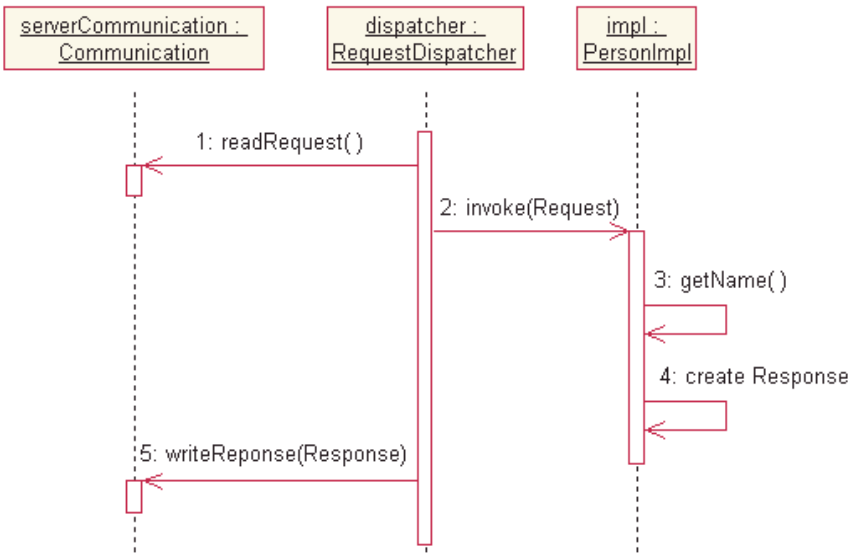


Figure 13: Sample Dynamic Model Of A Remote Method Call (Server Side)

Figure 13 presents the server side of the remote method call. The *RequestDispatcher* continuously waits for incoming *Requests* from the network through a *Communication* object. When a *Request* comes in, the *RequestDispatcher* reads it from the network using the *readRequest()* method, chooses an object implementation based on the *ObjectId* of the incoming *Request*, and forwards the *Request* to that object. In the case shown in Figure 13, it is a *PersonImpl* that the *Request* is forwarded to.

Note that *PersonImpl*, which is a subclass of *ServerStub*, is the *Server* implementation and the *server stub* at the same time. The *PersonImpl* reads the *MethodName* and *ParameterList* attributes of the incoming *Request* and invokes the method implementation of *getName()*. When the implementation of *getName()* has finished, the *PersonImpl* creates a *Response* object, stores the method result value (the name of the *Person*) in its *ResultValue* attribute and returns the *Response* object it to the *RequestDispatcher*, which in turn sends it back to the network.

Figure 14 combines the classes of the Distributed Object Computing framework (see Figure 10) and the classes of the sample application domain (see Figure 11) in a single class diagram.

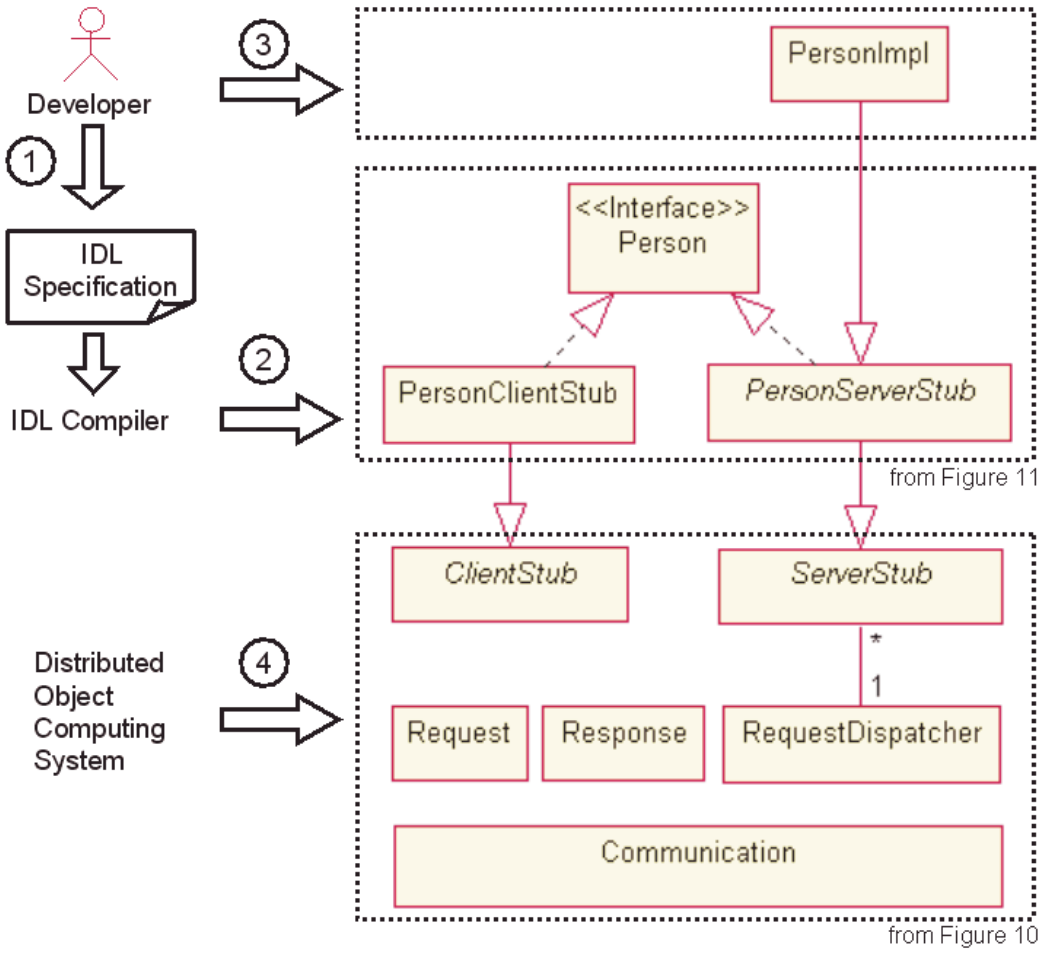


Figure 14: Distributed Object Computing Application Development

A developer creates and uses these classes as follows:

1. First, the developer specifies the interface of the *`Server`*, using an abstract interface definition language (IDL), resulting in an abstract specification of the server interface.
2. After the *`Server`* interface is specified, the source code of *`Person`*, *`ClientStub`* and *`ObjectAdapter`* are generated. A compiler that takes the interface definition and generates the source code of the ClientStub and ObjectAdapter classes as well as the Server interface can execute this step automatically. In this case, the interface *`Person`* and the classes *`PersonClientStub`* and *`PersonServerStub`* are generated.
3. The server objects (here *`PersonImpl`*) can now be implemented. Each class and each method that was specified by the abstract interface definition has to be implemented by the programmer.
4. The Distributed Object Computing system provides the classes *`ClientStub`*, *`ServerStub`*, *`Request`*, *`Response`*, *`RequestDispatcher`* and *`Communication`*.

The Distributed Object Computing Framework eases the development of distributed applications by hiding network transport issues from the programmer. Distribution of objects and network communication are transparent to the programmer who implements the `Client` and `Server` objects as if they communicate with each other through local method calls.

In practice however, developers of distributed applications have to be aware that remote method calls are slower than local method calls, since remote method calls involve parameter marshalling, network communication and request dispatching.

In the following we illustrate why remote method calls can easily lead to performance problems in distributed applications.

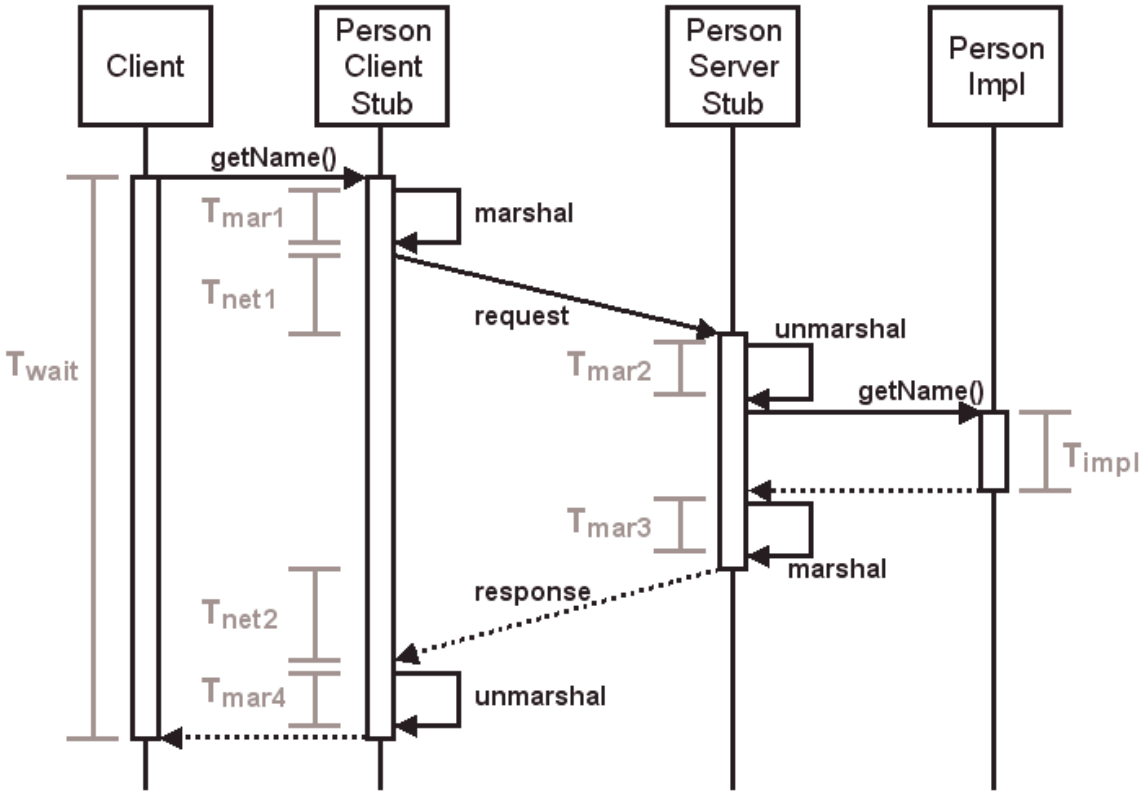


Figure 15: Remote Method Call: Client Waiting Time

Figure 15 shows the sequence diagram of a remote method call, annotated with time labels. If the programming model is synchronous, a client initiating a method call has to wait until the method call returns. This waiting time is called *Twait* and is the sum of three components: The marshalling time *Tmar*, which is the sum of *Tmar1*, *Tmar2*, *Tmar3* and *Tmar4* in Figure

15. *Tmar* is the time it takes to marshal and unmarshal *Request* and *Response* objects and dispatching them to the appropriate *ServerStub* and *ClientStub*. The second component is the network time *Tnet*, which is the sum of *Tnet1* and *Tnet2*. This is the time it takes for the *Request* and *Response* objects to be transferred from the client to the server. The third component is *Timpl*, the time it takes for the server to execute the method implementation.

The client waiting time Twait depends on a number of factors. In the context of our work, we regard the following factors:

- *Marshalling rate*. The marshalling rate defines how fast a Distributed Object Computing system can marshal method parameters and method result values into data packets that can be sent over a computer network. The marshalling rate depends, among other factors, on the type of a given method parameter. Marshalling user-defined structure types typically takes long than marshalling native types like integer values or character strings.
- *Number and size of method parameters and result values*. The network time – as well as the marshalling time – depends on the number and the size of method parameters and result values. Bigger request and response messages will take longer to be marshalled and transferred over the network than short data packets.
- *Network Bandwidth*. The available network bandwidth constrains how fast a data packet of a given size can be transferred over the computer network that connects client and server.
- *Network Latency*. The time it takes for a single byte to travel from client to server is determined by the ‘distance’ between client and server, the network hardware and topology, the network protocol and the number of network devices (routers, switches, relays, modems, satellite connections, etc) between client and server.
- *Method implementation*. The execution time of a method implementation is highly application dependent. For example, a method that triggers a complex database query will take significantly longer than a method that returns pre-computed data.

In the next section we show how these factors affect the client waiting time *Twait*, and therefore the performance of a distributed application built with a Distributed Object Computing system.

1.3. Distributed Object Computing Performance

Distributed Object Computing systems help to ease the development of distributed applications by providing location transparency and handling network communication issues. Location transparency means the programmer of a distributed application does not need to know whether client and server objects are located in the same address space in the same process or located across process boundaries, communicating with each other using remote instead of local method calls. However, although remote method calls may seem to behave like local method calls from the client's as well as the server's point of view, there is a difference when comparing local with remote method calls. The difference is the client waiting time *Twait*. If the client calls the server using a remote method call, *Twait* is

Twait = Tmar + Tnet + Timpl

whereas in the case of a local method call, *Twait* is simply *Timpl* if we leave out the time it takes to execute a local method call.

In the following we discuss how *Tmar* and *Tnet* affect the client waiting time *Twait*. In order to do this, we present a test application that executes remote method calls using a Distributed Object Computing system. The test application implements a very simple interface, which was introduced in Figure 11 in section 1.2. Figure 16 presents an IDL definition of this sample interface.

```
interface Person
{
        string getName();
};
```

Figure 16: Test Application Interface

The interface *Person* provides a single method *getName()*. The server side method implementation of *getName()* returns a pre-computed string value of a fixed length. The hardware and software configuration of the test bed is described in detail in Appendix 9.1.

The goal of the test run is to find out how the method result length, the network delay and the network bandwidth affect the marshalling time *Tmar*, the network time *Tnet* and thus the client waiting time *Twait* for the sample *Person* interface.

1. *Method Result Length*. The range of the method result length (the length of the character string that the *getName()* method implementation returns) goes from 0 to 32000 characters.
2. *Network Delay*. The range of the network delay goes from 0 to 160 milliseconds. A delay value of 0 milliseconds is the theoretical lower limit for all computer networks.

3. *Network Bandwidth*. The range of the network bandwidth goes from 1600 Kilobytes per second down to 50 Kilobytes per second.

The results of the test runs[5] show that the method result length, the network delay value and the network bandwidth affect the client waiting time `Twait`:

1. *Method Result Length*. If the method result value increases, the marshalling time `Tmar` and the network time `Tnet` of the remote method calls increase as well.
2. *Network Delay*. The greater the network delay, the longer it takes for both `Request` and `Response` objects to be transferred over the network. Thus, the network time `Tnet` depends on the network delay.
3. *Network Bandwidth*. Higher bandwidth means faster data transfer. Therefore, if the bandwidth grows, the network time `Tnet` decreases.

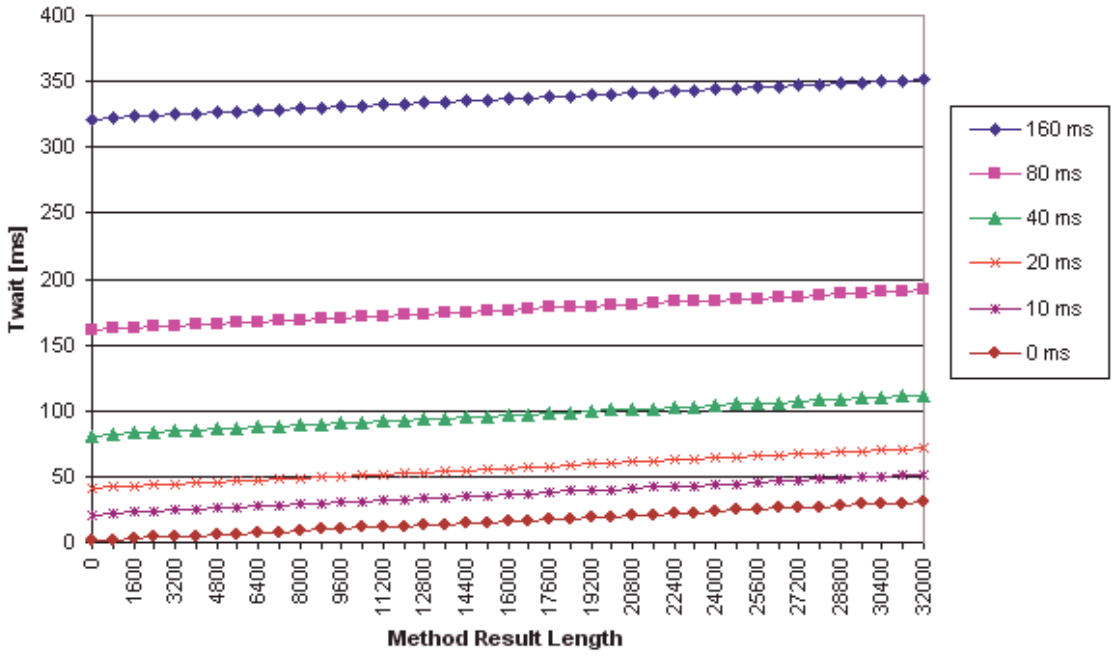


Figure 17: Twait For Different Network Delays

[5] The complete test results are shown in the Appendix, section 9.

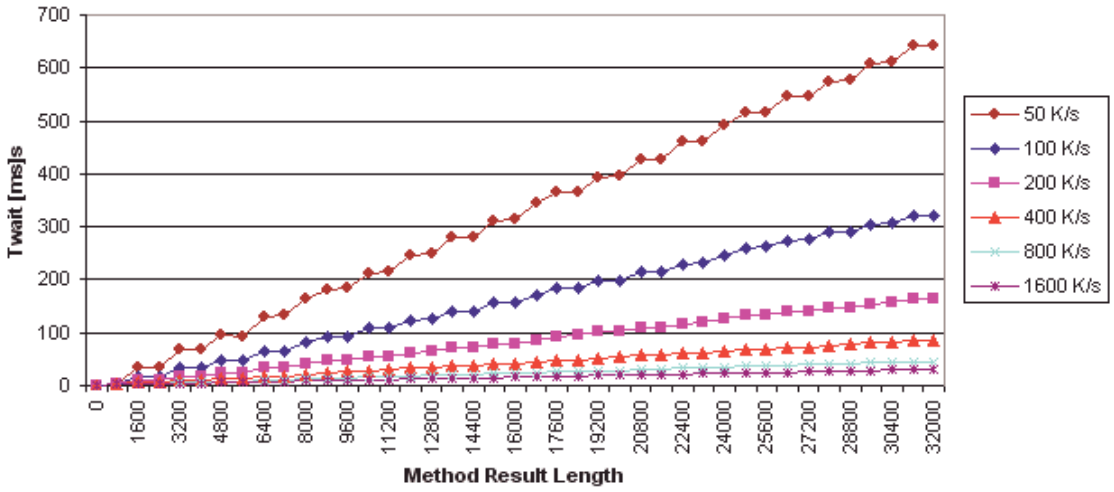


Figure 18: Twait For Different Bandwidths

The test run results shown in Figure 17 and Figure 18 indicate that, for the testbed environment described in Appendix 9.1 and the sample application, the client waiting time `Twait` is a linear function of the network delay `Tdelay`, the network bandwidth `bw` and the method result length `l` and can be written as:

```
Twait(Tdelay, bw, l) = T0(Tdelay) + c(bw)*l
```

`T0` is the offset of the `Twait` function and is determined by the network delay `Tdelay`. The growth factor `c` of the `Twait` function and is determined by the network bandwidth `bw`.

Based on the measurement results, `Twait` can be written as:

```
Twait(Tdelay, bw, l) = 1  +  2 * Tdelay  +  1.00156 * l/bw
```

As one can see from the test runs, the overhead of calling a remote method is significantly higher than the overhead of a local method call: Calling a remote method over a network with a near-zero network delay takes from 1 millisecond (method result length is 0) to 30 milliseconds (method result length is 32000 characters).

Even if we let the marshalling time `Tmar` approach zero for all remote method calls, this would lead to a considerable performance speedup only for near-zero delay networks, because `Tmar` is a relatively small component that adds to the client waiting time `Twait`.

Increasing the network bandwidth usually yields considerable performance speedups. However, while it might be possible to switch to high-speed networks (High-Speed ATM or Gigabit Ethernet, for example) in local area networks (LANs), the real world provides

networks that do not provide unlimited network bandwidth and – even more severe – are too expensive to upgrade. Decreasing the amount of data that has to be transferred over the network is a technique to improve the bandwidth usage of a given network. The programmer of a distributed application has to make sure that no more data is transferred over the network than is needed.

The network delay affects the performance of remote method calls considerably. Unfortunately, decreasing the network delay is not always possible for real-world networks, since it is determined by the network technology, topology and hardware. The only thing a programmer can do is to make sure that as few remote method calls as possible are sent over the network. As indicated by our test runs, it is better to call a few remote methods with big result values than calling many methods with small result values. The benefit B of combining n method calls into one single remote call can be written as:

```
B = n*Twait(Tdelay,bw,l)-Twait(Tdelay,bw,n*l) =

= n*(1+2*Tdelay+1.00156*l/bw) – (1+2*Tdelay+1.00156*n*l/bw) =

= (n-1)*(1+2*Tdelay)
```

In the following section we present a sample application to evaluate how the number of remote method calls affects distributed application performance in our test environment. We will then introduce the technique of packaging remote method calls and discuss its effect on performance.

1.4. Problem Statement

In the previous section we have discussed and evaluated the performance of remote method calls and we have shown how factors like network bandwidth, network delay and method result length affect the client waiting time. In this section, we focus on the application level to evaluate how the number of remote method calls affect distributed application performance. We introduce a sample application and evaluate the performance of this application. We will then use the performance test results to

1. discuss the relationship between the number of remote method calls and the performance of a distributed application,
2. discuss a number of current-practice approaches which can be applied to speed up distributed application performance, and
3. discuss current-practice approaches in terms of performance gains and development effort.

1.4.1. Address Book Sample Application

The sample application is a distributed client-server system that is used to store and retrieve data about persons, like their name, email addresses and phone numbers. Moreover, users can create, edit and remove person entries and can use a search function to search person entries by name. The use case model of the Address Book application is shown in Figure 19.

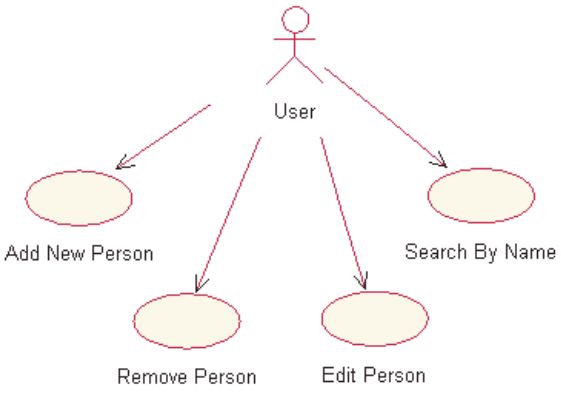


Figure 19: Address Book Use Case Model

For the rest of this section, we concentrate on the ‘Search By Name’ use case, since this is sufficient for the discussion of the performance aspects. The analysis class model of the Address Book application is rather simple, as shown in Figure 20: Only the methods needed

by the ‘Search By Name’ use case are shown. The class `AddressBook` is a container for the `Person` objects. Each `Person` object has a name, an email address and a phone number, which can be retrieved by get-methods.

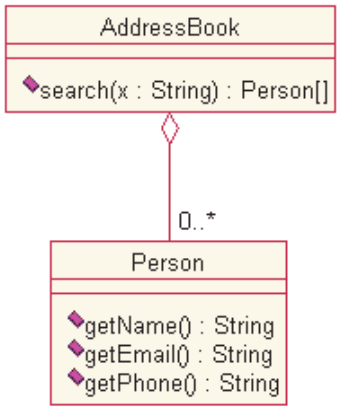


Figure 20: Address Book Analysis Model

A sample deployment of the Address Book application is shown in Figure 21: The Address Book Server provides access to `Person` objects to Address Book Clients. A Database Server stores the data of all persons.

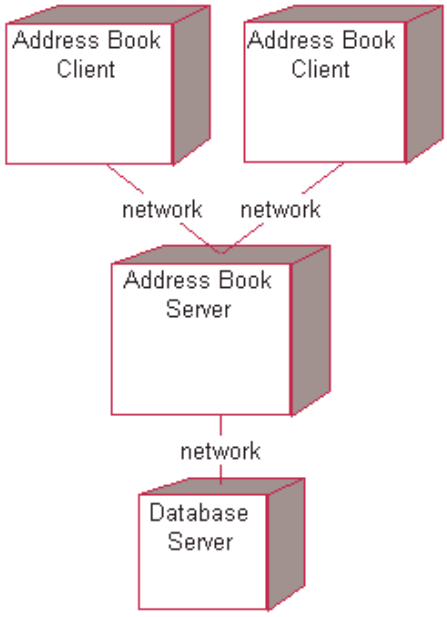


Figure 21: Address Book Deployment Diagram

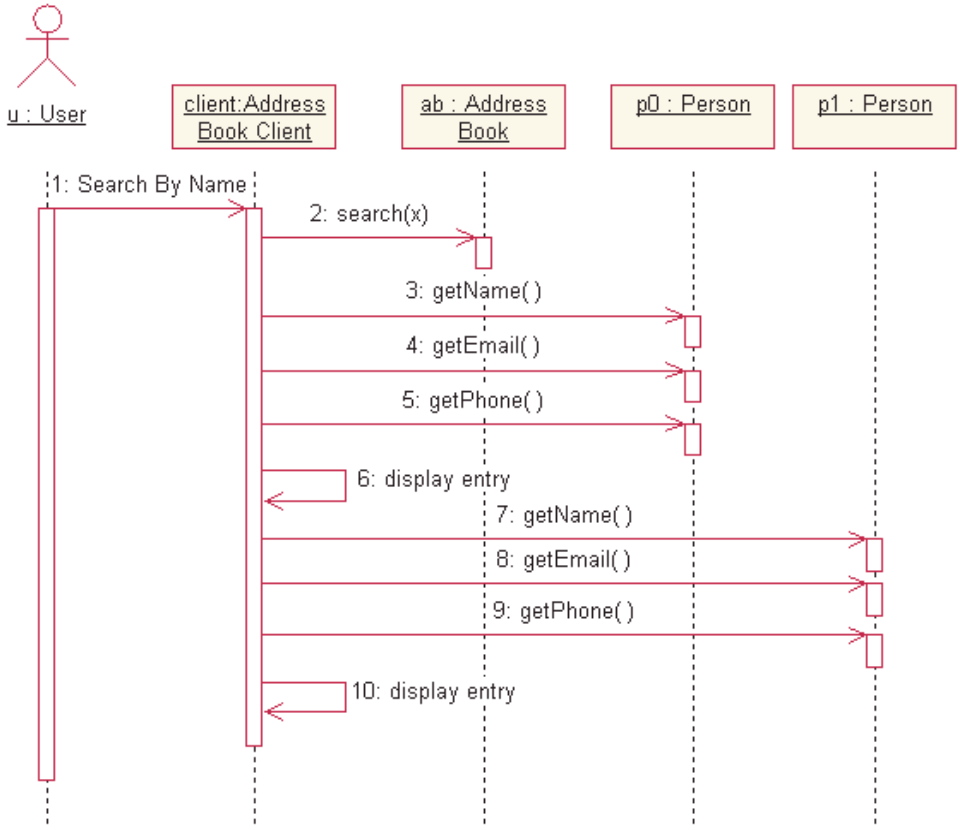


Figure 22: Search And Display Person Entries

The sequence diagram in Figure 22 illustrates the steps required for searching for a person and displaying the search result. After user `u` initiates 'Search By Name', the remote method call `search()` is sent to the server, with a search expression `x` as method parameter. The result of this method call is a list of `Person` interfaces. In this example, the search yields a result of two persons. To display the search result, three remote method calls `getName()`, `getEmail()` and `getPhone()` are initiated for each `Person` entry.

More general, the number of method calls, `Nm`, depends on the size of the search result list:

```
Nm = 1 + Na * Np
```

`Nm` is the number of remote method calls, `Np` is the number of `Person` entries returned by `search()` and `Na` is the number of attributes per `Person`. The time to get the Person list and iterating over it, calling three `get()` methods for each `Person` entry is called the 'cycle time'.

1.4.2. Address Book Implementation

We now describe a CORBA [10] implementation of the Address Book sample application, according to the development process introduced in Figure 14. Starting with the interface specification, we will describe the code generation of `ClientStub` and `ServerStub` source code. Then we present pseudo code fragments of the client and server implementations. Finally we will show the interaction between the application client, the stub and adapter objects, the CORBA system and the server implementation.

CORBA is an abstract specification of a Distributed Object Computing system and defines the stub protocol (see Figure 9) of remote method calls, mappings of objects to object references and remote lookup services for finding and locating remote objects. Moreover, the CORBA specification [10] describes a Naming Service that client applications use to find objects across a network, and vertical application-dependent services to be used, e.g., for medical applications and banking systems. In this dissertation we focus on the remote method call mechanism provided by the CORBA specification. The CORBA implementation that is used throughout this dissertation is described in Appendix 9.1.

In CORBA, the server interface is defined by an abstract interface definition language called CORBA IDL. The IDL definition serves as a contract between client and server and describes the data types, interfaces and methods that the server provides to clients. The abstract interface definition shown in Figure 23 is derived from the analysis class model of Figure 20 and specifies the list of object interfaces, data types and method signatures that the Address Book sample server implements. Since CORBA IDL does not provide object arrays, a new collection `PersonList` is defined, which holds a list of `Person` instances (line 7 in Figure 23).

```
1  interface Person
2  {
3          string getName();
4          string getEmail();
5          string getPhone();
6  };
7  typedef sequence<Person> PersonList;
8
9  interface AddressBook
10 {
11         PersonList search( in string x );
12 };
```

Figure 23: Address Book IDL Definition

An IDL compiler reads the IDL definition file and generates source code for the `ClientStub` and `ServerStub` classes in the destination language, which is Java [50] in our case. Note that IDL is language independent and IDL compilers exist for other destination languages

such as C/C++, Ada, and Perl. The CORBA standard specifies the mapping of IDL entities and data types to destination language classes, interfaces and data types. Since we are using Java as destination language, each IDL-defined interface generates a Java interface, a `ClientStub`, a `ServerStub` and a `Helper` class. Note that the CORBA nomenclature of stub objects differs from the names in our Distributed Object Computing Framework presented in section 1.2. According to the CORBA standard, a trailing '`Stub`' stands for a `ClientStub` type and a trailing '`POA`' (Portable Object Adapter) stands for a `ServerStub` type.

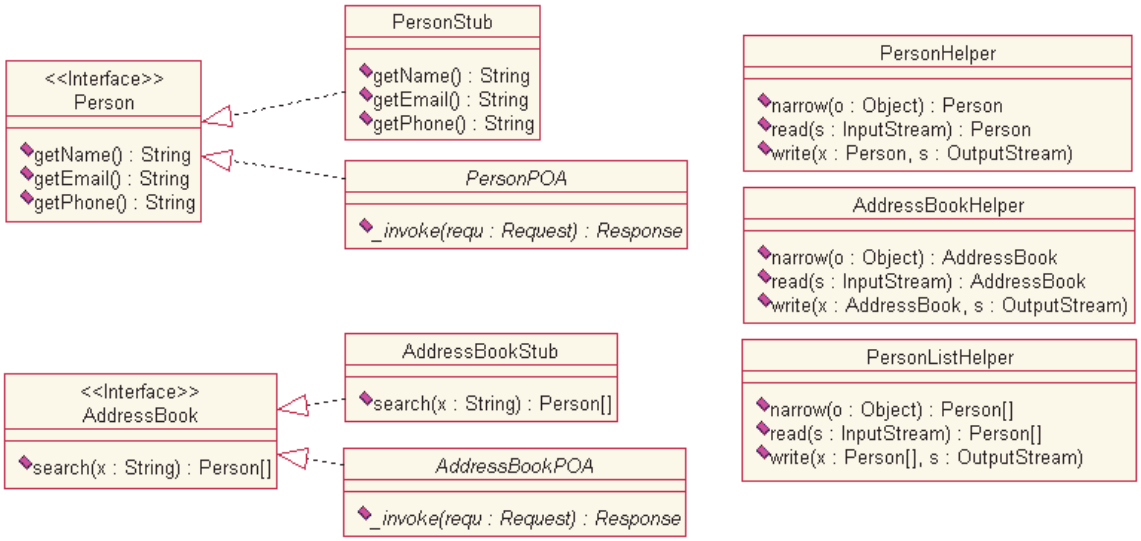


Figure 24: AddressBook IDL Compiler Output

Figure 24 shows the generated classes for the IDL entities `AddressBook`, `Person` and `PersonList`. The interfaces `AddressBook` and `Person` define the methods that are specified via IDL. The client stub classes `PersonStub` and `AddressBookStub` extend the respective interfaces and provide implementations for the inherited methods. These implementations handle parameter marshalling, creating and sending `Request` objects, receiving `Response` objects and unmarshalling method result values. The server stub classes `PersonPOA` and `AddressBookPOA` provide the `_invoke()` method that is called by a `RequestDispatcher` to forward incoming `Requests`.

The `POA` classes do not provide implementations of the IDL-defined methods. Instead, implementation classes (shown in the next section) must extend the `POA` classes, and must provide an implementation for each method.

The helper classes `PersonHelper` and `AddressBookHelper` provide methods for writing and reading instances of the respective classes to and from data streams. A `narrow()` method provides for type-safe casting (also called 'narrowing') an untyped object reference to an

object reference of the desired type. For the data type `PersonList`, no stub classes or interfaces are generated. Only a helper class, which is used to narrow, read and write `PersonList` objects is provided.

Server Implementation

The generated source code described above handles the mechanism of remote method calls: Sending `Requests`, waiting for `Responses`, dispatching incoming method call `Requests` to server stubs and so on. The actual implementations of the IDL-defined methods are still missing. The Address Book implementation classes are shown in Figure 25. The shaded interfaces and classes were generated by the IDL compiler, see Figure 24.

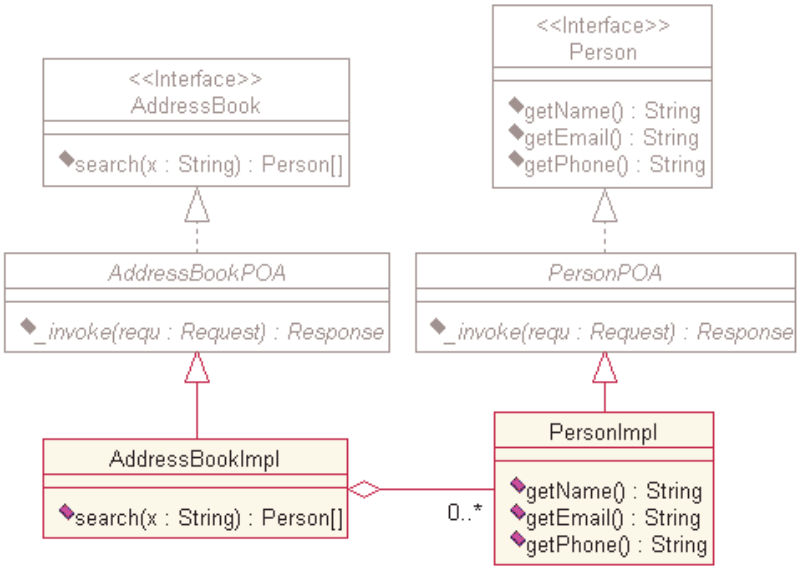


Figure 25: Address Book Implementation Classes

By convention, we use the suffix '`Impl`' for implementation classes. `AddressBookImpl` holds a list of `PersonImpl` objects. Both implementation classes, AddressBookImpl and `PersonImpl`, are subclasses of their respective `POA` classes, thereby inheriting the ability to receive remote method call `Requests` and send back `Response` objects.

Figure 26 presents a simplified pseudo code of the server implementation. Upon startup, the server creates an `AddressBookImpl` object (line 5) and registers it with the CORBA Naming Service, a system-wide registry that maps string names to object references (line 6). It then enters the CORBA main loop by calling `wait_for_invocations()`, a method that blocks until the user shuts down the server program (line 7).

```
1  public class Server
2  {
3          public static void main()
4          {
5                  AddresssBookImpl aBookImpl = new AddresssBookImpl();
6                  CORBA.NamingService.register(aBookImpl, "Address Book");
7                  CORBA.wait_for_invocations();
8          }
9  }
10
11 public class AddresssBookImpl extends AddressBookPOA
12 {
13         private PersonImpl[] PersonList;
14
15         public AddresssBookImpl()
16         {
17                 PersonList = load_DB();
18         }
19
20         public Person[] search( String x )
21         {
22                 Person[] resultList = new Person[0];
23                 for each p in PersonList
24                 {
25                         if p.matches(x) then add(resultList,p);
26                 }
27                 return resultList;
28         }
29 }
30
31 public class PersonImpl extends PersonPOA
32 {
33         private String Name;
34         private String Email;
35         private String Phone;
36
37         PersonImpl( String _Name, String _Email, String _Phone )
38         {
39                 Name = _Name;
40                 Email = _Email;
41                 Phone = _Phone;
42         }
43
44         public String getName()  { return Name; }
45         public String getEmail() { return Email; }
46         public String getPhone() { return Phone; }
47 }
```

Figure 26: Pseudo Code Of The Address Book Server Implementation

The *AddressBookImpl* – upon creation – loads a list of *PersonImpl* instances from a backend database (line 17). The implementation of the *search()* method (lines 20-28) iterates over the *PersonImpl* list and adds each *PersonImpl* that matches the search expression to a result list. This result list is then returned. The *PersonImpl* class is merely a container for the 3 attributes *Name*, *Email* and *Phone*.

Client Implementation

```
1  public static void doSearchAndDisplay(String x)
2  {
3      CORBA.Object obj_ref = CORBA.NamingService.resolve_name("Address Book");
4      AddressBook aBook = AddressBookHelper.narrow( obj_ref );
5      Person[] pList = aBook.search(x);
6      for each Person p in pList
7      {
8          displayPerson( p.getName(), p.getEmail(), p.getPhone() );
9      }
10 }
```

Figure 27: Pseudo Code Of The Address Book Client

Figure 27 presents the pseudo code for the Address Book sample client that implements the 'Search By Name' use case. To get an object reference for the initial *AddressBook*, the client queries the CORBA Naming Service (line 3). The object reference is then converted in an *AddressBook* type (line 4). This *AddressBook* object reference represents the server side *AddressBookImpl* object implementation.
After calling the *search()* method that returns a list of *Person* object references (line 5), the client displays the person list on the screen by iterating over the search result list, calling *getName()*, *getEmail()* and *getPhone()* for each *Person* (lines 6-9).

In the next section we will discuss the interaction of the application objects and the CORBA implementation when the client queries the CORBA Naming service, executes the *search()* method and displays the resulting list of *Person* attributes.

1.4.3. Remote Method Call Object Interaction

In this section we present the interaction of the application objects and the CORBA implementation based on the Address Book sample application. We assume that both the CORBA Naming Server and the Address Book server application are running and ready to receive requests. The first task of the Address Book client is to query the Naming Service to get an initial reference for an `AddressBook`.

Query Naming Service

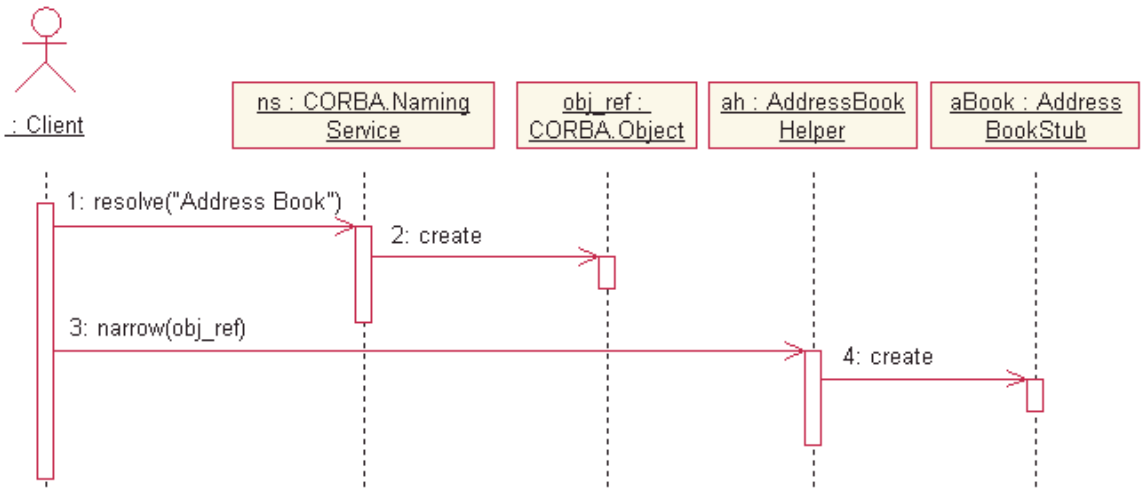


Figure 28: Query Naming Service

Figure 28 illustrates the steps that are executed when the client queries the CORBA Naming Service. The client calls the `resolve()` method of the CORBA Naming Service, which creates and returns an untyped object reference. That object reference represents a server side object implementation, an `AddressBookImpl` object in our case. The `AddressBookHelper narrow()` method is then used to convert the untyped object reference into a typed one: An `AddressBookStub` object is created and its `AddressBook` interface is returned to the client (see also Figure 27, line 4). So, the client can call all methods that are part of the `AddressBook` interface. The fact that the interface is implemented by an `AddressBookStub` is unknown to the client.

Execute Search Function

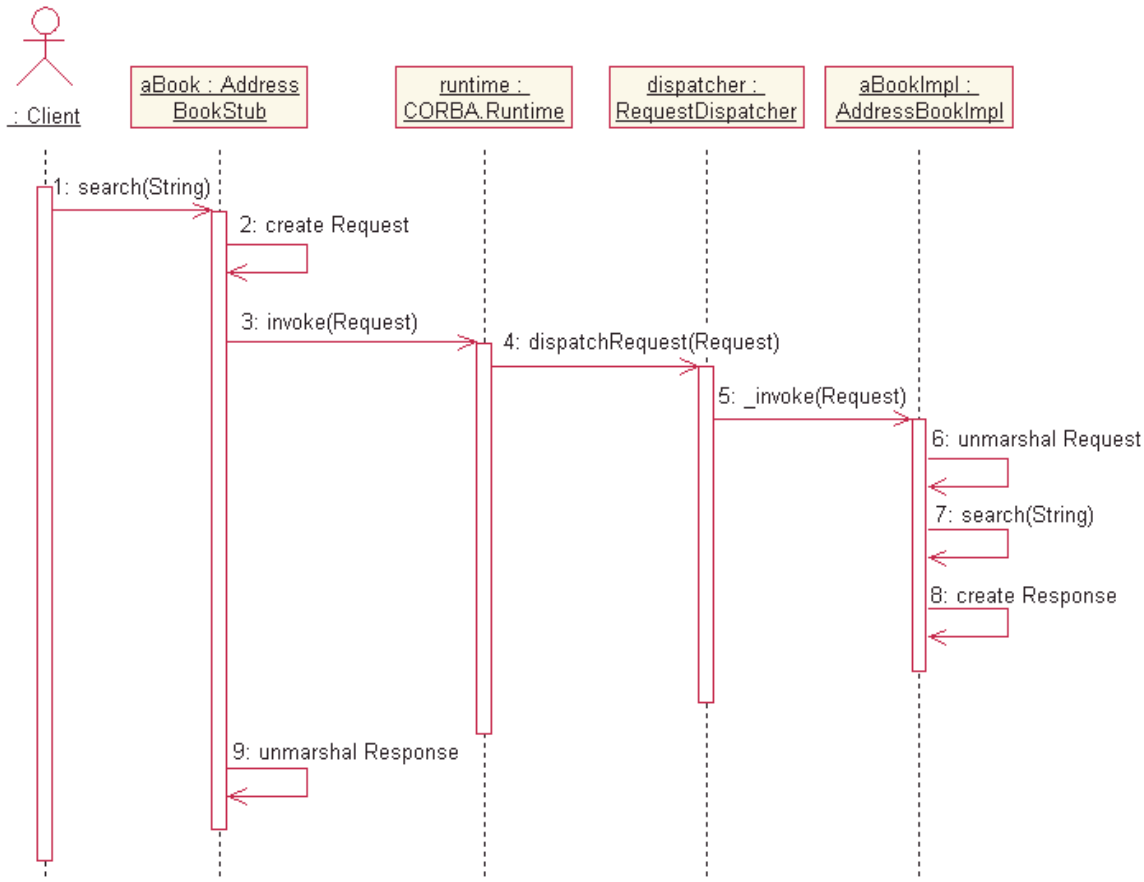


Figure 29: Execute The Search Function

In Figure 29, the client calls the `search()` method of the `AddressBookStub`. The stub creates a `Request` that contains a description of the method to be remotely called: The name of the method, an identifier of the object that should be called, and so on. The `Request` is forwarded to the CORBA Runtime system, which sends the `Request` over the network and waits for a `Response`. Note that we do not show network communication objects, as in Figure 12 and Figure 13. Instead, we assume that the network communication subsystem is contained in the CORBA Runtime object.

At the server side, the `RequestDispatcher` receives the `Request` and forwards it to the according object implementation, which is an `AddressBookImpl` in this case. The implementation object extracts the search expression string from the `Request` object and calls the method implementation of `search()`. The result value of `search()` is a list of `Person` objects. The `AddressBookImpl` object creates a `Response` object, inserts the

method result value into it and returns it back to the `RequestDispatcher`, where it is sent back to the client.

In the meantime, the `AddressBookStub` has been waiting for the `Response` of the `search()` remote method call. When the `Response` arrives, the `AddressBookStub` reads it from the network subsystem of the CORBA Runtime.

This process is shown in greater detail in Figure 30. The `AddressBookStub` uses a `PersonListHelper` to read a `PersonList`, which is actually an array of `Person` interfaces, from the network. The `PersonListHelper` reads the number of list entries from the `Response` data and then uses the `read()` method of a `PersonHelper` to read as much `Person` object references from the `Response` data as indicated. For each `Person` object reference, a `PersonStub` is created (in this example, the result list contains 2 entries). Finally, the list of `Person` objects is returned to the `AddressBookStub` and the client (see line 5 in Figure 27).

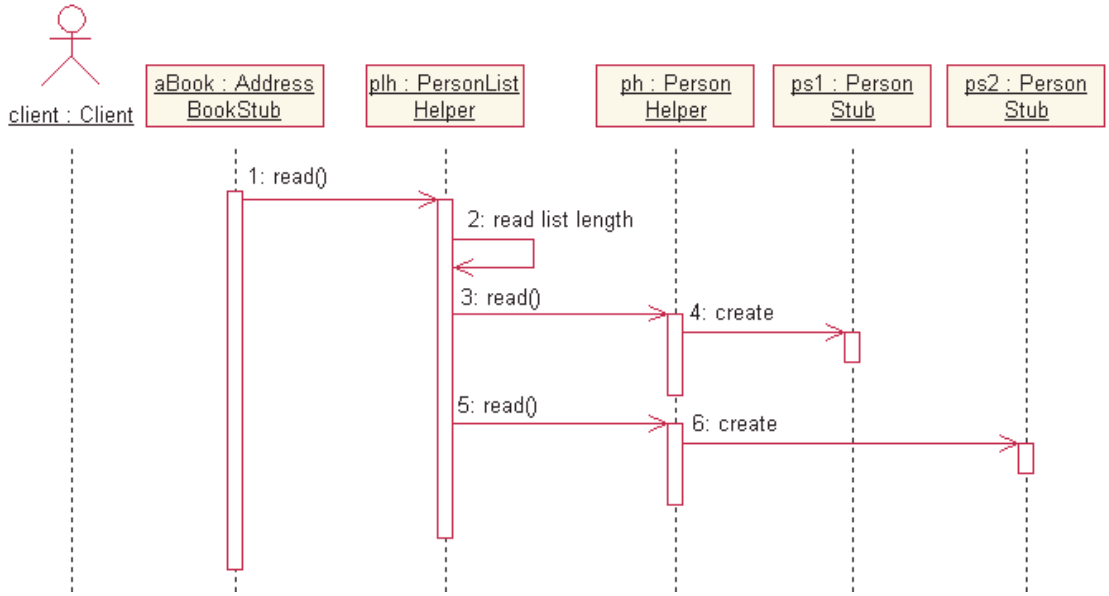


Figure 30: Read Search Result

Display Result List

After having received the `search()` result list, the client can now iterate over the result list and retrieve the attributes of all result list entries. The client calls `getName()`, `getEmail()` and `getPhone()` for each of the `Person` entries that matched the search expression and are returned by the `search()` method.

Figure 31 presents a sequence diagram of getting the name of a `Person`. The client calls the `getName()` method, not knowing that the receiver of the call is actually a `PersonStub` object that initiates a remote method call and receives a result value that it returns to the client. Calling `getEmail()` and `getPhone()` work analogous.

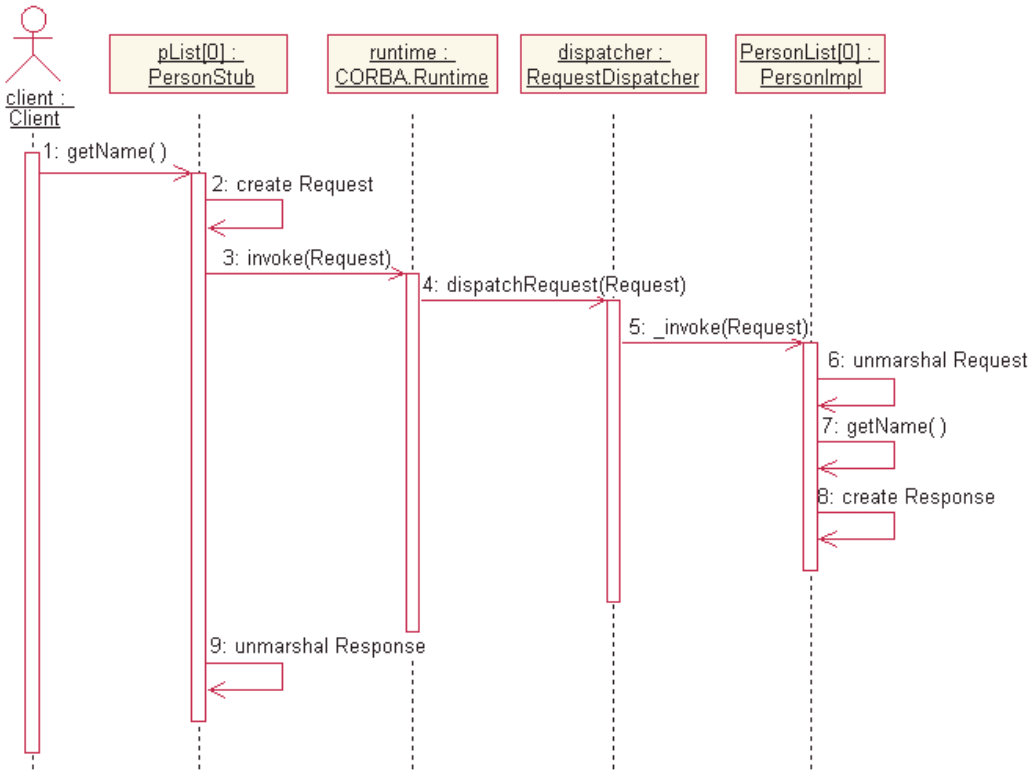


Figure 31: Get Attribute Value

1.4.4. Address Book Performance

In this section we present performance measurements of the Address Book sample application, based on the implementation presented in section 1.4.2 and the testbed configuration described in Appendix 9.1. For an exact description of all test run parameters as well as a complete list of measurement result values see Appendix 9.3

A test run consists of fetching the `Person` list by calling `search()` and then iterating over the result list of `Person` entries. The length of the result list returned by `search()` was set to a fixed value of `Np=10` and the number of attributes per `Person` was set to `Na=3`, therefore the number of remote method calls per test run was `Nm=31`. Each attribute had the fixed

length of 10 characters. Figure 32 shows the test run times for different network delay and bandwidth configurations.

Search Result Length (Np)	*Fixed*	*Np = 10*
Attributes Per Person (Na)	*Fixed*	*Na = 3*
Attribute Length (l)	*Fixed*	*10 Characters*
Network Delay	*Variable*	*0 to 160 ms*
Network Bandwidth	*Variable*	*50 to 1600 Kbyte/s*

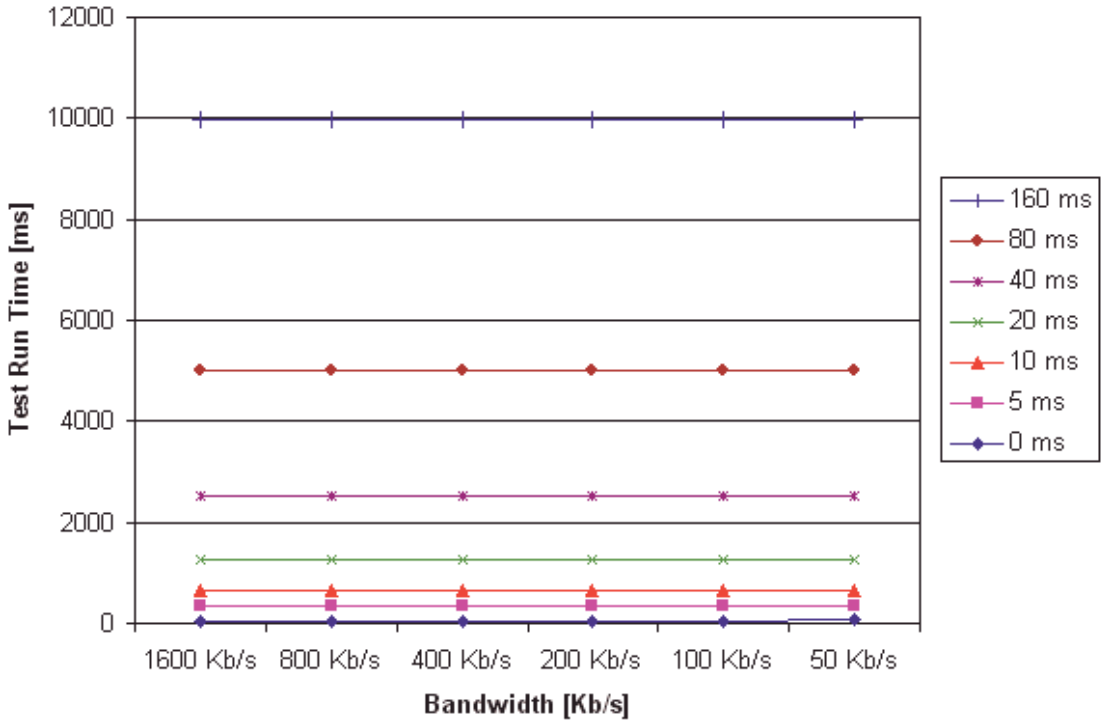


Figure 32: Test Run Time For Different Network Parameters

The most important result of the test run is that the test run time (the time it takes to retrieve the person list and the attribute values of its entries) is determined almost exclusively by the network delay. The influence of the bandwidth is negligible. Since the marshalling time is very small compared to the network time, it is negligible, too, except in the case where the network delay is zero, in this case $Tmar > Tnet$. The result of the test run is that the application performance is determined mostly by the network delay and the number of remote method calls per test run.

The second test run illustrates further how the number of remote method calls affect application performance. In this test run we varied the number of attributes of a Person from

Na=3 down to *Na=0*. Having this, the number of remote method calls decreases from *Nm=31* down to *Nm=1*, according to the formula presented in section 1.4.1.

Search Result Length (Np)	*Fixed*	*Np = 10*
Attributes Per Person (Na)	*Variable*	*Na = 0 to 3*
Attribute Length (l)	*Fixed*	*10 Characters*
Network Delay	*Variable*	*0 to 40 ms*
Network Bandwidth	*Fixed*	*Not limited (Full Ethernet speed)*

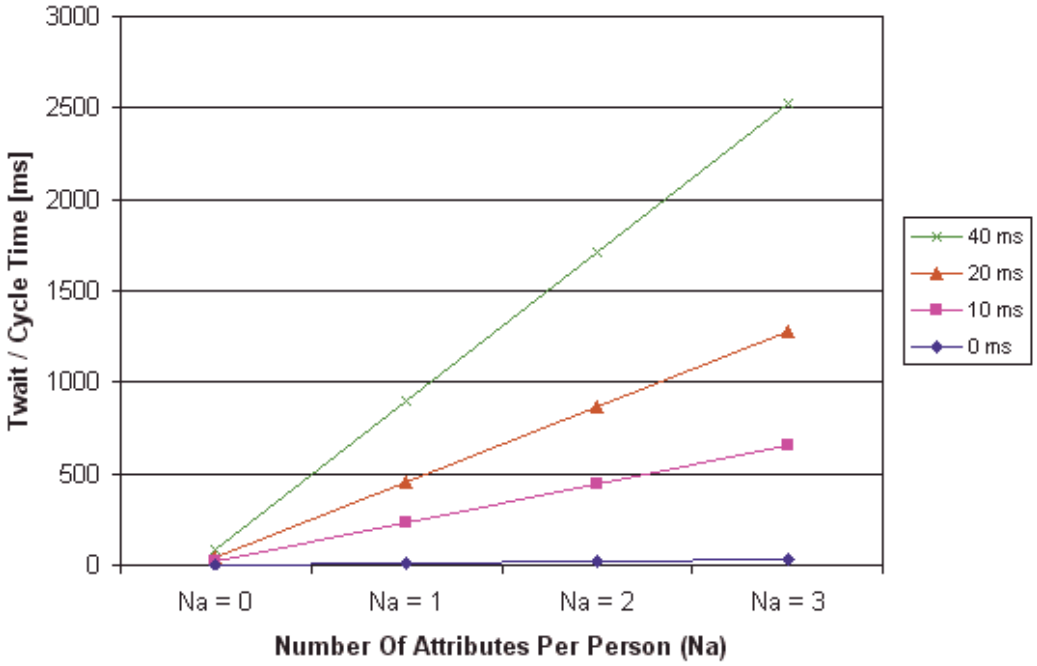


Figure 33: Test Run Time For Different Number Of Person Attributes

Figure 33 presents the performance measurement results in a graphical representation. The test run time is a linear function of the number of attributes and the number of remote method calls. In the case *Na=0*, only 1 remote method is called per test run, namely the *search()* method. In the case *Na=1*, 11 remote methods are called per test run: *search()* and *getName()* for each Person contained in the result list, where the number of Person entries is *Np=10*. The exact result numbers can be found in Appendix 9.3.

For real-world network installations, the network delay is determined by the physical hardware, the number and performance of the routers, switches, hubs, satellites, telecommunication relays, and any other network components between two communication endpoints. Moreover, the network delay is determined by the distance of two communication

endpoints. The larger the network delay gets, the more the distributed application performance is affected by the number of remote method calls. Actually, the issue of the number of remote method calls is the most important issue that makes single process application programming different from distributed object application programming.

In object-oriented programming it is common practice to have a large number of highly specialized methods. Having every method focus on one small aspect leads to fine-grained object interfaces. This is no problem as long as client and server object are in the same address space, since local method calls can be executed very fast. In fact, having a large number of highly specialized methods helps in dealing with the complexity of software development, as method concerns are focused and redundancy is kept low.

However, when the cost of executing remote method calls is high – as in distributed applications – programmers must strive for coarse-grained object interaction where the number of remote method calls is kept as low as possible. In practice, programmers of distributed applications use common practice approaches to reduce the number of remote method calls. Some of them are presented in the next section.

1.5. Current Practice

A straightforward transformation of a fine-grained analysis object model into an implementation object model can lead to severe performance drawbacks if the application is deployed over a distributed network of computers. Since every remote method invocation involves a full network roundtrip, performance goes down as the network delay value grows. Thus, the number of remote method calls must be kept as low as possible if application performance is critical.

In this section we present current practice approaches of reducing the number of remote method calls:

1. *Fat Operations*. “Combine several methods into one single method.”
2. *Data Structures*. “Give up object orientation by defining data structures that are passed by value.”
3. *Objects By Value*. “Pass objects by value, not by reference, and execute object methods locally.”
4. *Asynchronous Method Invocation*. “Use asynchronous invocation model.”

The descriptions of the approaches use the Address Book sample application introduced in section 1.4.1. We measure the performance gains and discuss the advantages and drawbacks of each approach.

1.5.1. Fat Operations

The idea of the “Fat Operations” design pattern [19] is to reduce the number of remote method calls by adding coarse-grained methods that combine the calls to several existing fine-grained methods in one call. The *AddressBook* interface defines three attributes per *Person*, each of them having its *get()* accessor method. Applying the Fat Operations design pattern, a new method is introduced that returns all attributes of a specific *Person* object in one single remote method call. Figure 34 shows the IDL definition of the Address Book application, introducing a new method *getAll()* that returns a *StringList*.

```
typedef sequence<string> StringList;

interface Person
{
        string getName();
        string getEmail();
        string getPhone();
        StringList getAll();
};
typedef sequence<Person> PersonList;

interface AddressBook
{
        PersonList search( in string x );
};
```

Figure 34: Fat Operations IDL Definition

The implementation of *getAll()* creates a new *StringList* object and inserts the Name, Email and Phone attribute values of the *Person* object implementation. The resulting *StringList* object is then marshalled and transferred back to the client in one piece. The client can now extract the attribute values from the *StringList* and display them on the screen. The *getAll()* method is called for each of the *Person* objects included in the *search()* result list. Therefore, the number of remote method calls in the Address Book sample is reduced to

$$Nm = 1 + Np * 1$$

where Nm is the number of remote method calls per test run and Np is the number of search result entries.

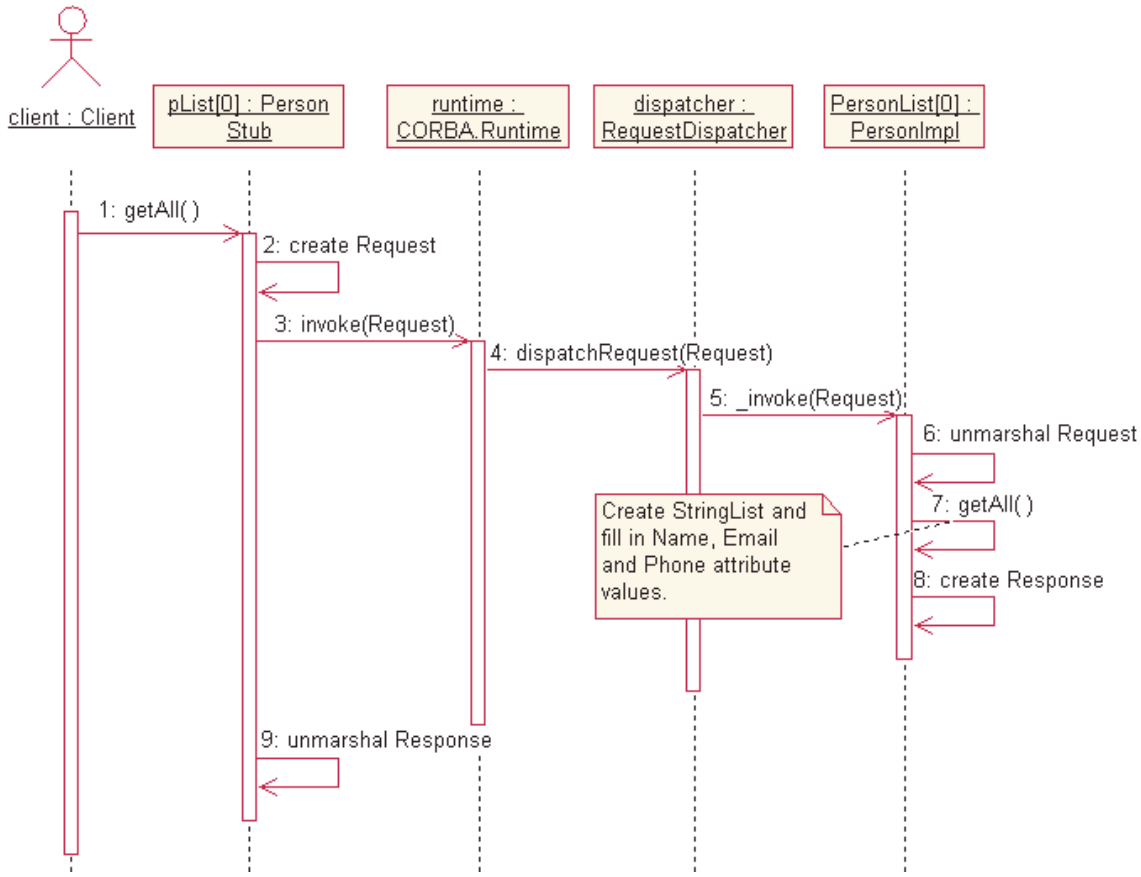


Figure 35: Calling A Fat Operation

Figure 35 presents a sequence diagram of calling the `getAll()` method. The sequence diagram looks similar to Figure 31, except with `getAll()`, all attribute values of a `Person` object are marshalled and transferred in one single remote method call, whereas is Figure 31 three remote methods must be called per `Person` object, one for each attribute.

By using fat operations, the number of network roundtrips can be reduced. In the Address Book sample application, the number of network roundtrips is no longer proportional to the number of attributes of a person. Instead, all attributes of a `Person` can be retrieved in one single remote method call. In general, the performance speedup of calling a fat operation depends on how many fine-grained method calls are saved by calling the fat operation.

However, these benefits do not come for free. The drawbacks of this approach are:

- *Implementation effort*. If a new method is introduced in the IDL definition, the `ClientStub` and `ServerStub` source code has to be re-generated and re-compiled. The client and server implementation has to be re-written and re-

compiled too, to support the new fat operations. After implementing, testing and documenting all affected parts of the software system, the new client and server software has to be shipped and deployed. Thus, introducing fat operations is expensive if an application is already deployed.

- *Reuse*. Most fat operations are tailored to one specific client task. If the Address Book client application has to display all attributes of a `Person`, the `getAll()` method is exactly what the client developer is looking for. However, if the client should only display the name and the email address of a `Person` object, the client developer has two choices. First, the fat operation `getAll()` is used anyway. Since the `getAll()` method transfers all attributes of a `Person` object, network- and server resources are wasted. Second, the client developer uses `getName()` and `getEmail()` directly, in which case the network latency problem comes up again. In any case, the client developer should have a method `getNameAndEmail()`, which fulfills the client's needs in one single network roundtrip.
  Anticipating what the client exactly needs is not always possible. For example, in many address book applications that are commercially available, the user can configure which `Person` attributes are displayed on the screen. Ideally, the IDL interface of a distributed address book application would provide a `get()` method for each possible combination of `Person` attributes, which is generally not feasible, due to combinatory explosion.
- *Maintenance*. Fat operations introduce redundancy, with all its disadvantages. First, the principle of one-method-per-task and one-task-per-method is violated. Every fat operation does the tasks of all the fine-grained operations. Second, additionally maintenance is needed when the set of the fine-grained operations change. If for example a new method `getFirstName()` is introduced, the fat operation `getAll()` and all software parts that use `getAll()` must be updated accordingly. This is similar to databases, where controlled denormalization improves application performance at the cost of introducing redundancy into the data model.

1.5.2. Data Structures

The ‘Data Structures’ approach uses data structures in addition to object interfaces and returns result values ‘by value’ instead of ‘by reference’. Figure 36 shows – next to the original `Person` interface – a data structure `PersonData` that declares three attribute elements. When a client calls `searchData()`, the method implementation of `searchData()` constructs a sequence of `PersonData` structures, fills in the attribute values of all `Person` objects and returns the sequence to the client. In Figure 36, the client has the choice of getting a list of `Person` object references by calling `search()` or getting a list of `PersonData` structures by calling `searchData()`.

```
 1  interface Person
 2  {
 3          string getName();
 4          string getEmail();
 5          string getPhone();
 6  };
 7  typedef sequence<Person> PersonList;
 8
 9  struct PersonData
10  {
11          string Name;
12          string Email;
13          string Phone;
14  };
15  typedef sequence<PersonData> PersonDataList;
16
17  interface AddressBook
18  {
19          PersonList search( in string x );
20          PersonDataList searchData( in string x );
21  };
```

Figure 36: Data Structures IDL Definition

Since the sequence of data structures is marshalled and transferred over the network in one piece, a performance speedup is achieved by reducing the number of remote method calls. In case of the Address Book sample, the number of remote method calls per test run (get the `PersonDataList` and iterate over it) is reduced to

$$Nm = 1$$

where Nm is the number of remote method calls per test run.

Figure 37 show a sequence diagram where a client calls the `searchData()` method to retrieve a list of `PersonData` data structures from the Address Book server in a single remote method call.

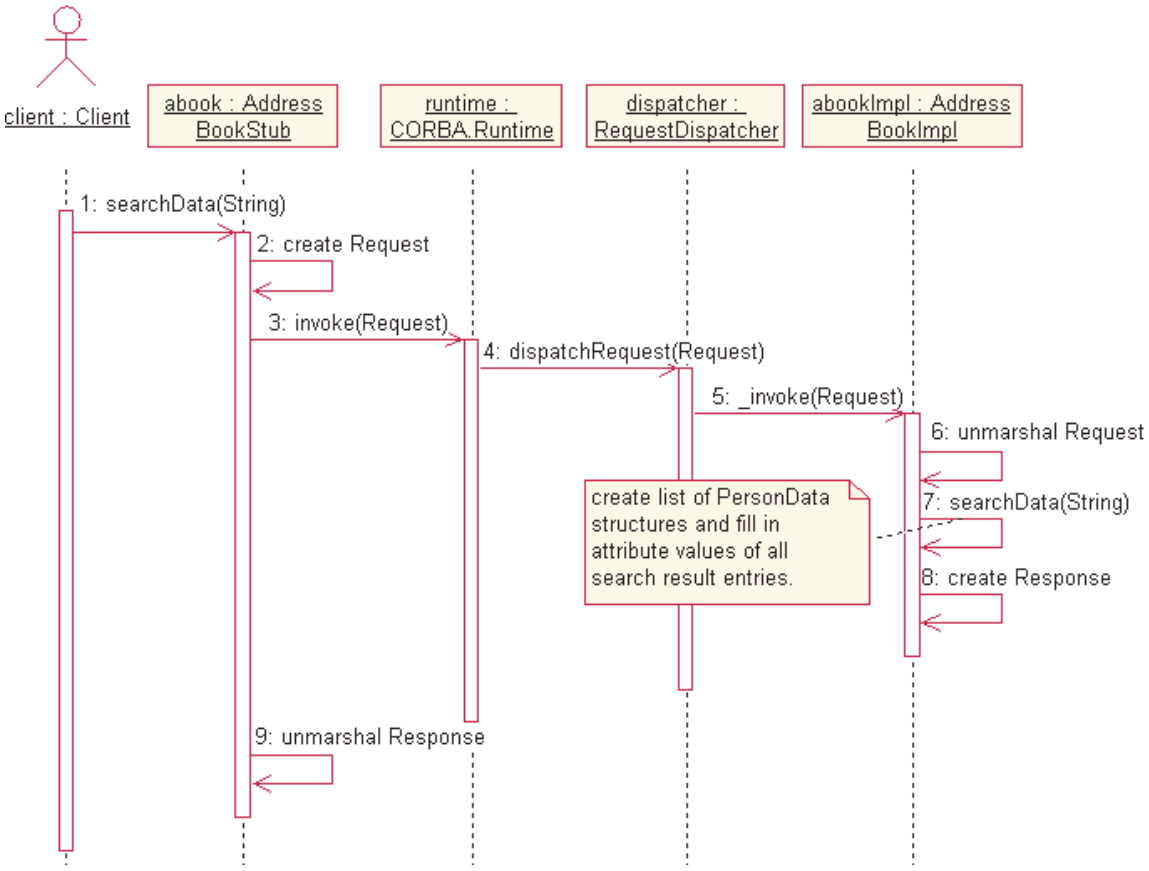


Figure 37: Using Data Structures

The 'Data Structures' approach has the following drawbacks:

- *Implementation effort*. Integrating this approach into an already existing and deployed system yields a considerable implementation effort. Since clients and servers have to be extended to support the newly introduced data structures, the software development has to go through a full design/implement/deploy cycle. Another issue that has to be considered when using data structures is data staleness. If a client receives a sequence of `PersonData` objects and holds it for a long time, it has to make sure that it fetches the list from the origin server from time to time to avoid working with old and timed-out data. When using interface methods like `getName()`, `getEmail()`, data freshness comes for free, as each method result is fetched from the server each time.

- *Reuse*. As with fat operations, data structures are tailored to a specific client task. The attributes of the data structure should match the client task as precisely as possible. If the client needs – for a specific task – more attributes than held by the data structure, it has to revert to the original object-oriented interface, which results in additional network roundtrips. If on the other hand the data structure defines more attributes than the client needs, network time and server resources are wasted. This may not be harmful with small data values, but if the Address Book application were designed to display bitmap images too, marshalling and transferring these images in vain would be a waste of resources, especially if large high-resolution images are used. So the benefit of data structures depends on whether the IDL developer can anticipated the clients' needs, which is generally not possible but for the near future.
- *Maintenance*. Since data structures are self-describing, adding or removing an attribute is straightforward, but not without implementation effort. When introducing a new method or attribute for an object interface, the according data structure in the IDL specification may have to be updated, too, as well as the server and client source code.
- *No Object-Orientation*. A further drawback of the data structures approach is that data structures are values instead of objects. Therefore, there are no references to data structures that can be passed around like object references, hindering the use of pass-by-reference call semantics. Another drawback is that there is no inheritance between data structures, which restricts the reusability of data structures. Finally, since structures are passed by value, modifying an attribute value is only local and does not update the server state.

1.5.3. Objects By Value

The ‘Objects By Value’ approach [11] is an extension to the CORBA standard and fills the gap between CORBA interfaces and CORBA data structures by introducing a new IDL language construct ‘value object’. The CORBA standard states that

> “…there are many occasions in which it is desirable to be able to pass an object by value, rather than by reference. This may be particularly useful when an object’s primary ‘purpose’ is to encapsulate data, or an application explicitly wishes to make a ‘copy’ of an object.”.

The idea of the Objects By Value approach is to have a language construct ‘value object’ that acts part as an object and part as a data structure. A value object differs from a regular object in that it potentially carries attributes and that the method implementations are executed locally. When used as a remote method result value, a copy of the value object is created on the receiver’s side, thereby employing pass-by-value semantics.

```
 1  interface Person
 2  {
 3          string getName();
 4          string getEmail();
 5          string getPhone();
 6  };
 7  typedef sequence<Person> PersonList;
 8
 9  valuetype PersonV supports Person
10  {
11          private string Name;
12          private string Email;
13          private string Phone;
14  };
15  typedef sequence<PersonV> PersonVList;
16
17  interface AddressBook
18  {
19          PersonVList search( in string x );
20  };
```

Figure 38: Objects By Value IDL Definition

When the Objects By Value approach is applied to the Address Book sample application, the IDL interface definition looks like shown in Figure 38. The interface `Person` with its `get()` accessor methods remains untouched. The new value object `PersonV` (declared with the ‘`valuetype`’ keyword in Figure 38) declares 3 class-private attributes. It is further declared as ‘`supports Person`’, which means that it implements the `Person` interface with its accessor methods `getName()`, `getEmail()` and `getPhone()`. The implementation of these

`get()` methods simply return the private attributes of the value object. The declaration of the `search()` method is modified in that it returns a list of `PersonV` value types instead of `Person` object references.

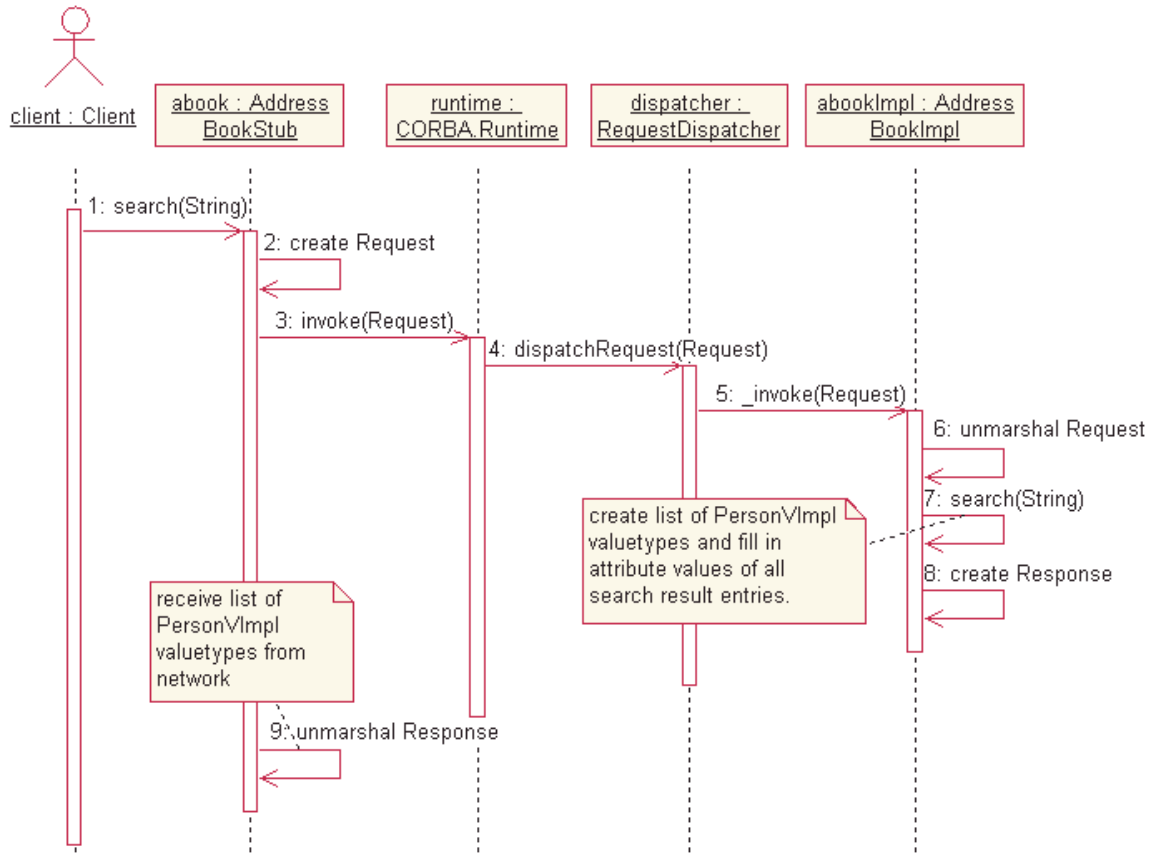


Figure 39: Objects By Value

If the client now calls `search()`, as shown in Figure 39, a list of `PersonVImpl` objects is created by the method implementation of `search()`. Since `PersonV` is defined as value object, the state of each value object (the attributes `Name`, `Email` and `Phone`) is marshalled and transferred back to the client side. The unmarshalling process in the client-side `CORBA Runtime` instantiates new `PersonVImpl` value objects, restores their state, and returns the result list to the client. The client can now call `getName()`, `getEmail()` and `getPhone()` on these value objects like with `Person` objects, with the notable difference that the `get()` methods are now implemented by the local `PersonVImpl` objects and do not initiate remote method calls. The `PersonVImpl` implementations of the `get()` methods simply return the value of the private attributes `Name`, `Email` and `Phone`.

The number of remote method calls for one test run (execute `search()` and iterate over the result list) is now

$$Nm = 1$$

where Nm is the number of remote method calls per test run.

The difference between data structures and value objects is that data structures are simple record-type values containing data, whereas value objects are real objects that contain data and code. The difference between value objects and normal CORBA remote objects is that remote objects are located in the server and clients use objects references to refer to remote objects. When a client invokes a method on an object, the method call is forwarded to the object implementation in the server. Value objects are always local to the client, so calling a method on a value object is executed in the address space of the client. No remote method call is initiated. Based on this behavior, value objects provide semantics of pass-by-value similar to that of standard programming languages. Switching from pass-by-reference to pass-by-value semantics can reduce the number of remote method invocation.

The Objects By Value approach has advantages over Data Structures, including the following:

- *Reuse*. Since value objects can encapsulate class-private data, the information hiding principle is maintained. Moreover, value objects support interface inheritance. Therefore, using value objects has advantages over data structures as far as reuse is concerned.
- *Maintenance*. Since valuetypes are self-describing, adding or removing an attribute or a method is straightforward and the extra effort for remote operations is low. The encapsulation of object references in valuetypes makes it possible to implement some methods as remote methods, whereas other methods of the same value object can be implemented locally. The decision where the implementation of a method resides – locally on the client or remote on the server – is private to the value object and can be changed without the need to update any software that is using the value object.

However, using the Objects By Value approach has some disadvantages over using object references and remote method calls, including implementation effort:

- *Implementation effort*. Introducing value objects provides backward compatibility for the client implementation. Since value object can be declared as extensions to already existing interfaces, an already existing and deployed client continues working. However, server implementations have to be updated and extra code for the value object implementation classes has to be written, so introducing value objects does not come for free in terms of implementation effort. Also, as with data structures, the issue of data staleness has to be addressed.

1.5.4. Asynchronous Method Calls

Distributed Object Computing systems are designed to mimic the behavior of local method calls: The client calls a method, waits until a result value is available and then goes on with the next instruction. The advantage of this synchronous programming model is that it is well understood by developers. However, sometimes it is desirable to let a client continue with the execution while a method implementation at the server is in progress. The client can fetch the method result later, either by polling, or by being notified when the method implementation has finished executing and the result value is available. The 'CORBA Asynchronous Messaging Interface' [12] includes a specification for an asynchronous programming model: The client calls a method but instead of being blocked until the result value is available, a placeholder object is created and control is given back to the client immediately. When the result value is available, it is stored in the placeholder object and can be retrieved later by the client.

```
1  interface Person
2  {
3          string getName();
4          string getEmail();
5          string getPhone();
6  };
7  typedef sequence<Person> PersonList;
8
9  interface AddressBook
10 {
11         PersonList search( in string x );
12 };
```

Figure 40: Asynchronous Method Calls IDL Definition

The Asynchronous Messaging Interface definition was introduced for situations where a client could continue to do useful work during the time of the method execution, for example when the method implementation executes a complex database query. In the following we experiment with the Asynchronous Messaging Interface, executing performance measurements, to see if an asynchronous method call model can speed up the Address Book sample application. The client task of getting the attributes of a `Person` entry is presented in Figure 41. The client calls the `get()` methods of the `PersonStub` and is not blocked. It is not defined when and in which order the `get()` method calls are forwarded to the `PersonImpl` object. Later, the client uses the `ReplyHandler` placeholder object to retrieve the method results.

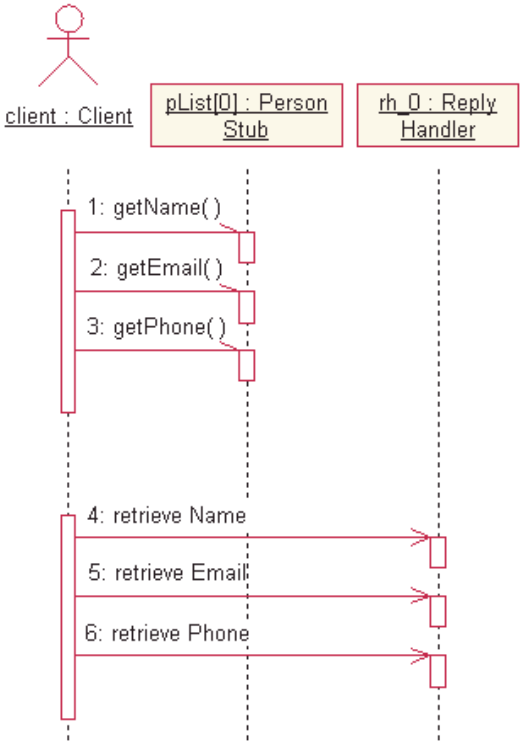


Figure 41: Asynchronous Method Calls

The advantage of the Asynchronous Invocation Model is that the IDL definition of the software system does not need to be changed - the IDL definition shown in Figure 40 is the same as the original Address Book IDL definition – thus, reuse is high. Moreover, the Asynchronous Messaging Interface is entirely a client side feature, the server implementation is never aware whether it is invoked synchronously or asynchronously, which means that server implementation code does not need to be modified when using the Asynchronous Messaging Interface.

However, the disadvantages of using the Asynchronous Messaging Interface are:

- *Implementation effort*. The implementation of the client has to be modified to query the return values asynchronously. The issue of concurrent execution and race conditions has to be addressed, for example by definition of critical sections.
- *Maintenance*. Introducing asynchronous operations to any software system yields extra maintenance effort, because when two methods `a()` and `b()` are called sequentially, there is no guarantee that the result values are available in the same order. This is a characteristic that is so integral to standard programming languages that giving it up leads to code that is hard to maintain. For example, if method parameters depend on result values of other methods, which must be

called previously, mechanisms of distributed concurrency control must be implemented, adding complexity to the system source code.

1.5.5. Performance And Discussion

In this section we present performance measurement results of the current practice approaches discussed so far. The hard- and software configuration of the test bed is described in Appendix 9.1. Here, we only present summaries of the test runs, as well as an interpretation of the measurement results. An exact description of the test runs as well as the complete list of measurement result values is included in Appendix 9.4.

Figure 42 shows a graphical representation of the client waiting time `Twait` for each of the current practice approaches, Figure 43 shows the network time `Tnet` for each approach. One can see from the diagrams and the result numbers listed in Appendix 9.4 that for the Address Book sample, the main component that adds to the client waiting time `Twait` is the network time `Tnet`. Only in the case of asynchronous method calls, there is a major gap between *Tnet* and *Twait*, as the marshalling time `Tmar` is relatively high in this case.

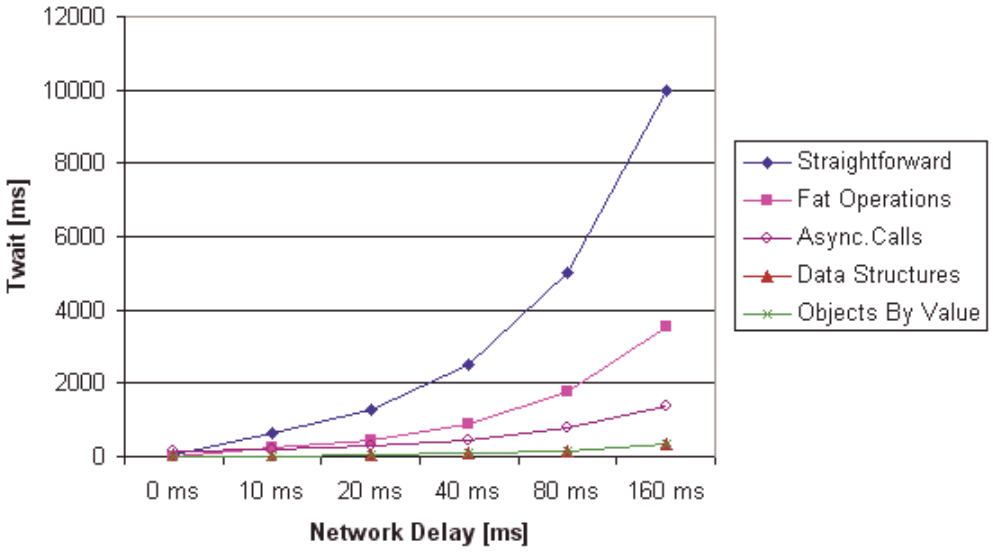


Figure 42: Test Run “Current Practice” / Twait Values

While using the Fat Operations approach can speed up the Address Book performance, using the Data Structures or Objects By Value approach is far more beneficial for performance, especially with high network delay times. Using the Data Structures approach or the Objects

By Value approach does not make any measurable difference in terms of performance. The performance of the asynchronous method calls lies inbetween the Fat Operations and the Data Structures and Objects By Value approach.

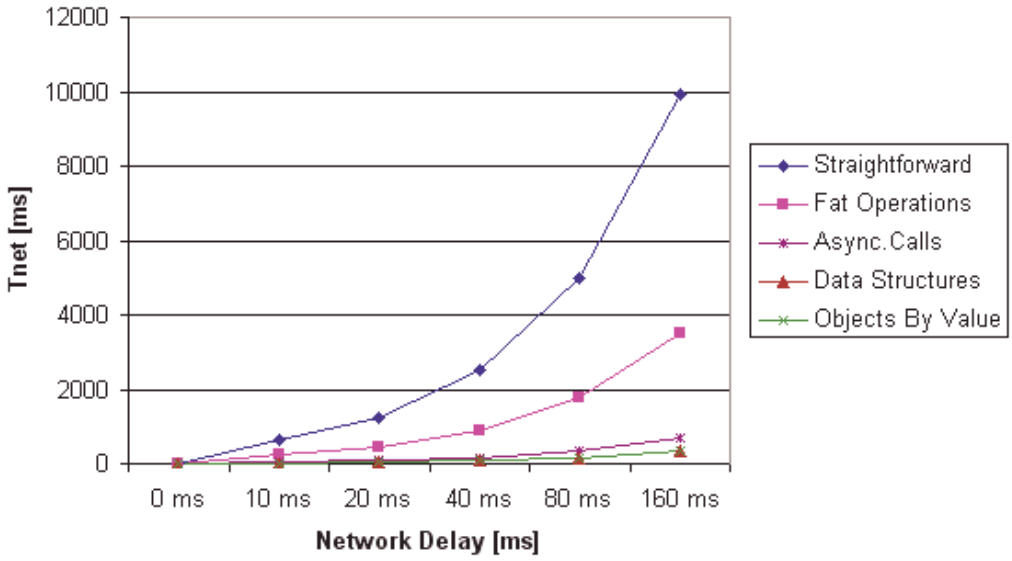


Figure 43: Test Run "Current Practice" / Tnet Values

The measurement result values (see Appendix 9.4) indicate that for the Address Book sample test runs:

- The waiting time *Twait* is dominated by the network time *Tnet*. The influence of the marshalling time *Tmar* is marginal, except in the case where the network delay time *Tdelay* is set to 0. In this case, the marshalling time *Tmar* is equal or greater than the network time *Tnet*.
- The network time *Tnet* is dominated by the number of remote method calls *Nm* and the network delay time *Tdelay*.
- Similar to *Tnet*, *Tmar* depends on the number of remote method calls: Marshalling many small request and response messages takes longer than marshalling all data into one large package. It is up to the Distributed Object Computing system to provide an explanation for this behavior. Examining the marshalling performance of existing Distributed Object Computing systems is out of scope for this dissertation. For the interested reader, we refer to Distributed Object Computing performance research, e.g.[18].

To summarize: All of the current practice approaches that we presented in this chapter can be applied to any distributed application system. The performance speed-up will depend on how

many remote method calls can be avoided by applying an approach. All approaches have the drawback that the application client and/or server implementation source code has to be modified to implement the changes made by introducing a current practice approach. Thus, the presented approaches are not transparent to client and server implementation.

The claim of this dissertation is that caching and prefetching remote methods can speed up distributed applications by avoiding network roundtrips. In the remainder of this dissertation we present such a caching and prefetching system, and a CORBA-based implementation of the system and its validation.

2. CACHING AND PREFETCHING

Remote method calls are costly and the performance of distributed applications can be substantially improved by reducing the number of remote method calls. The basic hypothesis of our work is that the number of remote method calls can be reduced by caching and prefetching method result values.

After describing caching and prefetching in general we address issues like cache consistency (the cache has to represent the data of the origin server), cache replacement (if the cache gets filled up, cached data items have to be evicted from the cache memory to make room for new data items) and prefetching prediction (the system has to decide which data items should be prefetched). An exhaustive overview over research work on caching and prefetching is beyond the scope of this dissertation, we concentrate on caching and prefetching techniques that are applicable to our work.

2.1. Caching

In this section we explain how caching and prefetching can be used to increase application performance and present approaches for ensuring cache consistency and managing cache replacement.

Caching of data items can be applied whenever there is a client/server architecture where the server provides data items to a client (see Figure 44a). Caching means that a copy of the data item requested by a client is stored as a cache item in a cache near the client (see Figure 44b), so that, if the same data item is requested again, the client can access the cache item faster than retrieving the data item from the server. Examples are network file systems and the Internet.

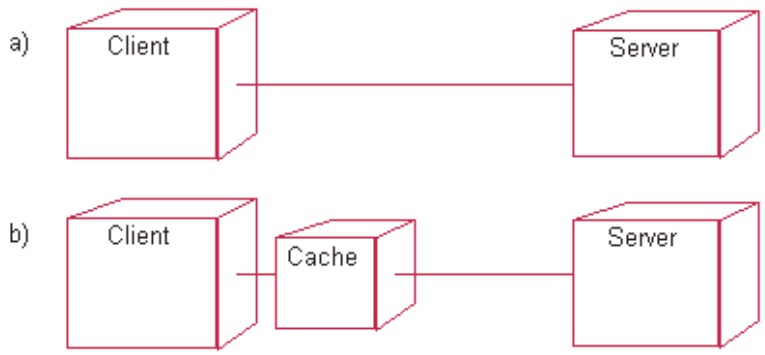


Figure 44: Client/Server Caching Architecture

A popular application of caching in the Internet is the Web: Web browsers communicate with Web servers to request Web documents like html files, images and multimedia content via the Hypertext Transfer Protocol (HTTP) [22] The Web browser acts as the client and the Web server acts as the server. Typically, these Web documents are stored in the file system of the Web server. Alternatively, a Web server can generate Web documents on-demand, which is the case for weather forecasts, stock numbers and news pages, for example. There are two common caches in the Web. One local at the client and one that is located on the network between client and server. The later is often called a Web proxy server[6].

In the case of network file systems, a server hosts files and any application program that reads a remote file acts as a client. Most often, a network file system is realized as an extension of the client operating system, where typical file operations like open, read, write and close are forwarded to the remote server, transparent to the client. The cache can either be the harddisk or the main memory of the client computer.

In all cases described so far, if requesting data items from the cache is faster than requesting it from the server, performance speedups can be gained by caching data items and the potential speedup that can be gained when accessing an already-cached data item is determined by a number of factors, among them the ratio of the client-server and the client-cache connection speeds and network delay times.

2.1.1. Cache Consistency

If the state of all cache items stored in the cache mirrors the state of the respective data items on the server, then the cache is called consistent. When a data item is updated on the server,

[6] A program that caches Web documents and acts as a Web server when contacted by a client.

but not in the cache, then the cache becomes inconsistent. If a client requests data that is served from the cache, out-of-date data may be delivered to the client.

In the following we present a classification of three types of inconsistency and then discuss how the issue of cache consistency has been addressed in network file systems and Internet Web caching. We distinguish three types of cache inconsistencies: local inconsistency, system inconsistency and global inconsistency.

Local inconsistency can occur when a client write operation bypasses the cache, for example when a client reads a data item from a cache or server, modifies the data item and then writes the same item back to the server. If the modification is not executed on the cache item, too, then the next read operation will return the cache item, which is out-of-date by now.

The second type of inconsistency is system inconsistency, where the source of cache inconsistency is not local to the receiver of out-of-date data, but within the system defined by the client and the server. For example, system inconsistency can occur when two or more clients are connected to the same server and execute concurrent read and write operations on the server. If a client reads a data item from the server, a copy of the data item is stored in the cache. If another client modifies the same data item on the server, then the cache item of the first client becomes inconsistent because it does not represent the state of the data item on the server.

Global inconsistency occurs when some backend data storage, which the server uses for storing data items, is modified by a third entity. Then the cache of each client that has previously requested and cached that data item is inconsistent.

In the following we briefly present four approaches for ensuring cache consistency. Strong cache consistency means that the cache of each client is consistent at any time, whereas weak cache consistency means that a cache may hold out-of-date data items, which is justifiable for a number of applications.

Expiration Model
In this approach, each cache item gets a timestamp (expiration time, also called ‘time-to-live value’) that expresses how long the cache item, once being received from the server and put in the cache, should be considered up-to-date. If a client requests a data item, the cache tests whether a cache item exists for the requested data item. If there is a cache item and it has not yet expired, it is delivered to the client. If there is no cache item or it has expired, the data item is fetched from the server and a new cache item, possibly with a new expiration time, is stored in the cache.

Client Validation
In this approach, the client validates the up-to-dateness of a cache item by checking the server for eventual modifications on the requested data item before taking the cache item from the

cache. If the cache item represents the current state of the respective data item, it can be taken from the cache, otherwise the data item is fetched from the server.

Server Invalidation
With server invalidation, the server, upon each modification of one of its data items, notifies all clients about the update. Each notified client then flushes its cache, thereby ensuring that the data items are fetched from the server the next time it will be requested. The granularity of the notification is application specific. If a client receives an update notification, it can decide to flush the complete cache or only for those data items that are reported to have changed in the server.

Leases
The Leases approach is an extension of the Server Invalidation approach. Here, the client tells the server that it wishes to be notified upon server modifications. The notification request is called lease and is valid for a certain time interval (lease-time). After the lease-time has elapsed, the client is no longer notified upon server modifications. It can then negotiate a new lease if it whishes to be notified again. Alternatively, the client can now switch to other cache consistency approaches like expiration model or client validation.

There exist more advanced cache consistency approaches than the ones described here, which can be not described in this dissertation due to the sheer number of approaches. Therefore we representatively focus on two cache consistency approaches, one for a network file system and one for the Internet.

The NFS [20] network file system allows clients to work on remote files using normal system calls like open(), read(), write() and close(). The fact that files are residing on remote hard disks is hidden by the NFS implementation. The NFS protocol uses RPC [07] for client/server communication. NFS clients cache recently used portions of files in order to increase the NFS performance by reducing the number of RPC calls to the server. Maintaining consistency between these caches is a problem whenever a client writes to a file and one or more other clients read the file, which is called ‘write sharing’ in NFS and can lead to system inconsistency according to our classification. If the writer closes the file before any readers open the file for reading, which is called sequential write sharing, NFS maintains cache consistency by requiring the writer to forward all the writes to the file server on close and having readers check to see if the file has been modified upon open. If the file has been modified, the client flushes the cache for this file and reloads it from the file server.
A more complex case is concurrent write sharing, where write operations are intermixed with read operations. Cache consistency in this case requires that a reader always receives the most recently written data. The NFS file system does not provide full cache consistency. The simplest mechanism for maintaining full cache consistency is the one used by the Sprite [37] file system, which disables all client file caching whenever concurrent write sharing might occur [21]. The Sprite file server maintains a list of currently open files and detects write sharing when a file open request for writing is received and the file is already open for reading or vice versa.

In the case of local inconsistency, cache consistency is ensured by the client flushing its own cache if a file is written. The case of global inconsistency cannot occur with network file systems, because all file modifications can only be done by the NFS server daemon, which is a program that manages all files that are potentially accessible by remote clients.

The Hypertext Transfer Protocol (HTTP/1.0) [78] is used for transferring Web documents like html files, images and multimedia files from Internet servers to clients. The goal of caching in HTTP/1.0 is to reduce the number of network round-trips and to reduce the amount of data that has to be transferred over a network. Moreover, caching helps in reducing the load of Web servers by intercepting HTTP/1.0 requests and serving them from caches.
To reduce the number of network roundtrips, HTTP/1.0 clients store copies of Web documents – once requested – in a local cache and reuse these copies when the documents are requested again. This approach is only applicable if documents are not subject to change. For example, dynamically generated Web pages (weather information, news tickers, and so on) are marked by the HTTP/1.0 server as 'non-cachable' and are therefore not cached by HTTP/1.0 clients or proxy servers. Documents that are not marked as 'non-cachable' may be marked with an expiration date that specifies when the document has to be considered out-of-date and must be reloaded from the Web server. In the Internet, global inconsistency can occur if a Web administrator uploads new html pages on a server and thereby modifies Web documents that may be already cached by clients or by proxy servers. When a client has a cached copy of one of these Web documents, it may use the cached copy as long as its expiration time has not yet expired, thereby possibly presenting the user an out-of-date Web document. After the expiration time has elapsed, the client fetches the Web document from the server, thereby receiving the new version of the Web document. A Web user who suspects that out-of-date Web documents are displayed by the Web browser can use the 'Reload' button of the Web browser to force the Web browser to fetch the Web document from the server, bypassing the cache.
A Web client may use the Client Invalidation approach to ensure that Web documents displayed to the user are up-to-date. Therefore, Web clients send to the server a 'conditional request', which contains the last modification date of the cached copy that the client possesses for the requested Web document. If the server has a newer version of the Web document, it sends it to the client. Otherwise the cached copy is used by the Web client. This approach reduces the network transfer volume between Web client and Web server.
The cache consistency strategy described by the HTTP/1.0 specification provides weak cache consistency for cachable Web documents by a combination of the Expiration Model and the Client Validation cache consistency approach. HTTP/1.1 [22] described more advanced caching mechanisms, the difference between HTTP/1.0 and HTTP/1.1 caching is described in [79].
Ongoing research work on Web caching indicates that strong cache consistency is feasible with little or no extra cost than current weak consistency approaches ([80], [81]).

2.1.2. Cache Replacement

Typically, the size of a cache is limited and if it gets filled up, the cache will have to evict some cache items to make room for new data. Two widely applied examples of cache replacement strategies are Least-Recently-Used (LRU) (see [44] for a description) and Least-Frequently-Used (LFU) (see [45] for a description), which are also used in other application areas, e.g., database systems and memory systems. LRU discards the least recently used items first, that is, the cache item that has not been used for longest time in the past is deleted from the cache. The LFU strategy maintains a counter for each cache item and increases the counter whenever a cache item is accessed by the client. The cache items that are used least often, i.e. that have the smallest counter value, are removed from the cache first. The SIZE replacement strategy discards the biggest cache items first. Cost, modification and expiration time (aging) are factors that are commonly integrated in variants of cache replacement strategies.

According to Belady [47] the optimal cache replacement strategy is the one which replaces the object that will not be used for the longest time in the future. Obviously, this optimum can only be achieved if the cache knows about all future data accesses, which is generally not the case. However, it poses the theoretical limit for a cache replacement algorithm.

In Internet caching, there are a number of approaches for cache replacement, a survey and a comparison of these approaches is presented by Podlipnig and Boeszoermenyi in [43].

2.2. Prefetching

While caching is used to gain performance when data items are accessed repeatedly, prefetching is used to speed up the very first access of a data item. The idea of prefetching (also called predictive caching or proactive caching) is to transfer data from the server to the client cache before the data is explicitly requested. At the time the client needs the data, it is already in the cache and can be accessed faster than fetching it from the server. In the following we explain how prefetching can be used to speed up caches and we present how prefetching is used in the Internet to speed up user-perceived Internet latency

As Dan Duchamp writes in his article [24] on prefetching Web documents: “The idea of prefetching Internet Web pages has surely occurred to many people as they used their browsers. It often takes ‘too long’ to load and display a requested page, and thereafter several seconds often elapse before the user’s next request. It is natural to wonder if the substantial time between two consecutive requests could be used to anticipate and prefetch the second request.”

A prefetching cache for the Internet tries to predict the Web documents that the user will request in the near future. It then preloads the predicted documents from the Web server and stores them in a cache. If the user requests a document that was prefetched before, the latency of sending a request to the server and waiting for the response is eliminated. For the user, the Internet connection seems to become faster. Web prefetching exploits the fact that Web documents can be loaded in the background, while the user is reading another page.

Care has to be taken to only prefetch those Web documents that can be stored in the cache. In the Internet, not all types of documents can be cached, since they contain cookies [25] or are dynamically generated by CGI [26] programs, for example. Prefetching such documents is a waste of network bandwidth, since the cache can never be used for these documents, they have to be always fetched from the server.

2.2.1. Prefetching Prediction

Prefetching prediction strategies decide which data items have to be pre-loaded from the server. Obviously, prefetching all possible data would be the best prefetching strategy. Basically, this strategy replicates the whole server in the client cache. However, a client usually needs only a portion of the data that a server provides. If this is the case, a prefetching prediction strategy has to be implemented that tells the cache what to prefetch.

The prediction strategy has to ensure accurate prefetching, which means that data that is not needed by the client later should not be prefetched and data that is needed by the client should be prefetched with a probability as high as possible. In the former case, too much data is prefetched and network resources as well as server resources are wasted and the cache is flooded with useless data. In the later case, a high prefetching accuracy ensures a high probability that the client can access needed data fast, which is beneficial for application performance.

Computer memory systems use prefetching to move data from the main memory to the CPU's cache memory before an application program requests the data. As processors have become faster, the system bus has become one of the main bottlenecks in modern PCs. Typical bus speeds are 66 MHz, 100 MHz, 133 MHz or 400 MHz on Intel Pentium based systems [30]. The speed of accessing the cache is orders of magnitude higher than the speed with which the main memory can be accessed. An early hardware prefetching work, which was reported by Smith [31], proposes a one-block lookahead scheme for prefetching. That is, when a memory access instruction brings block n into the cache, block n+1 is prefetched and loaded into the cache as well. This simple prefetching prediction strategy made use of the fact that memory accesses often occur in sequential order. More recent work combines hardware prefetching with software prefetching: Gornish et.al. propose in [51] that a language compiler is used to identify the earliest point in a program where a data block is accessed and can be prefetched.

The prefetching information is gathered at compile time and is not volatile. Therefore, a high prefetching accuracy can be achieved.

In file systems, predicting future file accesses enable the file system of the operating system to preload the files from the harddisk into the filesystem cache. Kroeger and Long describe a system that monitors file accesses and dynamically builds a tree-like data structure that describes file access interdependencies ([49], [53]). If the client requests a file, the system forecasts the probabilities of the files which may be accessed next. All files with a probability above a certain threshold value are then preloaded into the filesystem cache. The authors compared their predictive cache to an LRU cache and found a 15% to 22% improvement in cache hits of their approach.
The algorithm assumes that file access patterns are repetitive, which means that the client accesses the same sequence of files over and over. Thus, the authors argue that a data compression mechanism can be used to build a probability model of file interdependencies and to use this model for prediction.

The idea of using data compression techniques for prefetching was first advocated by Vitter and Krishnan in [32] and [33]. The intuition is that data compressors (like the Lempel-Ziv algorithm [34]) typically operate by postulating (either implicitly or explicitly) a dynamic probability distribution on the data to be compressed. Data expected with high probability are encoded with few bits, and unexpected data with many bits. Thus, if a data compressor successfully compresses the data, then its probability distribution on the data must be realistic and can be used for effective prediction.

In the Internet, each Web document is identified by a Uniform Resource Locator (URL). If a client requests a Web document from the server, it sends the URL of the desired Web document and waits for the Web document to be transferred. A survey of Web prefetching systems is given in [24]. Two commonly used approaches are request graphs and structural analysis, which we describe in the following.

Monitoring the URLs that a client requests enables a prefetching algorithm to derive URL interdependencies that may be used for predicting the URLs that a client will request in the future. This approach is used in [76] and [77], for example.

Another prediction approach that is used in the Internet is structural analysis of Web documents, as described in [46] for example. When a client receives a Web document, a background process parses the Web document and extracts the embedded links. All links are then prefetched in the background, while the user is reading the page. When the user clicks on a link, the probability is high that the requested URL is already in the cache. Structural analysis is also used in processor architectures, where the machine code of a program is searched for memory access instructions. The referenced memory pages can than be transferred to the CPU cache in advance, improving the overall execution speed of the program.

Prefetching prediction can be implemented either in the client or in the server. With client-side prediction, the prediction algorithm has to forecast only the future needs of its client. With server-side prediction, the server establishes information about what will be requested next and gives hints to clients. Each of the approaches has its advantages, we will discuss the advantages and disadvantage of each approach later, in the context of our work.

3. DOC CACHING AND PREFETCHING

In the previous chapter we described caching and prefetching in general and its applications in domains like network file systems and the Internet. Now we concentrate on caching and prefetching for Distributed Object Computing. After introducing the basic idea of our approach, Distributed Object Computing Caching and Prefetching (DOC-CaP), we present a sample deployment of the Address Book application and discuss how caching and prefetching can be applied to this sample application, thereby addressing cache consistency, cache replacement and prefetching prediction. Finally, we present alternative approaches and related work on caching and prefetching for Distributed Object Computing systems.

3.1. Introduction

In DOC-CaP, the result values of remote method calls are prefetched and stored in a client-side cache. When a client calls an already cached method, the result value is taken from the cache, thereby avoiding a remote method call. The cache system has to ensure that the cache items that are returned to clients are up-to-date. That means that a method result value taken from the cache has to be equal to the result value that a remote method call would return. If this is not the case, out-of-date cache items may be returned to the client.

In the following we describe the DOC-CaP approach with the Address Book sample application introduced in section 1.4.1.

Figure 45 illustrates the basic idea of the DOC-CaP approach with the AddressBook sample application: Whenever the *Client* calls a method, for example *getName()*, the *PersonClientStub* predicts the method calls that are likely to be called in the future. The *PersonClientStub* then creates a *MultiRequest*, which is a container for multiple *Request* objects. The *PersonClientStub* sends the *MultiRequest* to the server and waits for a *MultiResponse*.
The server side *RequestDispatcher* (see Figure 46), after receiving a *MultiRequest*, extracts and iterates over the contained *Requests* and forwards them to the object

implementations, one by one. After having executed all method implementations, the `RequestDispatcher` creates a `MultiResponse`, which is a container for multiple `Response` objects, and sends it back to the `ClientStub`.
After extracting all result values from the `MultiResponse`, the `PersonClientStub` stores the result values in a cache and returns the result value of the originally called method, `getName()` in Figure 45, to the `Client`. If the `Client` calls a method that is already in the cache, the `PersonClientStub` takes the result value from the cache, thereby avoiding a remote method call.

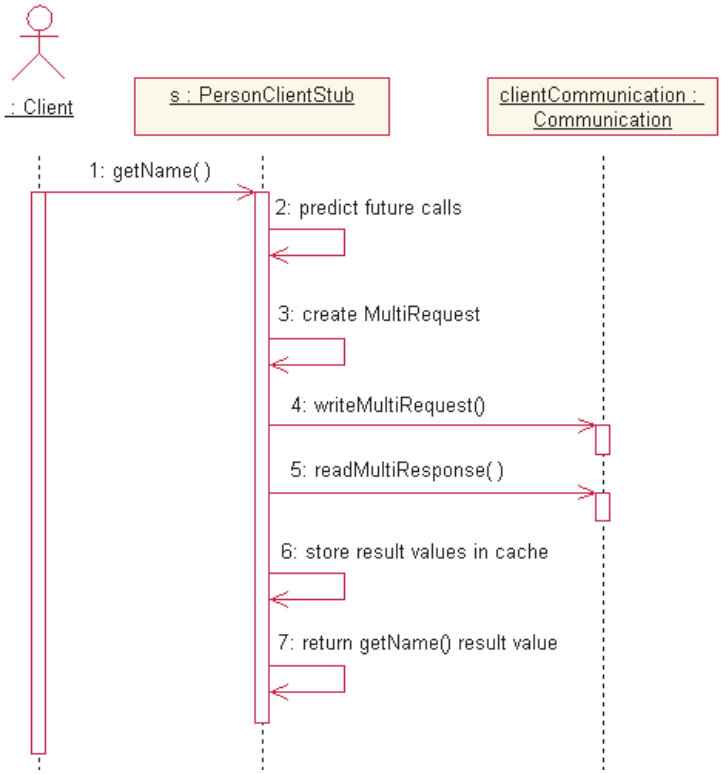


Figure 45: Prefetching And Caching Method Result Values (Client Side)

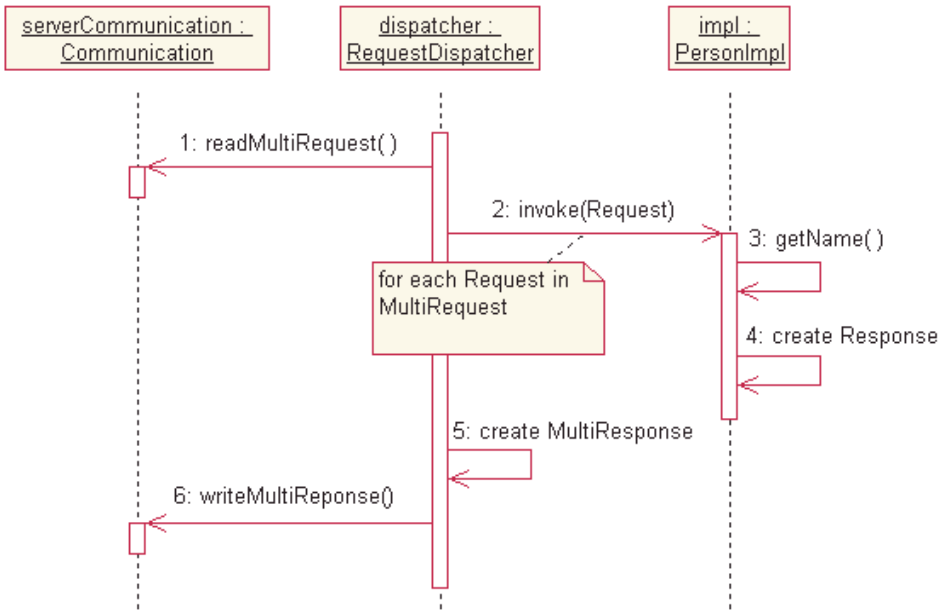


Figure 46: Prefetching And Caching Method Result Values (Server Side)

The DOC-CaP approach uses caching and prefetching for reducing the number of network roundtrips. Care has to be taken to ensure cache consistency and to achieve a high accuracy of prefetching prediction. These issues will be discussed in the following sections, based on ideas that have been presented in chapter 2.

3.2. DOC-CaP Caching

The deployment diagram in Figure 47 demonstrates this problem with the AddressBook sample application introduced in section 1.4.1. For the discussion in the following sections, we assume that the Address Book application uses a database for storing all person data. The AddressBook Server implements the AddressBook object model and provides its functionality via a Distributed Object Computing system. Finally, two AddressBook clients are connected to the AddressBook server.

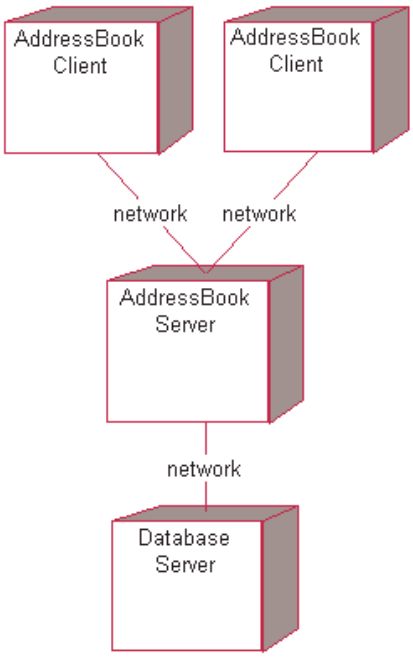


Figure 47: Address Book Sample Deployment Diagram

3.2.1. DOC-CaP Cache Consistency

In section 2.1.1 we introduced three types of cache inconsistency: local inconsistency, system inconsistency and global inconsistency. In this section, we address these types of inconsistency for DOC-CaP and present our cache consistency approach 'Expiration Model with Client Invalidation'.

```
interface Person
{
        string getName();
        void setName( in string newName );
};
```

Figure 48: Person IDL Definition

Local inconsistency in DOC-CaP can occur when a client calls and caches a method `a()` and then calls a method `b()` and the method implementation of `b()` modifies the state of the server in a way that another call of `a()` would yield a result value not equal to the cached value. As an example, let's assume we have a new method `setName()` in the `Person` interface, which sets the name of a `Person` (see Figure 48).

Figure 49 shows a series of method calls that leads to local inconsistency. When the `Client` calls `getName()` the first time, the `PersonStub` executes a remote method call and stores the `getName()` result value in the cache. The client then calls `setName()`, which results in a remote method call, updating the name of the `Person`. The client then calls `getName()` again, and one would expect that the new name should be returned. But instead, the `PersonStub` takes the method result from the cache, thus returning an out-of-date data item to the client, which is denoted by the lightning bolt in the sequence diagram. According to the default behavior of Distributed Object Computing systems (sending one `Request` per method call) the second invocation of `getName()` would have been forwarded to the server, returning the up-to-date name of the person[7].

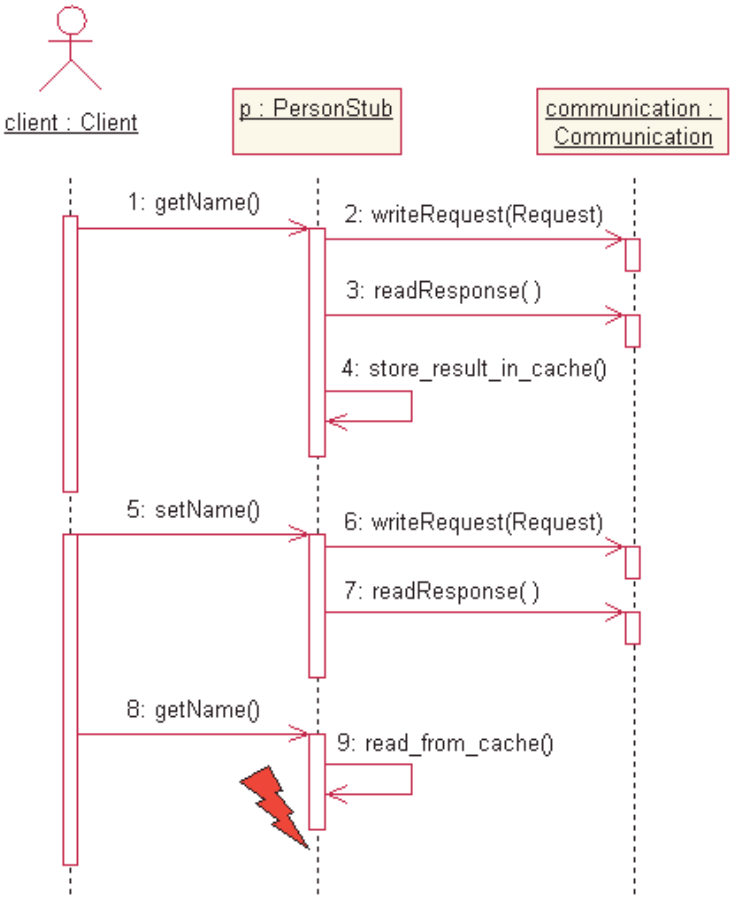


Figure 49: Sequence Diagram Of Local Inconsistency

[7] Without loss of generality we assume that method calls are served by server implementations in the same order as they are received.

System inconsistency in Distributed Object Computing systems occurs when two or more distributed clients execute method calls concurrently on the same server and calling one method affects the result value of the other method. Figure 50 shows a scenario where *client1* calls *getName()* in a *PersonStub p1* (for simplicity, we left out the *Communication* objects). The *PersonStub p1* sends a remote method *Request*, waits for the *Response* and stores the result value in the cache. Then, *client2* calls *setName()* on a *PersonStub p2* that represents the same server side *Person* implementation as *p1*. The *setName()* call modifies the name of the server side *Person* implementation. When *client1* calls *getName()* again, *p1* returns the cached value, which is out-of-date by now, indicated by the lightning bolt in the sequence diagram.

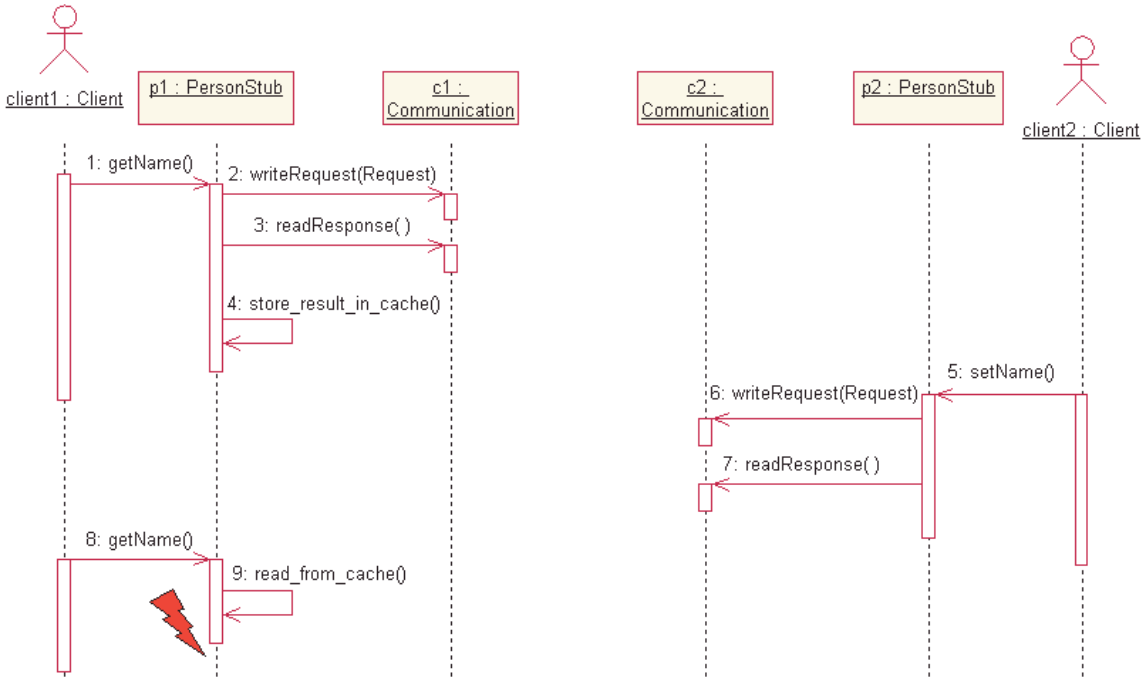


Figure 50: Sequence Diagram Of System Inconsistency

Global inconsistency in Distributed Object Computing systems occurs when some backend data source, which the server relies on, is modified. Figure 51 shows an example where a *Client* calls *getName()* and the result value is stored in the cache. Then, a database administrator updates the database that is used for data storage by the server. If the *Name* entry of the *Person* that is represented by the *PersonStub p1* is modified, the *Client* receives out-of-date data the next time it calls *getName()* again, which is denoted by the lightning bolt in the sequence diagram.

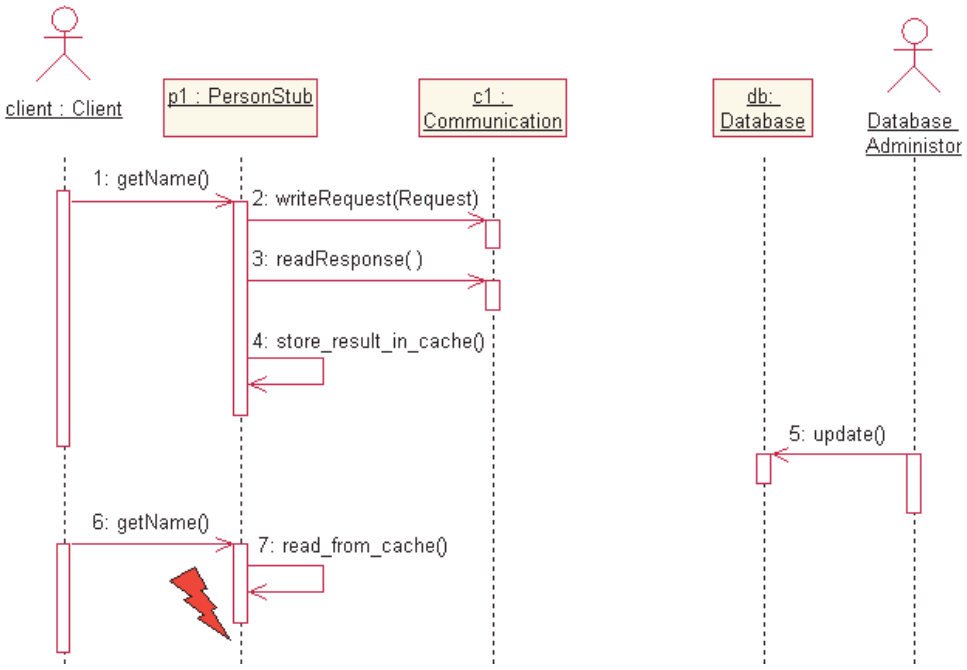


Figure 51: Sequence Diagram Of Global Inconsistency

In section 2.1.1 we introduced three approaches for ensuring cache consistency: Expiration Model, Client Validation and Server Invalidation. In the following we will discuss for each cache consistency approach its applicability in DOC-CaP. After discussing the approaches, we present our cache consistency mechanism, 'Expiration Model with Client Invalidation'.

Expiration Model in Distributed Object Computing systems
With this approach each method result value stored in the cache is tagged with a time-to-live value that expresses how long the cache item should be considered up-to-date. If a client calls a method, the cache system tests whether the requested method result value is stored in the cache and still fresh. If it is, then the cached result value is returned. Otherwise, a method `Request` is sent to the server and the method result value is retrieved from the server.
Figure 52 presents the case when a client calls a method with the result value already in cache and not yet expired. The cached result value can be safely returned to the calling client, thereby saving a remote method request. Figure 53 illustrates the case where the time-to-live value of a cache item has expired. A remote method request is created and sent to the server. After receiving the method response, the method result value is stored in the cache and then returned to the client.

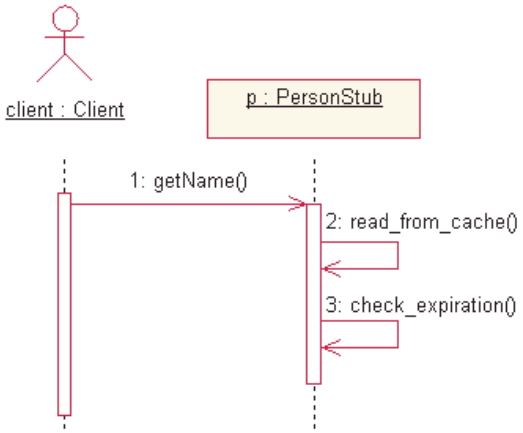


Figure 52: Expiration Model (Cache Value Not Expired)

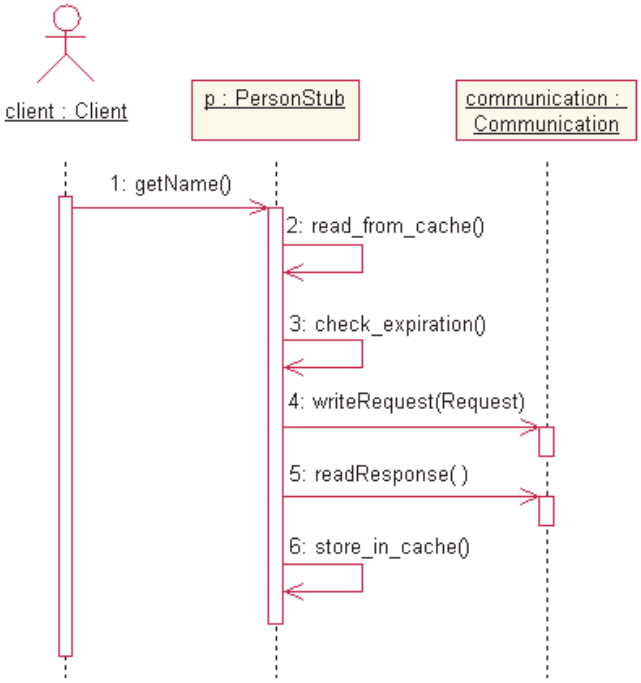


Figure 53: Expiration Model (Cache Value Expired)

The advantage of the expiration model is that a simple time comparison operation is sufficient to evaluate whether the cache item is up-to-date. Since validating a cache item is done by the client and no remote server interaction is necessary for checking the up-to-dateness of a cache item, the expiration model is suitable for applications where a high number of clients is connected to the server.

The drawback of the expiration model is that it does not guarantee cache consistency in all cases. If the server state is modified while the time-to-live time of a requested cache item has not yet elapsed, a client may receive an out-of-date method result value from the cache that does not represent the current server state.
When using an expiration based cache consistency protocol, finding the right time-to-live values for cache items is an important yet hard-to-solve issue. There is a tradeoff between cache consistency and performance. The higher the time-to-live value for cached method result values, the longer will cache items be stored in the cache and considered to be fresh, thus saving network roundtrips. On the other hand, short time-to-live values lead to an increase of the number of remote method invocations, because the cached data is more often considered out-of-date.
Time-to-live values can be set automatically (as, for example, in [58], where time-to-live values are dynamically adjusted based on the rate-of-change of a data item) or set manually by the programmer. Since we consider time-to-live values for method result values as highly application dependant in Distributed Object Computing applications, they cannot be set automatically. The rate of change of a method result can range from once per millisecond (think for example of a method returning the current time in milliseconds) to almost never (the `getName()` method result value of a specific `Person` will certainly change rarely, if at all).
Moreover, the time-to-live value for a method result is highly dependent on what constraints the application programmer sets on the cache consistency for a certain method result value. For example, for a person's name, the application developer may decide that it would be not harmful to risk a cache inconsistency of e.g. five minutes. In contrast, a fire department would expect that an incoming fire alarm, although a rarely occuring event, should be displayed promptly.

Client Validation in Distributed Object Computing systems
In this approach, the client side cache system validates the freshness of a cache item by checking the server for eventual modifications on the requested data item before taking the data item from the cache. For example, when a client (see Figure 54) calls `getName()`, the `PersonStub` could send a conditional `Request` to the server, where a conditional request contains information about the cached result value. If the cached result value is equal to the result value of the method implementation on the server, nothing has to be done and an empty `Response` can be returned to the `PersonStub`, thereby denoting that the cached result value can safely be returned to the client. If the method implementation returns a result value that is not equal to the cached result value, the new result value has to be added to the `Response` and stored in the client-side cache, thereby overwriting the old result value.

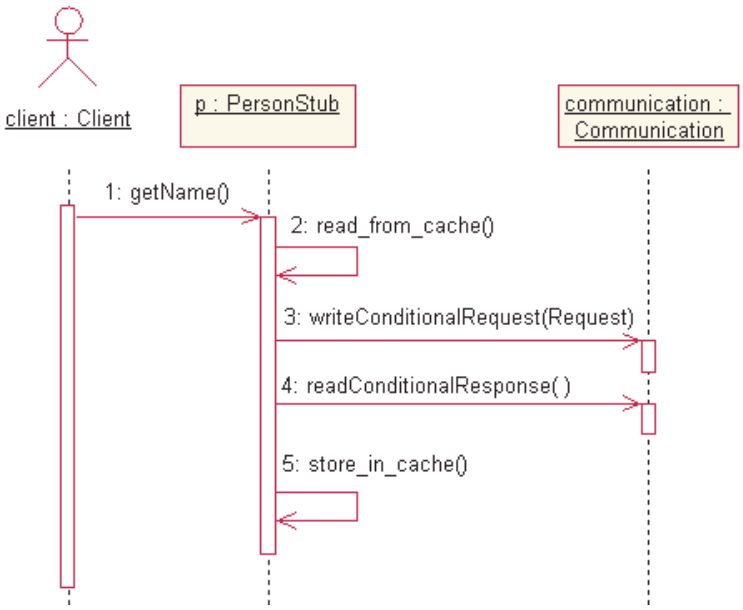


Figure 54: Client Validation

In Web caching, client validation is successfully used for reducing the network transfer volume, as described in section 2.1.1. However, client validation does not reduce the number of network roundtrips, as for each method invocation by the client, a complete network roundtrip is needed to send a conditional request to the server and receive the – possibly empty – response.

In Distributed Object Computing, one might think that the client invalidation approach assures that the client does never receive out-of-date method results, thereby ensuring strong cache consistency. However, there are cases where even with client validation out-of-date data is returned to the client. The scenario shown in Figure 55 illustrates how this can happen. Let's assume that two clients, *client1* and *client2*, are connected to the same server. Let's further assume that each client uses a *PersonStub* to communicate with a server side *Person* object implementation and that both *PersonStub* objects represent the same server side *Person* object implementation.

In the scenario shown in Figure 55, *client1* calls *getName()*, and the currently cached result value for *getName()* is "John", which is also the current name of the server side Person object. Now *p1* sends a conditional request to the server and expects the server to answer with a response containing the new person name or an empty response. In the meantime, *client2* calls *setName()*, with intent to set the person name to "Jake".

The question is now whether *p1* will receive the new person name, which is "Jake" or whether it will receive an empty *Response*, which indicates that "John" is still the current name. The answer is that it depends on which *Request*, the one sent by *p1* or the one sent by *p2*, reaches the server first and is executed. If the p1 *Request* the server before the *p2 Request*, the server side *Person* name is still "John" and the server will send an empty

Response back to *p1*. Otherwise, if the *p2 Request* reaches the server first, the person name is updated before the *p1 Request* reaches the server and the new person name is sent back to *p1*.

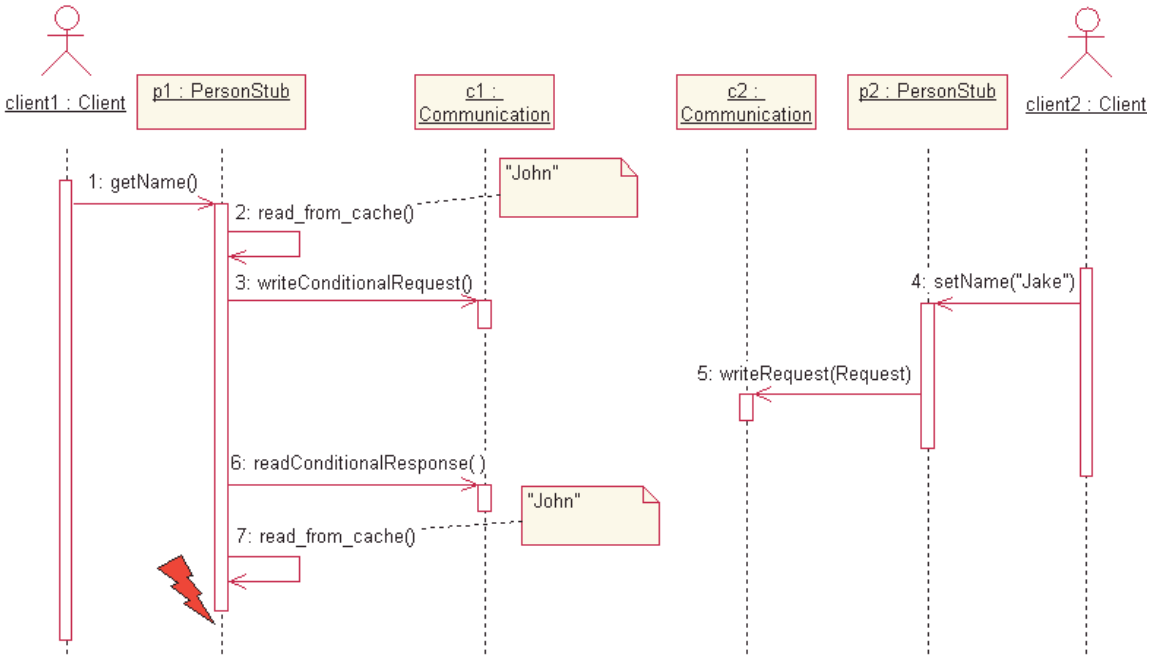


Figure 55: Race Conditions With Client Validation

In distributed systems, the time it takes for a network packet to travel from one point to another is not deterministic and generally cannot be foreseen. Particularly, it is not guaranteed that network packets sent by a number of clients reach the server in the order in which they are sent. As Lamport pointed out in his classic paper [23], there is generally no explicit clock which could be used to synchronize events in distributed systems. Therefore, if a distributed application relies on the ordering of events, which may be the ordering of method calls in Distributed Object Computing applications, a synchronization protocol has to be implemented. If no synchronization protocol is used, as is the case in standard Distributed Object Computing systems, client invalidation does not guarantee strong cache consistency.

Server Invalidation in Distributed Object Computing systems

With server invalidation, the server notifies all clients to inform them about updates. Each notified client then invalidates its cache. Figure 56 shows an example of server invalidation. *Client1* queries the name of a *Person* by calling *getName()* in a *PersonStub*. If the result value is in the cache, no remote method call is sent to the server. Then, another *client2* calls *setName()*, modifying the name attribute of the *Person* implementation in the server. Upon this modification, the server notifies all connected *PersonStubs* that it has

been modified and caches must be cleared. (Here, the notification is received by a `Communication` object and forwarded to the `PersonStub`.)

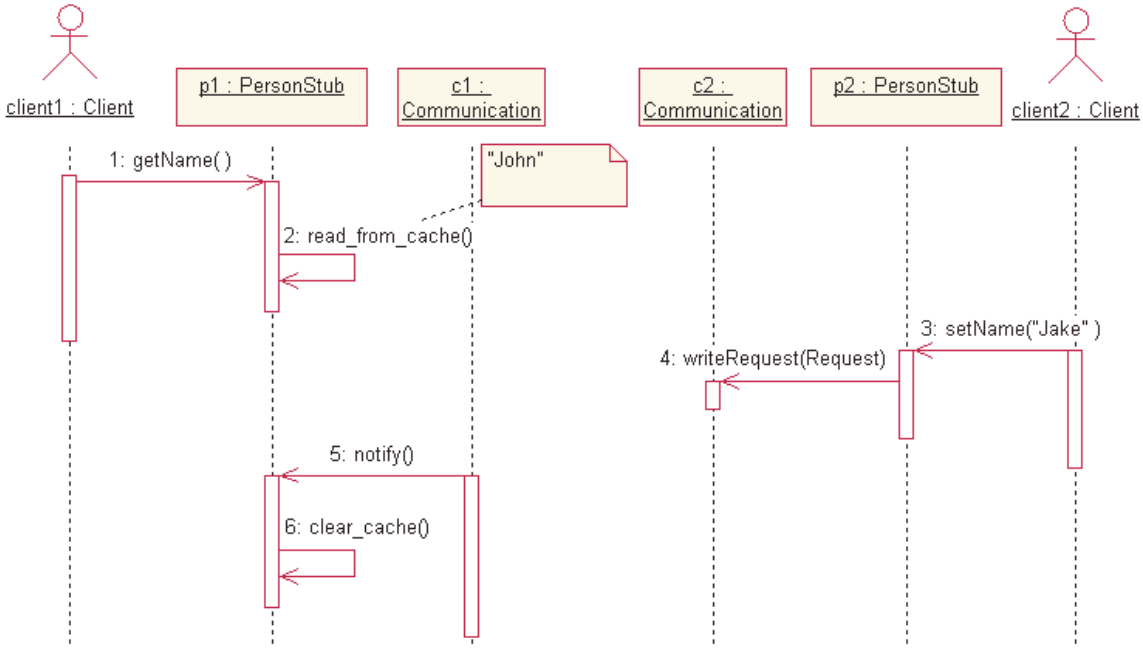


Figure 56: Server Invalidation

How the notification is done in detail is a matter of implementation. For example, the server could just tell the clients that ‘something’ has changed, in which case each client would have to completely invalidate its cache. Or, the server could specify more exactly what has changed, thereby giving the clients a chance to invalidate just a small part of their caches. However, in each case a network roundtrip from the server to all clients must be initiated. The advantage of this approach is that network communication is only necessary in the case of server state updates.

The drawback of this approach is that upon each server update, a callback from the server to the client is needed. In practice, there are approaches like piggypacking [82] (the invalidation notification is appended to a normal method response) and delayed notification (multiple notifications are sent to a client in one single network roundtrip) that help to reduce the number of remote client notifications. But even with these approaches, each method call that updates the server state triggers a burst of callbacks from the server to a number of clients. The higher the number of clients, the more network roundtrips will be necessary to notify all clients.

Moreover, server invalidation does not guarantee that all client caches stay up-to-date. If a client calls a method that updates the server and shortly thereafter, a second client calls a method that is read from the cache before the notification of the first client’s call is received, out-of-date data may be returned to the second client.

The server invalidation approach works best if the server has complete control over its data. In file systems, for example, the operating system has complete control over file read and write actions. A file cannot be modified without knowledge of the operating system. This is not the case in Distributed Object Computing, where it is generally not possible for a server application to know when its state has been updated. For example, if the server application stores some of its data in a remote database and a third instance (e.g. a database administrator) updates the database, the server state would have been modified. But because the database was modified without knowledge of the server application, no client is notified.

Leases in Distributed Object Computing systems
With leases, a server notifies all its clients about server state modifications for a certain time (lease-time). If the lease time has elapsed for a client, it can request a new lease from the server or switch to alternative cache consistency approaches like expiration model or client validation. The granularity of a lease can be chosen by a client: The client can request to be notified upon any server modifications, or it can chose to be notified upon modifications of a specific subset of the server data. Thus, the lease approach helps to adapt the network traffic of the server invalidation approach to the client needs. However, the same issues as in the server invalidation approach apply to the lease approach, for example the problem of maintaining cache consistency in all cases.

DOC-CaP Consistency Approach: Expiration Model With Client Invalidation
The cache consistency approach in our Distributed Object Computing Framework is based on an expiration model extend by client invalidation.
Whenever a client calls a method in a `ClientStub`, the cache checks whether the result value of the requested method is in the cache and is not yet expired. If both conditions are true, the cache item is returned to the client. Otherwise, a remote method request is sent to the server, the method result value is stored in the cache and returned to the `ClientStub`. Additionally, the expiration time for the new cache item is set. If the client calls the same method again within the expiration time, its result value is considered fresh and returned to the client without sending a remote method call to the server, thus avoiding one network roundtrip.
With the expiration model, the risk of returning out-of-date data is high if time-to-live values for cache items are high or the server state changes often. Therefore, we combine the expiration approach with client invalidation: Whenever a client calls a method that might change the server state, the client invalidates its own cache. No update notifications are sent to other clients.

The drawback of using an expiration model approach is that even with client invalidation, out-of-date cache items might be returned to a client. In the following we describe how this can happen, and we discuss the likelihood of out-of-date cache items.

For the discussion, we use a modified version of the AddressBook sample application introduced in section 1.4.2. The IDL definition is shown in Figure 57.

```
1  interface Person
2  {
3          string getName();
4          void setName( in string newName );
5  };
6  typedef sequence<Person> PersonList;
7
8  interface AddressBook
9  {
10         PersonList search( in string x );
11 };
```

Figure 57: Address Book IDL Definition

The *AddressBook* interface is used to get a list of *Person* references. Each *Person* has a name, which can be retrieved by *getName()* and set by calling *setName()* with the new name as parameter. Let's further assume that the Address Book system consists of one *AddressBook* server that is connected to a database management system for storing the *Person* data, and a number of *AddressBook* clients that are connected to the *AddressBook* server via a network (see also the deployment diagram in Figure 47).

For the discussion of the DOC-CaP cache consistency approach, we make the following assumptions about distributed applications using DOC-CaP:

1. The distributed application consists of one or more clients and one server[8]. The clients communicate with the server via remote method calls.
2. Remote method calls are synchronous. A client calling a stub method is blocked until the stub method returns.
3. Clients are single-threaded[9]: If a client calls two methods, one after another, it is guaranteed a *Request* for the second method is not sent before the *Response* for the first method has been received by the client. Consequently, it is guaranteed that the method implementations are called in the same sequential order as called by one client (i.e. the implementation of the second method does not start executing before the implementation of the first method has finished executing). Note that this assumption is not made for two methods that are called by two different clients.
4. A method sent to the server can be of two kinds: If a method does not - under any cicumstances - modify the state of the server, we call this method a 'get' method. If a method might change the server state, we call this method a 'set' method.

[8] A server may be single-threaded or multi-threaded. A single-threaded server can execute at most one remote method implementation at a time. A multi-threaded server can execute any number of remote method implementations concurrently.
[9] The programmer is still free to implement multi-threaded clients, where a client can invoke methods concurrently in different threads. In this case, we view each thread as a single-threaded client on its own.

Discussion Of Local Inconsistency
Local inconsistency occurs if a client modifies the state of a server in such a way that the cache of that client becomes outdated. In the following we demonstrate that local inconsistency cannot occur with the DOC-CaP cache consistency approach.

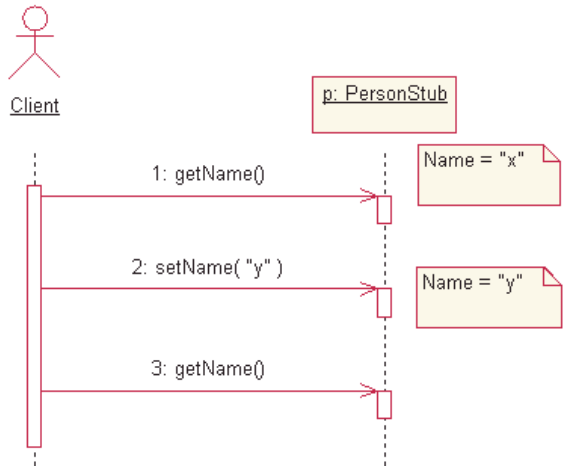


Figure 58: Local Inconsistency

Figure 58 presents an example of client invalidation. Let's assume that the name of the `Person` the `PersonStub` refers to is "x". Without caching, the first call to `getName()` returns "x". The `setName()` call updates the server state by setting the `Person` name to "y". Since it is guaranteed that the second `getName()` method is executed after the `setName()`, it is ensured that the second call to `getName()` returns "y".
With caching, the first call to `getName()` is forwarded to the server. The result value "x" is stored in the cache and returned to the client. Upon `setName()`, the client clears its cache, since `setName()` modifies the server by setting a new `Person` name. Note that with client invalidation, only this client's cache is cleared. At the time the second `getName()` is called by the client, the `getName()` call is forwarded to the server, since there is no cache item for `getName()`. The result value "y" is then stored in the cache and returned to the client.

For the general discussion of local inconsistency we make some further assumptions about distributed applications using DOC-CaP:

1. The distributed application consists of one single-threaded client and a server. The client communicates with the server via remote method calls. Note that this assumption includes a client communicating with two or more different servers, as each server can be treated as one or more separate threads within one multi-threaded server.
2. The state of a server does not change between the execution of two successive remote method calls. In particular, no entity can change the state of the server but the client

that is connected to that server, via the invocation of a ‘set’ method. (For example, there exists no server-side database that is read and written by the server and that can be modified by a database administrator.)

While the client is executing, it sends remote method calls to the server, one after another. At any time, the client can call either a ‘get’ or a ‘set’ method.

If the client calls a ‘set’ method, then the client-side cache is cleared and a remote method request is sent to the server. The result value is returned to the client, without being stored in the client-side cache. Therefore it is assured that, after a ‘set’ method has been called, the method result value reflects the server state and the cache is empty and does not contain outdated data.

If the client calls a ‘get’ method, then we have to consider two cases: If the result value of the called method is already in the cache and has not expired, it is returned to the client immediately, without contacting the server. It is sure that the cache value is still fresh, since no modification of a server state could have happened since the last ‘set’ method of this client (see assumptions).
If the result value of the called method is not in the cache or has expired, a remote method request is sent to the server and the method implementation is executed. The fresh result value is then stored in the client-side cache and returned to the client.

We conclude that local inconsistency (a client outdating its own cache) cannot occur in DOC-CaP if the assumptions stated above are met.

Discussion Of System Inconsistency
Client invalidation does not guarantee cache consistency in the case of system inconsistency, because only the client that initiated the update on the server state is invalidated. The other clients across the network remain untouched and possibly out-of-date.
As an example, Figure 59 shows two clients, `client1` and `client2`. We assume that the two clients are running on two different computers. Moreover, we assume that the two person stubs `p1` and `p2` represent the same server side `Person` implementation. First, `client1` calls `getName()`, the result value is “x”. Then, `client2` calls `setName()` and sets the `Person` name to “y”. Then, `client1` calls `getName()` the second time. According to Figure 59, the result value of the second `getName()` invocation should be “y”, because the `Person` name has been set to “y” by `client2`.

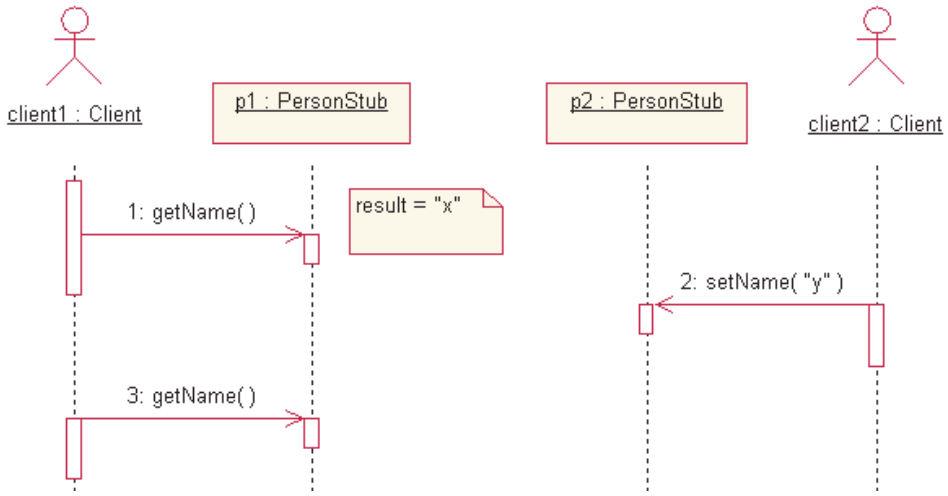


Figure 59: System Inconsistency

With the cache consistency approach used in DOC-CaP, there are two possible result values for the second invocation of `getName()`. If the expiration time of the first `getName()` result has not elapsed by the time `getName()` is called the second time, the cached result value "x" is returned. If it has elapsed in the meantime, a method request is sent to the server, and the result value is "x".

For the general discussion of system inconsistency we make some further assumptions about distributed applications:

1. The distributed application consists of two or more single-threaded clients communicating with a server.
2. The network connection between clients and servers is unreliable insofar as there is is no guarantee about packet delivery times. In particular, the network does not guarantee that a network packet that was sent before a second network packet reaches its destination before the second packet reaches its destination. (An example of such a network is TCP/IPv4 over Ethernet, where network delays and packet losses contribute to a delivery time that cannot be assessed in advance.)
3. The state of the server does not change in between the execution of two successive remote method calls. In particular, no entity can change the state of the server but the clients that are connected to that server, via the invocation of 'set' methods.
4. The clients are not synchronized, they run independently from one another. In particular, there is not client-to-client communication.

While the clients are executing, they send remote method calls to the server. At any time, a client can call either a 'get' or a 'set' method.

As long as all clients are calling only ‘get’ methods, the server state is not modified and system inconsistency cannot occur. If one client calls a ‘set’ method, the state of the server may change and system inconsistency may occur. The likelihood with which system inconsistency may occur depends, among other factors, on the expiration time of method result values that are stored in the cache of the clients.

It is up to the application programmer to trade off between the likelihood of system inconsistency and application performance: The programmer may decide that system inconsistency can be tolerated for a distributed application. Then caching does not need to be disabled for this application. Or, the programmer may decide that system inconsistency cannot be tolerated for a given application.

If system inconsistency can be tolerated for a distributed application, the programmer can fine-tune the likelihood of system inconsistency by setting the expiration time of cached remote method result values. If the expiration time for a cached result value is high compared to the rate-of-change of the server state reflecting this value, then the likelihood of system inconsistency may increase.

If system inconsistency cannot be tolerated for a distributed application, the programmer may chose to disable caching for certain parts of the application. An example of such an application would be a stock exchange information system, where one client writes current stock price values to a server and another client reads prices from the server and displays it to a user. A programmer may decide that caching of these values may be turned off to ensure that stock price values are displayed as fast and accurate as possible. However, even when caching is turned off, there is still the network roundtrip time that delays the propagation of price values from the server to the user, and that roundtrip time cannot be guaranteed in advance. If an application depends on guaranteed times, QoS-enabled networks and real-time operating systems must be used.

One may argue that the problem of system inconsistency can be avoided by disabling caching for remote method result values. But even when caching is disabled, the result values of remote methods called by a client might be the same as with caching enabled. This happens when, due to high a network delay, a ‘set’ method called by a second client does not reach the server until the ‘get’ method of the first client has finished executing. This example can easily be applied to the general case, where a number of unsynchronized clients communicate a server calling ‘set’ and ‘get’ methods interchangeably.

If a distributed application is using a Distributed Object Computing system as defined in section 1.2 and is deployed on an unreliable network (according to the above assumption), it is not guaranteed that if a client calls a method `m1` before another client calls a method `m2`, the two methods reach the server in the same order or are executed in the same order as they have been called by the clients. If the server is multi-threaded, then the order of execution of remote method requests is up to the server, even if the method requests reach the server in the

'correct' order, thereby introducing another level of uncertainty. If a distributed application has to ensure the 'correct' execution order of methods, it must use some sort of synchronization between clients and/or servers, for example by using locking/unlocking strategies (see [85] for an exmple). In practice, there exist a number of synchronization mechanisms for distributed applications, e.g. the CORBA Concurrency Control Service [86].

Discussion Of Global Inconsistency

As an example of global inconsistency, Figure 60 shows a client, `client1`, and a Database Administrator who has access to the AddressBook database. First, `client1` calls `getName()`, the result value is "x". Then, the Database Administrator updates the person's name entry to "y" directly in the database. Without caching, if the client now calls `getName()` again, the method call is forwarded to the server. The server might query the database now in order to get the person's name. In that case, the `getName()` result value would be "y". If the server holds the person's name in memory and does not query the database, the `getName()` result value would be "x".

With caching enabled, the `getName()` result depends on whether the cache item "x" that originated from the first invocation of `getName()` is already expired or not. If it is still fresh, "x" is returned to the client from the cache. If it is expired, the method call is forwarded to the server, which can return "x" or "y", depending on whether it queries the database or not.

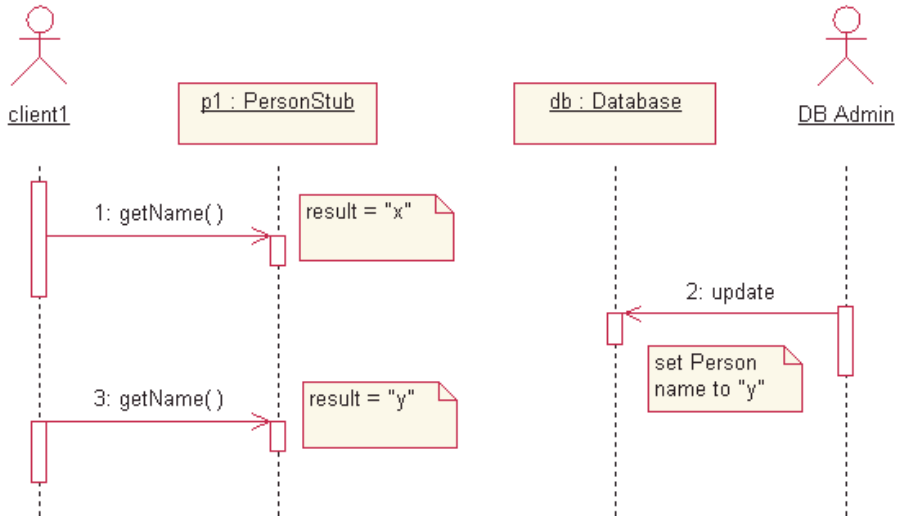


Figure 60: Global Inconsistency

For the general discussion of global inconsistency we make some further assumptions about distributed applications:

1. The distributed application consists of one or more single-threaded clients communicating with a server.

2. The state of the server may change at any time. In particular, the server state may change between the execution of two successive remote method calls.

As long as all clients are calling only ‘set’ methods, no method result values are stored in the cache and each ‘set’ method call is forwarded to the server. Global inconsistency cannot occur in this case. If a client calls a ‘get’ method, the result value of this method may or may not be in the cache. If the result value is not in the cache (or has expired), the ‘get’ method is forwarded to the server. If the result value is in the cache and not expired, the cache value is returned to the client and global inconsistency may occur.

The likelihood with which global inconsistency may occur depends, among other factors, on the expiration time of method result values that are stored in the cache of the clients, and on the update rate of the server state.

Like in the case of system inconsistency, it is up to the application programmer to trade off between the likelihood of global inconsistency and application performance. It is up to the programmer to control the likelihood of global inconsistency by adjusting the cache expiration times for remote method result values.

We conclude that global inconsistency (a modification of the server state outdating client caches) may occur in DOC-CaP. The likelihood of global inconsistency can be controlled by the application programmer.

Conclusion
The DOC-CaP cache consistency approach ensures that local inconsistency cannot occur with DOC-CaP. System inconsistency and global inconsistency may occur. The application programmer can controll the likelihood of system inconsistency and global inconsistency by adjusting the expiration time for remote method result values and by disabling caching for certain parts of a distributed application. We will describe in section 4 how expiration times can be adjusted and caching can be enabled or disabled in DOC-CaP.

3.2.2. DOC-CaP Cache Replacement

When a method is invoked by a client or prefetched by the DOC-CaP prefetching system, the method result value is stored in a client side cache. In DOC-CaP, the cache stores method result values in main memory. Since memory space is limited, the issue of cache space has to be addressed, for example by implementing a cache replacements strategy like described in section 2.1.2. The replacement strategy chosen in DOC-CaP is to tie the life cycle of cache items to the life cycle of `ClientStub` objects, which is described in the following.

Each `ClientStub` object has a lifecycle that is composed of three stages. First a `ClientStub` is created. For a client, calling a method that returns object references is the only (implicit) way of creating a `ClientStub` object. Particularly, `ClientStub` objects cannot be created by the client explicitly.
The client uses the `ClientStub` object as a representative of a server side object implementation. The client invokes methods in the `ClientStub` and these method calls are then forwarded to the server side object implementation.
After the client does not need to communicate with the server side object implementation any longer, it releases the respective `ClientStub` object. The implementation of releasing a stub object can vary between different Distributed Object Computing systems. In the case of CORBA, for example, the client programmer must call the method `_release()` provided by every `ClientStub` object. If the client is programmed in Java, the Java Garbage Collector takes care of releasing unused `ClientStub` objects that are no longer referenced.

When a `ClientStub` object is released and deleted, the cache contents of this stub object are also deleted. Thus, the lifecycle of a cache item is the same as the lifecycle of the `ClientStub` object that the cache item is assigned to.

Theoretically, the client cache may run out of memory, for example if a prefetching action yields method result values that are larger than the client side cache memory. However, we consider these cases very seldom and argue that methods that may return vary large result values should be defined as non-cachable by the programmer. (We will describe in section 4.2 how methods can be defined as non-cachable.) For the integration of a cache replacement strategy as described in section 2.1.2, we refer to future work.

3.3. DOC-CaP Prefetching

Prefetching is used to reduce network roundtrips for methods that are called for the first time. In this section we describe which methods can be prefetched and outline characteristics of the DOC-CaP prefetching prediction strategy.

3.3.1. Prefetchable Methods

In DOC-CaP, not every method can be prefetched. Whether a method is prefetchable depends on its side effects. A method is said to have side effects if calling the method might change the server state. A method is called 'without side effects' if it is sure that calling the method does not change the server state under any circumstances.

Methods without side effects are also called read-methods or get-methods. If such a method is prefetched and later called by the client, the result can be taken from the cache. If it is prefetched and never called by the client later on, network resources as well as server CPU time is wasted.

There is, however, a problem with prefetching methods that do have side effects. If such a method is prefetched, the method implementation is executed and the server state might be modified. If the client does not call this method later on, the server state is modified without the client wanting to do so. This can lead to serious problems. Consider the Address Book sample application presented in section 1.4.1. Let's introduce a new method `removeThis()` in the `Person` interface that should delete a person entry from the address book, see Figure 61.

```
interface Person
{
        [...]
        booolean removeThis();
};
```

Figure 61: A Method With Side Effects

If `removeThis()` is executed on a `Person` object, the `Person` entry is removed from its `AddressBook`. If the server keeps its person data in a database, a table row in a database is deleted. If this method is prefetched, without the client application ever invoking `removeThis()`, the person entry would have been deleted without the client wanting to do so. Consequently, we must not prefetch any methods that may possibly modify the server

state, which leads to the prefetching restriction that methods with side effects are not prefetchable.

Server state denotes not only the state of the server side object implementation that is invoked by a method but also any resources (referenced object instances, database entries, data files, memory contents) used by the method implementation.

Determining whether a method has side effects is not a trivial task. Theoretically, one can analyze the source code of a system and tell which method implementations are merely read-implementations, i.e. have no side effects. However, to the best of our knowledge, there are no practical tools that can automatically derive side effect information from application source code and externalize that information, although we consider it feasible under certain circumstances[10].
Other projects that have to have knowledge about method side effects rely on information that has to provided by the programmer. For example, the ICE Middleware [48] introduces the 'nonmutating' operation qualifier to denote whether a method implementation is 'const', which means that it does not change the state of its object. This information is then used for the implementation of error recovery mechanisms. Another example is the C++ programming language [28], where the 'const' keyword, applied to a method declaration, denotes that the method implementation does not modify the state of its object. A C++ language compiler can ensure that a 'const' method does indeed not modify the state of its object, but a C++ compiler cannot automatically detect whether a method is 'const'.

Prefetching Methods With In-Parameters

In-parameters are used in Distributed Object Computing systems to pass information from the caller of a method to the method implementation. When calling a method with in-parameters, the parameter values are marshalled and added to the method request. When the server receives the method request, it unmarshalls the parameter values and passes them to the method implementation. Since in-parameters cannot be used to pass information back to the caller, they are not included in the method response sent back to the client.

When the DOC-CaP prefetching system predicts a certain method to be prefetched and the predicted method has in-parameters, the prefetching system must also predict the value of these call parameters before prefetching the method. Let's assume we add a method `getAttribute()` to the `Person` interface, as in Figure 62.

[10] For very easy method implementations, it is conceivable that an automatic tool can discover whether the method implementation is free of side effects. But this is generally not possible for all kinds of methods and all kinds of programming languages.

```
interface Person
{
        [...]
        string getAttribute( in string attributeName);
};
```

Figure 62: Ingoing Parameters

The method *getAttribute()* returns the value of the person attribute specified by the in-parameter *attributeName*, which can be “name”, “email”, “phone” and so on. If the client issues a call such as *getAttribute(“name”)*, the name of the remote person object is returned.

Whether a method with in-parameters is prefetchable depends on the prefetching prediction algorithm that is used. If the prediction algorithm is able to predict, along with the name of the methods that will be called in the future, the values of in-parameters, then methods with in-parameters are prefetchable. If the prediction algorithm is not able to predict in-parameter values, then these methods will not be prefetchable.

Prefetching Methods With Out-Parameters

Out-parameters are used to pass information from the method implementation back to the caller of the method. Out-parameters are not marshalled and added to the method request. After the method implementation has finished its execution and the value of the out-parameters have been set, they are marshalled into the method response and sent back to the client. The client can now retrieve the value of each out-parameter.

In Distributed Object Computing systems, out-parameters are often used if a method has more than one result value. This technique is also known in C++ programming, where pointers to variables are used to overcome the ‘one-result-value-per-method’ constraint. For example, the *fread()* file read function defined by ANSI C has to return the number of bytes actually read from the file, the bytes, or an error code in case the file read did not succeed. The *fread()* function returns the number of bytes that were read from the file, and the byte content itself is stored in a byte array that is passed to *fread()* as a pointer to a byte array.

When prefetching a method in DOC-CaP, the result values of all prefetched methods are stored in the cache. This is true for the normal method result value as well as all out-parameters of the prefetched method. From a logical view, out-parameters are the same as method result values.

Prefetching Methods With In/Out-Parameters

In/out-parameters are a combination of in- and out-parameters and are used to pass information from the caller of a method to the method implementation and back. In/out-parameters are marshalled both into the method request and, after the method implementation has finished and eventually modified the value of the in/out-parameter, the parameter value is marshalled into the method response.

It depends on the prefetching prediction algorithm whether methods with in/out-parameters are prefetchable. If the prediction algorithm is able to predict in -parameter values, then methods with in/out-parameters are prefetchable as well. Otherwise, methods with in/out-parameters will not be prefetchable.

3.3.2. Prefetching Prediction in DOC-CaP

In this section we describe the characteristics of prefetching prediction in DOC-CaP and show the differences to prefetching in file systems and memory systems.

In DOC-CaP, prefetching is initiated upon a method call, as shown in Figure 45 and Figure 46 in section 3.1, and the prefetching takes place synchronously. The prefetching starts when a method is called by the client and ends when the method response is received by the `ClientStub`.

One design goal of the DOC-CaP approach is transparency for client and server. This excludes the possibility of using an ‘informed prefetching’ approach like described in [01] and [02], where the client application issues hints about its future needs to the prefetching system. The prefetching system has to be self-learning and must be adaptable to changing client behaviors. Moreover, no structural information about method interdependencies are available, like in Web prefetching, where analyzing the hyperlinks in a Web page yields valuable information about prefetching candidates. A theoretical source of structural information would be the client code (source code or object code). Yeung and Kelly show in their work [29] that byte code analysis can be used to gain knowledge of a client’s behavior. Their system is based on Java and RMI [72]. However, since DOC-CaP is language neutral, byte code analysis techniques cannot be applied.

A characteristic of Distributed Object Computing applications is that they develop repetitive patterns of method invocations and client-server communication. Monitoring these method calls and the sequence in which they occur yields information about the client behavior. In [32], Vitter and Krishnan argue that data compression techniques can be used to analyze repetitive patterns of data access events and propose a technique for using the Lempel-Ziv compression algorithm [34] for prefetching prediction. In [33], Curewitz et. al. describe the practical issues of applying the technique.

The choice of a prefetching approach depends not only on design goals like transparency for client and server, but also on the location where prefetching prediction is implemented: Prefetching prediction can be performed either by the client (the consumer of data) or by the server (the data source).

Client Prediction Approach
In this case the prefetching prediction is implemented at the client side. The prediction algorithm collects information about past invocations of its client and incrementally builds up an invocation pattern model. Based on this pattern model, it predicts the methods that will be invoked by the client in the future. One advantage of the client prediction approach is that the invocation pattern model is built on a per-client basis. It automatically adjusts to the invocation patterns of a client. Since invocation patterns can vary between different clients, client prediction helps to increase the accuracy of the prediction results. The client invocation pattern can vary if different users are using different kinds of client software. Think for example of our address book application introduced in section 1.4.1, where one type of client software can be used to search names and read phone numbers, and another type of client software, the administrator client software, can be used to insert, update and delete person entries. The first type of client software will invoke different methods than the second type. If different types of users use two instances of the same client software, patterns of method invocations can also differ to a high degree. Since each user has its own preferences and ways to use a type of client software, maybe because the users differ in experience levels, the client will invoke different methods in a different order for each user.

Server Prediction Approach
When using this approach, the prefetching prediction subsystem is implemented at the server side. The server intercepts incoming method invocation requests, collects this information and builds up an invocation pattern model, which in turn is used for predicting future invocation requests. The advantage of this approach is that new clients do not need to build up invocation models. Instead, the server stores these models and predicts the future requests for a client. The prediction algorithm is running in one single place, not spread over all clients. This helps to reduce overall distributed application resource consumption like CPU cycles and memory space.
The drawback is that the invocation model used by the server side prediction algorithm does not distinguish between different types of clients. If the server, upon an incoming invocation request, does not know the originating client, the invocation model cannot be built on a per-client basis. For example, this is the case with CORBA, where the server does not know which client issued a remote method invocation.

DOC-CaP uses the client prediction approach, because it aims at supporting multiple client types with different invocation patterns. Moreover, each client can decide on its own whether it wants to use caching and prefetching.

3.4. Related Work

The problem of fine-grained object interfaces is the subject of many ongoing research projects. In the following, we present design patterns and research projects that also aim at using caching and prefetching for reducing the number of network roundtrips. While discussing each solution, we point out the differences to the DOC-CaP approach.

Fine-Grained Framework

The Fine-Grained Framework design pattern proposed by Mowbray [52] describes a mechanism for reducing the number of remote method calls for fine-grained object interfaces while enforcing client transparancy. Mowbray's idea is to organize object instances into working sets where objects likely to be used together are in the same working set. Next, a cache is defined and implemented by the application programmer that for a single retrieval operation retrieves the complete working set and for a single update operation updates the complete working set. The proposed cache implementation is at the stub layer to enforce client transparancy, that means that the client accesses a fine-grained interface and is not aware of caching that occurs in the stub layer.

When an object is invoked, the client stub checks if the object already exists in the object cache. If not, the stub retrieves the working set this object belongs to. Once the object is available in the cache, the client stub can perform the requested operation on the cached object rather than performing a remote invocation. The Fine-Grained Framework does not propose a solution for cache invalidation and update propagation, thus the problem of maintaining cache consistency is left to the application programmer who implements the cache. Mowbray proposes that cached object values should periodically be propagated to the server.

By proposing object working sets and having a complete working set being retrieved upon the first invocation of an object, a prefetching mechanism is implemented. However, the problem of side effects is not addressed by Mowbray's proposal.

Manually defining working sets and thus defining the granularity of object access is beneficial to application performance in many cases, as the programmer of a distributed application knows best which objects will be invoked together. Implementing a cache consistency strategy by hand is beneficial as well, since the programmer is free to decide where to enforce strong cache consistency, using a transaction-based mechanism, and where weak consistency may be sufficient.

However, the drawback of this approach is that its implementation effort is exceedingly high and the possibility of reusing the cache implementation of an application for other applications is very limited, as the framework is specifically tailored to object

implementations and applications. Moreover, the granularity of the prefetching strategy is defined at design time, thus not supporting run-time adjustment of prefetching granularities.

While the Fine-Grained Framework design pattern is merely a proposal for the restructuring of object communication and does not provide an implementation, there are research projects that aim at providing a solution that can be transferred to an application-independent and reusable implementation.

Client-side Component Caching

Pohl and Schill describe in [58] a system for caching attributes values of Enterprise JavaBeans [55]. An Enterprise JavaBean (EJB) can be seen as an object that implements an interface that is composed of methods and attributes, much like an object in Distributed Object Computing systems[11]. Attributes can be read and written, while for each attribute a get/set method pair is generated by the underlying EJB platform.

The system described by the authors relies on markup tags that classifies an attribute as ‚read-only' (changes never), ‚cachable' (changes rarely) or ‚volatile' (changes often, non-cachable). These markups tags are provided by the application programmer at design time. Alternatively, the cache system can keep track of an attributes rate of change and decide automatically how to classify an attribute.

The caching code is located at the stub layer of the application, thus ensuring client transparency. The authors assume that object attributes are more often read than written, thus making attribute values suitable for caching. A source code generation tool is used to automatically generate the implementation of the caching functionality and the CORBA Interceptor facility [56] is employed to hide the cache functionality from the client.

Cache consistency is realized with an expiration time model where the time-to-live value of an attribute is calculated automatically: The system keeps track of attribute changes and uses the mean time of change as the time-to-live value.

Pohl's and Schill's system implements caching in the stub layer of a distributed application, thus enforcing client transparency. Moreover, since the caching implementation is automatically generated by a generator tool, implementation effort is minimized and the approach can be considered applicable to a vast number of different applications.

However, the drawback of the approach is that it lacks a fully developed cache consistency approach. Taking the mean time of change as the expiration time for an attribute means that

[11] An Enterprise JavaBean is basically a CORBA object that implements application-specific functionality as well as functionality required by the EJB platform.

the system is not suitable for scenarios where attributes change rarely but updates have to be propagated to clients as fast as possible, for example in alert systems. In DOC-CaP, the time-to-live values for each method is specified by the application programmer at design time, thus allowing the adaption of the level of cache consistency to client needs.

Moreover, the authors do not provide a case study so that possible performance gains can only be estimated. Prefetching attribute values is not mentioned in their work, although an extension in this direction seems feasible.

By caching only attribute values and excluding the caching of method result values, the project makes the assumption that the retrieval of an attribute value from the server cannot have side effects. While this may be a considerable heuristic, one has to consider that retrieving an attribute value is implemented as a method by the underlying EJB platform. Theoretically, there may be side effects in this method, in which case the attribute should not be cached.

Method-based Caching

Pfeier and Jakschitsch describe a method-based caching system for multi-tiered server applications [61] based on the Java 2 Enterprise Edition Platform [59]. Result values of remote method calls are stored in a client-side cache, and upon invocation of the same method, the cached result value is returned, thereby saving a network roundtrip.

The authors distinguish between read and write methods. If a method is guaranteed to not modify any of the other method's result values, it is called a read method. Otherwise it is called a write method.

A cache model must be defined by the application programmer, which specifies in a formal way how methods modify each other's result values. Upon invocation of a write method, the system invalidates the cache entries for each read method that might be affected by the write method. The authors argument that their system provides strong cache consistency in the case where only one client is connected to a server and the server state does not change in-between method calls of that client.

If two or more clients are connected to the server or if the server state is updated by a third entity, the client-side cache may become out-of-date. We called these cases ‚system inconsistency' and ‚global inconsistency' in the discussion of cache consistency mechanisms, see sections 2.1.1 and 3.2.1. These sources of cache inconsistencies are called ‚cache bypassing' by Pfeier and Jakschitsch and are not handled by their system at the time of their publication. Possible solutions are proposed only briefly as future work, for example various server invalidation approaches or an expiration-based approach.

In the evaluation of the approach, the authors report on speedup factors of 6 to 10 in their specific test setup, which included an object benchmark based on RUBiS [54] and a Web frontend generated by Java Server Pages [60].

Pfeier's and Jakschitsch's system provides strong cache consistency for the local case (only one client connected to one server) through the use of a formal cache model and cache invalidations upon write methods. The cache model has to be manually created by the application programmer. Analogous to our observation, the authors state that automatically deriving the cache model from the source code of a given application will not be feasible with today's development tools and programming languages.

The authors did not report about any intentions to integrate prefetching mechanisms into their work, although this is feasible as the proposed cache model contains information about method side effects. However, for method prefetching it is not sufficient to have a model of method interdependencies. Instead, the system has to have knowledge about method side effects on the server, even for methods that do not affect the result values of other methods. The DOC-CaP approach asserts that methods with side effects are not prefetched, see section 3.3.1 for a discussion.

4. IMPLEMENTATION

The goal of DOC-CaP is to speed up distributed application performance by prefetching and caching remote method result values. For maximum reusability, maintainability and ease of use, prefetching and caching are implemented and executed transparently to the client and the server software. To avoid the need for client hints as in informed prefetching approaches, the prefetching system in DOC-CaP is self-learning. It gathers knowledge about client behavior while the distributed application is running.

DOC-CaP implements caching and prefetching functionality in the stub layer of a distributed application. This approach has two major advantages. First, the client source code as well as the server source code of a distributed application project can be written and tested as if no caching or prefetching takes place. Second, since all prefetching and caching code resides in the stub layer and the stub layer is automatically generated by an IDL compiler (see section 1.2), the effort of adding caching and prefetching functionality to the distributed application is minimal.

As described in section 3.3.1, a method may have side effects. Additionally, each method result value has a certain time-to-live value. Both of these properties affect the way in which DOC-CaP handles caching and prefetching the result value of a method. Therefore, DOC-CaP has to have knowledge of these method properties: The programmer supplies information about method side effects and time-to-live values in the IDL definition of a given application. The IDL definition is then used by an IDL compiler to automatically generate the stub layer of the application.

Inserting caching and prefetching code in the stub layer makes the DOC-CaP approach highly suitable for existing systems, in particular legacy applications that are built with Distributed Object Computing systems. The only thing that has to be done to incorporate caching and prefetching into an existing application is to annotate the IDL definition, generate the stub layer with the DOC-CaP IDL compiler and recompile the application.

We introduced an IDL definition language which provides constructs that let the programmer specify side effect and time-to-live information in a declarative manner. These declarations serve two things. First, the cache consistency approach implemented in DOC-CaP uses the information to invalidate cached data items. Second, the prefetching prediction uses the information to decide which methods are prefetchable. The cache consistency approach used in DOC-CaP, “Expiration Model With Client Invalidation” is described in detail in section 2.1.1. The prefetching algorithm is described in section 4.3.

4.1. DOC-CaP Framework

The DOC-CaP Framework provides the functionality for remote object communication with caching and prefetching and is derived from the Distributed Object Computing framework introduced in section 1.2. Figure 63 presents the major abstractions of the DOC-CaP Framework. Note that the class diagram does not show *Client* and *Server* implementations, as these are not affected by the DOC-CaP extensions.

The main difference between the Distributed Object Computing framework and the DOC-CaP framework is the addition of caching and prefetching facilities. The DOC-CaP framework is described in the following.

Request is the base class of two specified subclasses: *SingleRequest* and *MultiRequest*. A *SingleRequest* object holds information about the remote method call, whereas a *MultiRequest* is a container for *Request* objects and as such holds information about a number of remote method calls. *Request* objects are forwarded to a *Communication* object for transfer to the server. The *Communication* class converts a *Request* – which can be a *SingleRequest* or a *MultiRequest* – into binary data and sends this data over the network. The classes *Request*, *SingleRequest* and *MultiRequest* are designed according to the Composite Design Pattern [36].

The *Response* class and its subclasses *SingleResponse* and *MultiResponse* hold the information about the method result values. For each *SingleRequest* sent to the server, the server answers with a *SingleResponse*. For each *MultiRequest*, the server answers with a *MultiResponse*, where the sequence of *Response* objects contained in a *MultiResponse* matches the sequence of *Request* objects contained in the *MultiRequest*.

CcClientStub is the abstract base class for all client stub classes that incorporate caching and prefetching. It is derived from the *ClientStub* base class and inherits the `ObjectId` and the *invoke()* method for sending a *Request* to the network and waiting for a *Response*. Additionally, a *CcClientStub* object has an association with a *Cache* and a *StubData* object.

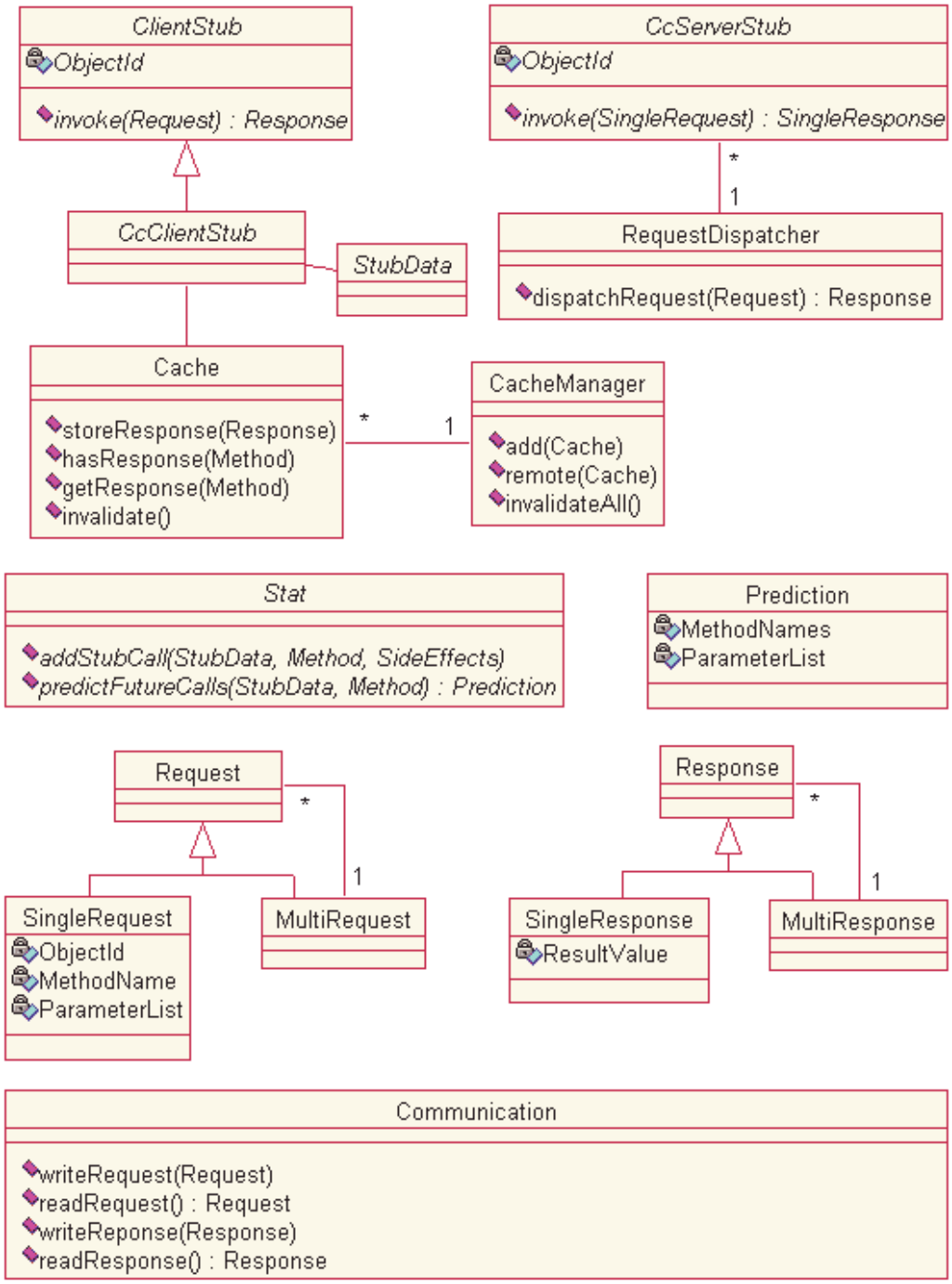


Figure 63: DOC-CaP Framework

A *Cache* object stores result values of prefetched methods for a stub. The methods of the *Cache* class are used to store result values, query the *Cache* if it contains the result value of a certain method and retrieve the result value. Additionally, the *invalidate()* method is used to invalidate the cache by clearing its contents. Note that a *Cache* does not support the removal of single result values, as this is not needed by the DOC-CaP cache replacement approach described in section 3.2.2.

The *CacheManager* is a container that holds associations to all *Cache* objects of the distributed application. When a *Cache* is created, it is added to the *CacheManager* through the *add()* method. When the *Cache* is deleted, it is removed from the *CacheManager* with the *remove()* method. The *invalidateAll()* method calls the *invalidate()* method of all *Cache* objects that are currently registered with this *CacheManager*.

StubData is an abstract base class for a container class that is used by implementations of the DOC-CaP framework to store information about a *CcClientStub*, for example a unique id of the stub and statistical information. It is an abstract class because different implementations of the DOC-CaP Framework will require different *StubData* implementations.

Stat is the abstract base class for the implementation of a specific prefetching prediction algorithm. New prediction algorithms can be incorporated in the DOC-CaP Framework by subclassing *Stat* and implementing the algorithm in the subclass. The abstract base class *Stat* provides the method *addStubCall()* to be called by the DOC-CaP Framework whenever a stub method is called by the client application. The DOC-CaP Framework passes three parameters to *addStubCall()*. The *StubData* object contains information about the *CcClientStub* that was called. The *Method* object contains data for uniquely identifying the called stub method. The *SideEffects* parameter is a boolean flag indicating whether this method has side effects or not. The method *predictFutureCalls()* is called by the DOC-CaP Framework to start the prediction algorithm and to get a *Prediction* object that holds information about the stub methods that are likely to be called in the future. The DOC-CaP Framework passes information about the stub and the called stub method through a *StubData* and a *Method* object.

Communication is the connection to the network. The method *writeRequest()* marshalls a *Request* object into a data format that can be transferred over the network and writes it to the network transport layer. The method *readRequest()* reads data from the network transport layer and unmarshalls it into a *Request* object. The methods *writeResponse()* and *readResponse()* are used for marshalling and unmarshalling *Response* objects.

CcServerStub is the base class for all server side stubs that are part of a DOC-CaP Framework implementation. The field *ObjectId* is used for the identification of a *CcServerStub*. The `invoke()` method is called to forward a `SingleRequest` to the object implementation that this *CcServerStub* represents. The *CcServerStub* calls the

method implementation of the *Server* object implementation (*Server* object implementations are not shown in Figure 63 since they are not part of the DOC-CaP Framework). After executing the method implementation, the *CcServerStub* creates a *Single*Response object, sets its *ResultValue* attribute and returns it.

Every *CcServerStub* must register with *RequestDispatcher*. The *RequestDispatcher* has a main loop that looks for *Request* objects from the network, via *Communication.readRequest()*. Upon receipt of a *Request*, the *RequestDispatcher* unfolds the Request object, which contains of *SingleRequest* and *MultiRequest* objects. Each *SingleRequest* contained in the *Request* is forwarded to the *CcServerStub* with the matching *ObjectId*. When the *Request* object is traversed and all *SingleRequests* have been executed, the *RequestDispatcher* creates a *Response* and sends it back to the client through *Communication.writeReponse()*.

In section 1.2 we have described how the components of a Distributed Object Computing system interact with each other to realize remote method calls. In the following we present the pseudo code for two sample stub methods, one for Distributed Object Computing systems, and one for the DOC-CaP system. Both samples are automatically generated by an IDL compiler. The sample stub methods are based on the AddressBook sample application introduced in section 1.4.1.

The source code in Figure 64 presents the *getName()* method of the *PersonStub*. It shows how stub methods are implemented in Distributed Object Computing systems. The implementation of Distributed Object Computing stub methods expose the following pattern: A Request is created (line 3), then the request is invoked by calling the *invoke()* method of the *PersonStub* (line 4) and the result value is returned (line 5).

```
1 public java.lang.String getName()
2 {
3    Request request = createRequest( "getName" );
4    Response response = invoke( request );
5    return (java.lang.String) response.ResultValue;
6 }
```

Figure 64: Pseudo Code Of An IDL Stub Method

The source code in Figure 65 presents the *getName()* method of the *PersonStub* class that implements caching and prefetching by using the DOC-CaP Framework. We assume that the *getName()* method has a time-to-live value of 5000 milliseconds and that it does not have any side effects.

```
1  public String getName()
2  {
3     String result = null;
4     if ( myCache.hasResponse("getName") )
5     {
6        Response response = myCache.getResponse("getName");
7        result = (java.lang.String) response.ResultValue;
8     }
9     else
10    {
11       Prediction pre = Stat.predictFutureCalls( myStubData, "getName" );
12       Request request = createRequest( "getName", pre );
13       Response response = invoke( request );
14       myCache.storeResponse( response );
15       result = (java.lang.String) response.getFirst().ResultValue;
16    }
17    Stat.addStubCall( myStubData, "getName", false );
18    return result;
19 }
```

Figure 65: DOC-CaP Stub Method (Cachable)

The stub method queries the cache whether it has already a *Response* for the *getName()* method (line 4). If the *getName()* result value is in the cache and its time-to-live time has not yet expired, then the *Response* is taken from the cache (lines 6-7). If the *getName()* result value is not in the cache or its time-to-live time has expired, the prefetching prediction algorithm is asked what stub calls are likely to happen in the future (line 11). The resulting *Prediction* object contains the methods that the prediction algorithm chooses for prefetching. Then, a *Request* is created that contains the *getName()* method as well as all methods that were predicted to be called next (line 12). This *Request* is then sent to the server by calling *invoke()* (line 13). The method *invoke()* marshalls all method requests into one message[12] and sends it over the network to the server side. At the server side, the *RequestDispatcher* receives and unfolds the *Request* and forwards all contained *SingleRequest* objects to the according *ServerStub* objects. Then it creates a *Response* and sends it back to the client. In the meantime, the stub waits for the *Response*, which contains the result values of all *Request* method requests. All result values contained in the *Response* are then stored in the cache for later retrieval (line 14). The result value of the originally called method – *getName()* in this case – is returned to the calling client (line 15). Before the *getName()* result value is returned to the client, the occurrence of this *getName()* stub call is reported to the prediction algorithm via *Stat.addStubCall()* (line 17).

As described in section 3.3.1, methods with side effects are not prefetchable. Moreover, if a method with side effects is invoked by the client, the client cache must be invalidated, which

[12] On the network transport layer, the message can be split up into multiple packets, depending on the network technology that is used.

means that the *Cache* of all client-side stubs have to be invalidated. The source code in Figure 66 presents an example of a method that has side effects on the server. The method *Person.setName()* sets a new name for a person and returns the new name as a *String* object.

```
1  public String setName( String name )
2  {
3     CacheManager.invalidateAll();
4     Prediction pre = Stat.predictFutureCalls( myStubData, "setName" );
5     Request request = createRequest( "setName", pre );
6     request.addInParameter( name );
7     Response response = invoke( request );
8     myCache.storeResponse( response );
9     Stat.addStubCall( myStubData, "setName", true );
10    return (java.lang.String) response.getFirst().ResultValue;
11 }
```

Figure 66: DOC-CaP Stub Method (Not Cachable)

Since the *setName()* method has side effects and is not cachable, the stub method does not look in the cache for a *Response* for the *setName()* method. Instead, the *CacheManager* is called to invalidate all *Cache* objects (line 3). The *CacheManager* invalidates all *Caches* by calling the *invalidate()* method for each registered *Cache* object. Then the prediction algorithm is asked to predict future calls (line 4). Note that while *setName()* is not prefetchable itself, it can still serve as a prefetching trigger upon which other methods might be prefetched. A Request for *setName()* and all predicted methods is created and the *name* parameter is added to the *Request* (line 5-6). Then the *Request* is invoked (line 7). Upon return of the *invoke()* method, all *Response* objects are stored in the cache and the prediction algorithm is informed about the current *setName()* stub call through *Stat.addStubCall()* (line 9). Finally, the result value of the originally called method is returned to the client (line 10).

4.2. XIDL – Extended IDL

In Distributed Object Computing systems, the stub layer of a distributed application is automatically generated from an Interface Definition Language (IDL) description. The IDL description specifies the object interfaces that the server implementation of a distributed application provides to its clients. One example of an IDL is the OMG IDL, which is part of the CORBA standard. The OMG IDL definition for the AddressBook sample application was shown in section 1.4.2, Figure 23.

We have extended the OMG IDL syntax by constructs that let the developer of a distributed application supply information about remote methods. This extended IDL is called XIDL in

the remainder of this document. Additionally, we have implemented an XIDL compiler, which automatically generates the stub layer of a distributed application from an XIDL file. The XIDL format as well as the implementation of the DOC-CaP XIDL compiler are described in this section.

Figure 67 presents the XIDL definition for the AddressBook sample. For backward compatibility, we decided that the information needed by the DOC-CaP XIDL compiler has to be provided via IDL comments. This assures that any XIDL definition can still be passed to a standard IDL compiler for producing standard non-caching stubs. In XIDL, the developer can specify for each method its time-to-live value and information about method side effects. This information affects the way how the DOC-CaP XIDL compiler generates caching and prefetching source code into the stub layer of the distributed application.

```
interface Person // ttl 5000
{
        string getName(); // modifies none
        string getEmail(); // modifies none
        string getPhone(); // modifies none
};
typedef sequence<Person> PersonList;

interface AddressBook // ttl 5000
{
        PersonList search( in string x ); // modifies none
};
```

Figure 67: AddressBook Sample XIDL definition

The time-to-live value of a method result value is given in a *ttl* comment. The *ttl* construct specifies the time-to-live value for the result value of a method in milliseconds. Specifying a time-to-live value on an interface is a shortcut for specifying the time-to-live value for each method of this interface. In Figure 67, each of the three *Person* methods, *getName()*, *getEmail()* and *getPhone()* have a time-to-live value of 5000 milliseconds. The *search()* method of the *AddressBook* interface has a time-to-live value of 5000 milliseconds as well.

The information about method side effects is given in a *modifies* comment. The developer of the distributed application uses the *modifies none* declaration to express that a method does not have side effects. Methods without the *modifies none* construct might have side effects[13].

[13] The problem of automatically deciding whether a method has side effects is discussed in section 3.3.1.

What happens if a method does not have side effects but a developer leaves out a *modifies none* declaration? In this case, the DOC-CaP XIDL compiler must assume that the method might have side effects. Consequently, the method will never be prefetched and upon calling this method, the cache will be invalidated. If a method does have side effects and the developer accidentally adds a *modifies none* declaration for this method, the DOC-CaP XIDL compiler assumes that the method does indeed have no side effects and might decide to prefetch this method. Moreover, upon calling this method, the client cache is not invalidated. Obviously, application correctness cannot be guaranteed in this case. The developer has to be careful to add *modifies none* declarations only for those methods that are guaranteed to have no side effects.

When the DOC-CaP XIDL compiler is invoked on a XIDL input file, it produces Java source code using the DOC-CaP Framework implementation classes for caching and prefetching methods. Moreover, the DOC-CaP XIDL compiler can be advised to generate source code for monitoring stub calls and the invocation of remote requests. This functionality is enabled or disabled with two command-line options:

-optimize	If this command line option is passed to the XIDL compiler, it generates DOC-CaP caching and prefetching code into the stubs and skeleton classes. If the ‘optimize’ option is left out, normal CORBA stub classes are generated, similar to the output of commercial OMG IDL compilers, for example JacORB [14].
-monitor	If this command line option is passed to the XIDL compiler, it generates additional code into the client stubs, which helps to track stub calls and remote method calls. So, the developer can see when remote method calls (normal CORBA requests as well as DOC-CaP requests) are transferred to the CORBA network layer.

The DOC-CaP XIDL compiler generates source code for a stub layer that uses DOC-CaP Framework classes for implementing caching and prefetching for distributed applications.

4.3. Prefetching Prediction Algorithm

While caching facilities are built into the DOC-CaP Framework, no prefetching prediction algorithm is provided by the framework. The DOC-CaP Framework supports the implementation of different prefetching prediction algorithms via the abstract class *Stat*. To implement a new prefetching prediction, one has to subclass from the *Stat* class and implement the desired prediction algorithm in the subclass. This section presents the prefetching prediction algorithm used in our implementation of the DOC-CaP Framework.

Before describing the prefetching prediction algorithm used in our implementation of DOC-CaP, we introduce two types of prefetching prediction spaces, fixed prediction spaces and

dynamic prediction spaces. In both cases, we look at sets of data entities that might be scheduled for prefetching by the prediction algorithm. A data item that may be scheduled for prefetching is called a prefetching candidate.

With fixed prediction spaces, the number of prefetching candidates is fixed and the data entities that are subject to prefetching can be numbered from 0 to a maximum value. For example, the main memory of a computer can be seen as a flat storage area where the individual memory blocks can be numbered from *0* to *n* where *n* is the address of the last memory block. A prediction algorithm for memory prefetching has to calculate the function *next(x)* where *x* is the number of the memory block that has been requested by the CPU and the result value of *next(x)* contains the numbers of the memory blocks that should be prefetched. In file systems, the prefetching space can be organized as harddisk sectors, for example. In Web prefetching, the prefetching space is organized as Web documents, which can be addressed by unique URLs[14].

If prefetching one data item produces more prefetching candidates, then the set of prefetching candidates is dynamic. This is the case in Distributed Object Computing systems. The number of remote methods that can be called by a client cannot be numbered from 0 to a maximum value. Instead, each method of an object can create and return objects, which can again have a number of methods. Since the number of objects might be changing with every prefetching action, the number of methods is changing as well. An example is the Address Book method *search()*. This method returns a number of *Person* objects, and each *Person* object has a number of methods (*getName()*, *getEmail()*, ...) that could be prefetched along with the call to *search()*.

Each method call is initiated by a client and is targeted against an object that implements the called method. In the case of Distributed Object Computing systems, method calls are targeted against stub objects, therefore we call them ‘stub calls’. Figure 68 presents a scenario of the Address Book sample application: A client calls the methods *getName()* and *getEmail()* in *PersonStub* objects *p1* and *p2*.

[14] This assumption does not apply for dynamically generated pages with changing query URLs.

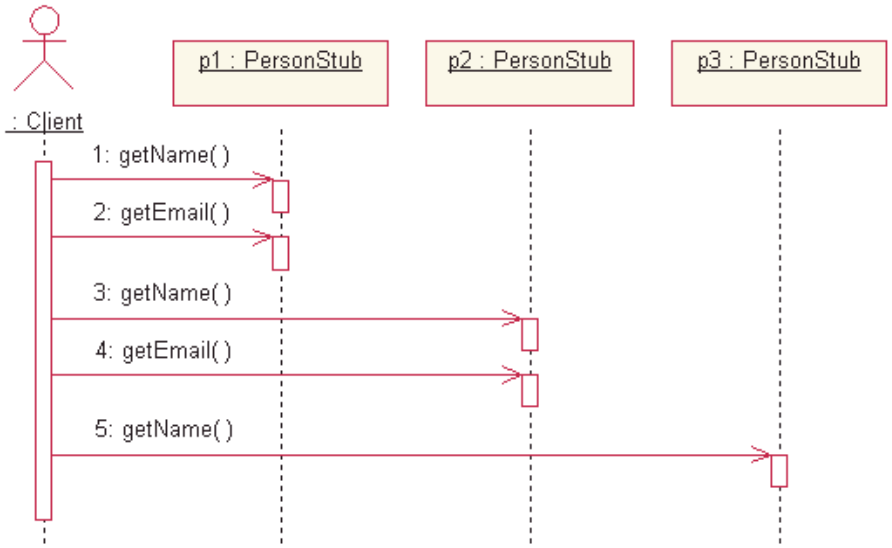


Figure 68: AddressBook Prediction Scenario

The question is if the client will call the method `p3.getEmail()` and if the result value of `p3.getEmail()` should be prefetched at the time `p3.getName()` is called. Since `getName()` was followed by `getEmail()` in the case of `p1` and `p2`, the probability is high that `getEmail()` will be called next in `p3` too.

In the following we present the DOC-CaP prefetching prediction algorithm used in our implementation and in the evaluation of the DOC-CaP approach. The algorithm has two parts. The first part, accessed via the `addStubCall()` method of the `Stat` class, keeps track of the stub calls initiated by a client. The algorithm maintains a data structure used to store information about methods that were called in the past. The second part is responsible for predicting future stub calls. This functionality can be accessed through the `predictFutureCalls()` method of the `Stat` class.

The algorithm uses a tree-like data structure to keep track of stub calls. There is one tree per IDL interface and each tree node contains information about a stub call (e.g. the method name) and a counter that indicates how often the stub method has been called by the client. We call such a tree an 'invocation tree[15]'. At the beginning, the invocation tree does not contain any nodes except a 'root' node that is labeled with the name of the IDL interface that the invocation tree represents. If a client invokes a method in a stub, a stub call event is reported to the prefetching prediction algorithm via the `addStubCall()` method of the `Stat`

[15] Note that the term 'invocation tree' is also used in other fields of computer science, for example in language compilers and processor architectures. Here, we use the term 'invocation tree' to describe the internal data structure that is used by our prefetching prediction algorithm.

class. The abstract method *addStubCall()* was introduced in the DOC-CaP Framework in section 4.1. In this section, we present the implementation of *addStubCall()*.

Figure 69 presents the pseudo code of our implementation of the *addStubCall()* method.

```
1  public void addStubCall( StubData stubData, Method method, boolean hasSideEffects )
2  {
3     Node currentNode = null;
4     if ( hasSideEffects )
5     {
6        currentNode = stubData.getRootNode();
7     }
8     else
9     {
10       currentNode = stubData.getCurrentNode();
11    }
12    Node childNode = currentNode.getChildNode( method );
13    childNode.Counter = childNode.Count + 1;
14    stubData.setCurrentNode( childNode );
15 }
```

Figure 69: Pseudo Code Of addStubCall()

The *stubData* object contains information about the *CcClientStub* object that was called by the client. For example, each *CcClientStub* object has a reference to the invocation tree node that represents the method that was last recently called in the *CcClientStub*. This node is called the 'current' node of the *CcClientStub* object. If a *CcClientStub* object is created, a *StubData* object is created, too, and the current node is the root node of the invocation tree for that *CcClientStub*.

The *method* object contains an identifier of the method that was called by the client. The identifier is a string value that uniquely describes a stub method.

The *hasSideEffects* parameter is set false if the method has a *modifies none* declaration in the IDL description, otherwise it is set true.

Depending on *hasSideEffects*, the current node is set as either the root node of the invocation tree for this stub or the current node that is stored in the *StubData* object (lines 4-11 in Figure 69). Then, a sub node of the current node is searched, which represents the called *method*. If no such child node is found, one is created implicitly (line 12). The counter of the child node is incremented by one (line 13) and then the child node is set as the current node of the *StubData* (line 14). Thus, each branch of an invocation tree represents a sequence of stub calls where each stub call method is without side effects. Each tree node has a counter that represents how often the sequence of methods from the root down to this tree node has been called by a client.

An example of an invocation tree for the Address Book sample scenario is shown in Figure 68. The first method that the client calls is the `getName()` method of the `PersonStub p1`, which implements the `Person` interface. At the beginning, the `stubData`'s current node is the root node of the `Person` invocation tree. The algorithm retrieves the child node of the current node that represents the method that was called, `getName()` in this case. If no child node exists for this method, a new invocation tree node is created for the called method and its counter is set to zero. Now the counter is incremented by one and the child node is set as the current node of the `PersonStub` that was called. Figure 70 shows the invocation tree after the `getName()` stub call.

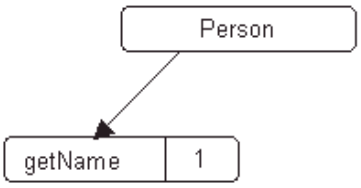


Figure 70: Invocation Tree (1)

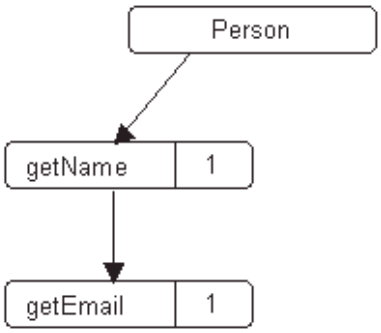


Figure 71: Invocation Tree (2)

The next call issued by the client is `p1.getEmail()`. Again, the algorithm takes the current node, creates a new child node representing the `getEmail()` method, increments the node counter by one and sets the new child node as the current node for the `PersonStub p1`. The invocation tree for the `Person` interface after this call is shown in Figure 71.

The client calls now `p2.getName()`. Since the `PersonStub p2` is called the first time, its current node is still the root node. This time, a child node for `getName()` exists already, its counter is incremented by one and it is set as the current node of `p2`. When `p2.getEmail()` is called, the already existing `getEmail()` node is increment by one and the invocation tree for the `Person` interface looks like shown in Figure 72, which is the same tree as in Figure 71 but with the counter values for both the `getName()` node and the `getEmail()` node set to 2. Figure 73 shows the invocation tree after the client calls `p3.getName()`.

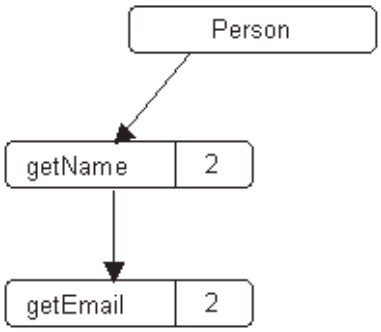


Figure 72: Invocation Tree (3)

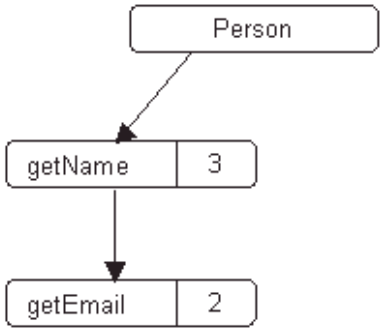


Figure 73: Invocation Tree (4)

If a method is called in a stub, the stub method asks the prediction algorithm to predict future stub calls. The prediction algorithm uses its invocation trees to find the methods that are likely to be called in the future. The prediction functionality is accessed through the `predictFutureCalls()` method of the `Stat` class. With the invocation tree shown in Figure 73, the algorithm would predict that a `getName()` method call will be followed by a `getEmail()` method call with a probability of 0.66.

The prefetching prediction scenario described so far is a scenario of a fixed prediction space. Upon a method call in a `Stub` object, the future calls that can be predicted are the methods that are implemented by that `Stub` object and the set of these methods is fixed.

However, there are cases where it is beneficial to extend the prediction algorithm for dynamic prediction spaces. In the following we present an extension to the prefetching algorithm to support dynamic prediction spaces as encountered in Distributed Object Computing systems.

With Distributed Object Computing systems, clients access server-side objects through client-side stubs. If a client does not have a stub for a server-side object, it cannot communicate with that object[16]. If a client wants to communicate with a server-side object, it has to gain access to a stub for that object. Since a client cannot create a stub object directly, the only way of getting a stub is via the invocation of a method that returns one or more object references. If a stub method is called that produces new stubs, the called stub is called the 'parent stub' and the newly created stubs are called 'child stubs'.

An example is the `search()` method of the `AddressBook` interface, see Figure 23 in section 1.4.2. This method returns a list of `Person` object references that are implemented as `PersonStub` objects. For each of the `Person` reference contained in the result list, the

[16] Some Distributed Object Computing systems do support dynamic invocation interfaces, where a stub is not needed. Caching and prefetching is not supported with these types of remote method calls.

methods `getName()`, `getEmail()` and `getPhone()` are called in the sample scenario of the AddressBook Sample application in section 1.4.2. If the `search()` result list contained 100 `Person` references, the invocation tree would look like shown in Figure 74. The parent stub of all `PersonStub` objects is the `AddressBookStub` that implements the `search()` method.

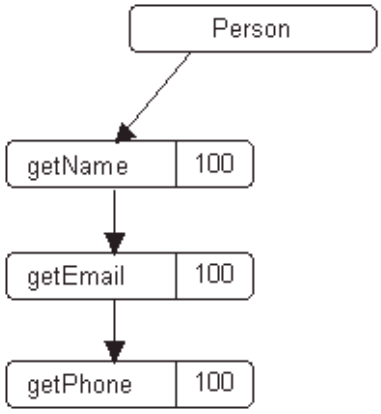


Figure 74: Invocation Tree (4)

The prediction algorithm would predict that `getEmail()` and `getPhone()` would follow a `getName()` call with a probability of 1.0. For each `getName()` call, the methods `getEmail()` and `getPhone()` would be predicted to be called next and scheduled for prefetching. Therefore, for iterating over the person list of 100 entries, 300 stub calls would result in 100 remote method invocations.

Whenever a remote method is called that returns object references, child stubs are created on the fly. We have extended the invocation tree prediction algorithm to support the prefetching of methods that are implemented by newly created child stubs. For example, when the `AddressBook.search()` method is called, it would be beneficial if the methods of the resulting child stub objects, which are `PersonStubs`, would be prefetched along with the `search()` remote invocation. Having this, iterating over the person list and invoking `getName()`, `getEmail()` and `getPhone()` for each `Person` entry of the `search()` result list would involve no remote method invocations, because all `PersonStub` methods would have been already prefetched in the `search()` remote invocation.

We have introduced a pseudo method ‘create_stub’ that is reported to the invocation tree prediction algorithm whenever a stub is created. Having this, the `Person` invocation tree would look like shown in Figure 75 after a client has invoked the `search()` method and has iterated over the resulting person list, calling `getName()`, `getEmail()` and `getPhone()` for each person entry: 100 PersonStub objects have been created and for each stub, `getName()`, `getEmail()` and `getPhone()` have been called once. Figure 75 shows also

the invocation tree for the `AddressBook` interface: One `AddressBookStub` has been created and the `search()` method has been called once.

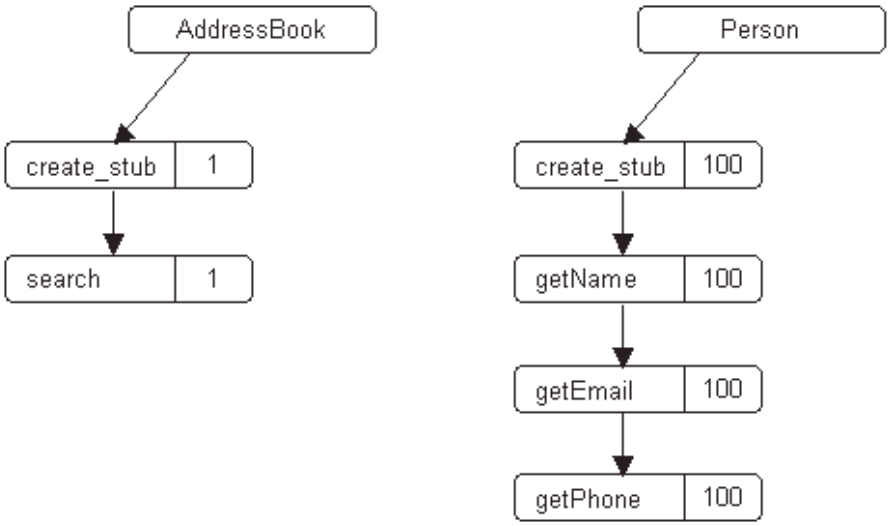


Figure 75: Invocation Tree (5)

The prediction algorithm can now predict that if a `PersonStub` is created, `getName()`, `getEmail()` and `getPhone()` will be called with a probability of 1.0. The DOC-CaP XIDL compiler knows from the XIDL interface definition that `search()` returns `Person` references. Additionally, it knows from the XIDL interface that the `Person` methods `getName()`, `getEmail()` and `getPhone()` do not have side effects. Therefore, it creates code into the `search()` stub method that prefetches `getName()`, `getPhone()` and `getEmail()` for the `Person` entries in the `search()` result list along with the `search()` call itself.

The prediction algorithm described so far can predict, upon a stub call, which methods of the same stub and which methods of newly created stubs will be called. However, there are situations where the sequence of method calls depends on the methods that were called in the past. We call the sequence of stub methods that were called in the past the ‘call context’ of a stub. In the next section we extend the Invocation Tree algorithm to use call context information for predicting future stub calls more precisely. For a motivation of the call context approach, we present a sample that is based on the AddressBook sample IDL definition introduced in section 1.4.2.

Figure 76 shows the definition of an AddressBook sample interface where a new method `searchEmail()` is added to the `AddressBook` interface. We assume that a sample AddressBook client provides the functionality of searching for email addresses and displaying

the search result list. The email address list should only display the email address of a person, not its name nor its phone number.

```
1  interface Person // ttl 5000
2  {
3          string getName(); // modifies none
4          string getEmail(); // modifies none
5          string getPhone(); // modifies none
6  };
7  typedef sequence<Person> PersonList;
8
9  interface Adrbook // ttl 5000
10 {
11         PersonList search( in string x ); // modifies none
12         PersonList searchEmail( in string x ); // modifies none
13 };
```

Figure 76: AddressBook XIDL Definition

If a client calls the *search()* method of the *AddressBook*, it will call *getName()*, *getEmail()* and *getPhone()* for all *Person* references of the *search()* result list. If, however, the client calls *searchEmail()*, it will only call *getEmail()* for each *Person* reference of the *searchEmail()* result list. Let's assume that the user of an *AddressBook* client activates the search function and the *search()* result list contains 100 *Person* entries. The client has to display the *Person* attributes and therefore iterates over the *PersonList*, calling *getName()*, *getEmail()* and *getPhone()* for each *PersonList* entry. At the end of the iteration, the invocation trees for the *AddressBook* and *Person* interfaces look like shown in Figure 75. Then the user activates the 'search email' function and the result list of *searchEmail()* contains 50 entries. The client displays the email addresses to the user, calling *getEmail()* for each *PersonList* entry. At the end of the iteration, the invocation trees look like shown in Figure 77. The problem here is that the next time *getEmail()* is called, the prediction algorithm will predict that *getEmail()* is followed by *getPhone()* with a probability of 100/150. If the prefetching threshold is greater than or equal to 100/150, *getPhone()* will be prefetched every time *getEmail()* is called, even when *searchEmail()* was called and it is guaranteed that *getPhone()* will never be called by the client in this case. If the prefetching threshold is lower than 100/150, *getPhone()* will never become prefetched upon a *getEmail()* call, even if *search()* was called and it is sure that *getPhone()* will be called next. The problem is that in the first case too much data is prefetched and in the second case, not enough data is prefetched. The difficulty is that the exact sequence of stub calls depends on the method call that has created the *Person* stubs.

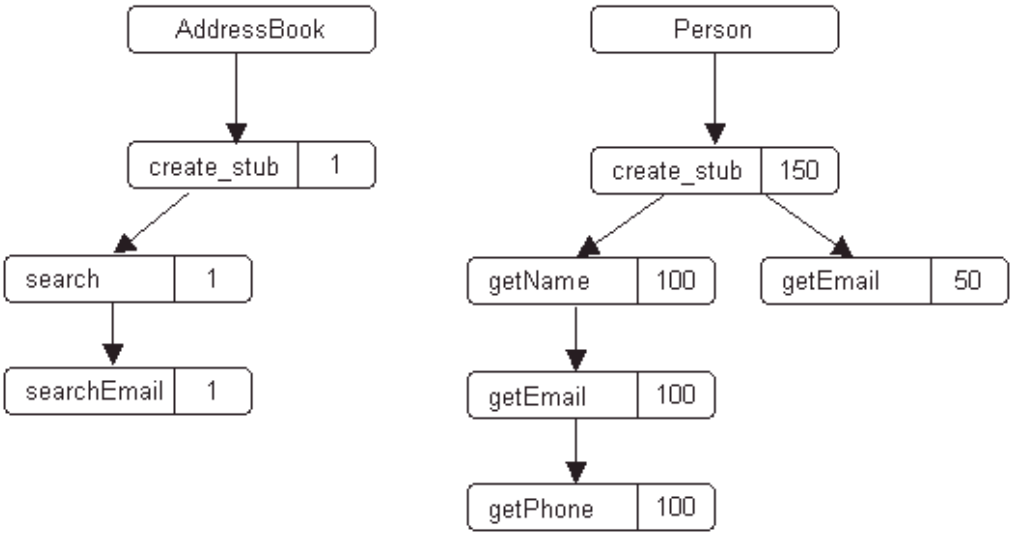


Figure 77: Invocation Tree (6)

To solve this problem we add information about the call context of parent stubs to the prefetching algorithm. Each 'create_stub' node is created as a child node of a call context node. The call context for a stub is defined as the name of the last recent stub call. If we insert call context nodes, the invocation trees for the `AddressBook` and the `Person` interfaces look like shown in Figure 78. Here, the `Person` invocation tree has two branches, one for the `AddressBook.search()` call context and one for the `AddressBook.searchEmail()` call context. For the `AddressBook.search()` call context, 100 `Person` stubs were created and `getName()`, `getEmail()` and `getPhone()` were called on each stub. For `AddressBook.searchEmail()` call context, 50 stubs were created and `getEmail()` was called 50 times.

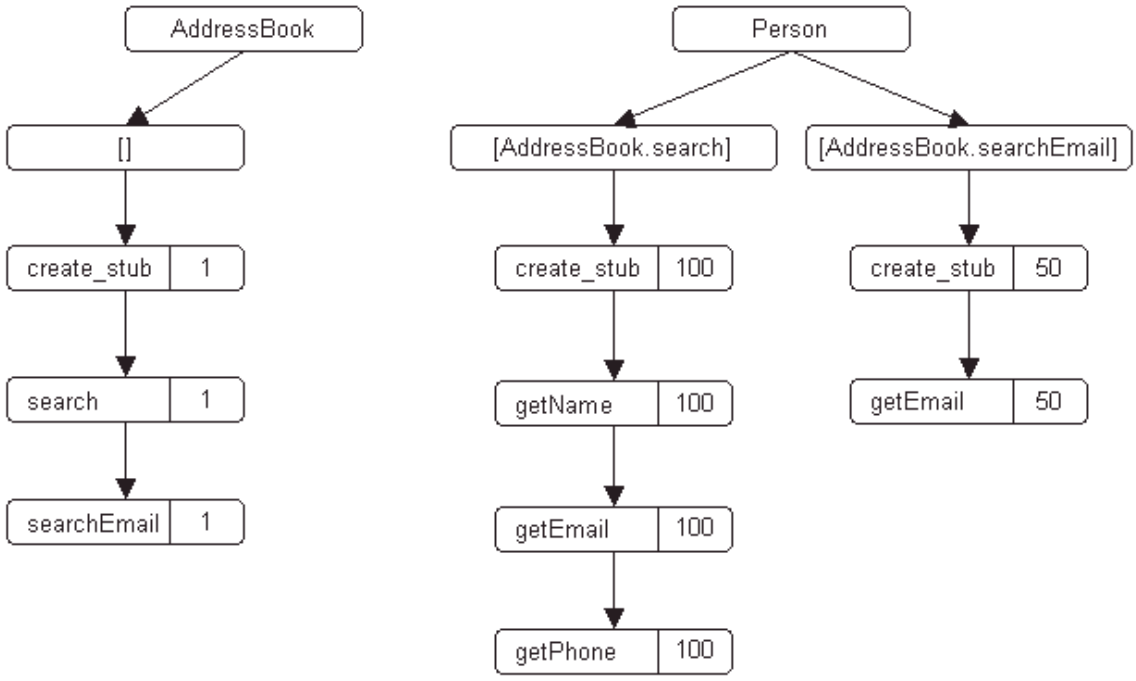


Figure 78: Invocation Tree (7)

When predicting future calls, the prefetching prediction algorithm searches the current call context of the parent stub. If such a context exists, the tree branch of this call context is used for predicting future calls. If the call context is not found, the whole tree is used for calculating the probability of future stub calls. The DOC-CaP Framework implementation that uses the prefetching prediction algorithm described in this section is evaluated in the next section.

5. EVALUATION

In this section we experimentally evaluate the performance of the DOC-CaP system with three test applications. For each of the evaluation experiments, we provide an implementation that uses CORBA for client-server communication and one that uses DOC-CaP. We then measure the performance of these applications and compare the numbers of the CORBA-based implementation and the DOC-CaP-based implementation. We conclude that the DOC-CaP approach can increase distributed application performance significantly.

It is important to note that the user-perceived performance of a distributed application has several factors, as described in section 1.4.4. For example, the time that is consumed by network communication depends on the number of remote method calls and the amount of data that has to be transferred between client and server. Additionally, the time it takes for a server to execute the method implementation adds to the client waiting time. The evaluation applications are implemented in a way that assures that the time it takes to execute a method implementation can be considered negligible. Therefore, our performance comparison of the evaluation applications considers only the communication overhead.

To evaluate the DOC-CaP approach, we chose the following applications:

Address Book Sample Application
The Address Book sample application was introduced in section 1.4 and referenced throughout this work. The Address Book server stores person data for retrieval by an Address Book client. The Address Book client program can be used to search for persons and display their names, email addresses and phone numbers.

TPC-W Benchmark
The Transaction Processing Council (TPC) [35], a consortium of several leading companies of the computer industry, has specified and standardized a number of benchmark applications. The TPC-W benchmark [65] simulates an electronic book store much like amazon.com, where a customer can browse books, add books to a shopping cart, and so on.

AQUA: Automobile Quality Assurance Application
This system is part of a quality assurance system developed at an automobile company. The AQUA system is used by automobile workers, Quality Assurance (QA) personnel and staff managers to collect, distribute and archive data about vehicles that do not meet the desired quality standards.
There are a number of reasons why vehicles fail to meet the quality standards: If they do not pass QA tests, or if some worker notices a damage while assembling parts of a vehicle, for example. When such an incident occurs, the affected vehicles are locked, which means that they cannot be shipped unless the damage is repaired. The AQUA system is used to keep track of such QA incidents.
Upon appearance, an incident can be created by entering data about a failure or a damage in the AQUA system: What types of vehicles are affected, how many parts are affected, who has noticed the damage, who is responsible for managing the repair process, descriptions of the incident, part numbers, vehicle identification numbers, and so on.
Whenever an incident is created, a repair process is started. A typical repair process lasts from one to several days and involves in-house repair personnel as well as external vehicle part suppliers. Each repair action is entered into the AQUA system for long-term storage and retrieval.

Evaluation Test Bed
The DOC-CaP implementation used for the experimental evaluation is based on Java and CORBA. The evaluation test bed, the Java implementation and the CORBA implementation are described in Appendix 9.1. Since the DOC-CaP-based test runs use the same CORBA implementation as the CORBA-based test runs, it is ensured that the evaluation results reflect the performance speedup the can be gained through caching and prefetching of method result values.

The DOC-CaP Caching and Prefetching approach can be applied to other Distributed Object Computing systems as well. Moreover, other programming languages can be used for implementing distributed applications with DOC-CaP, like C++ for example.

However, the performance numbers presented in this evaluation cannot be directly transferred to other Distributed Object Computing systems nor can they be transferred to other programming languages. The choice of both will affect the performance numbers of distributed applications built with or without DOC-CaP. This section presents the empiric evaluation of the DOC-CaP approach using CORBA and Java.

5.1. Address Book Application

For evaluation purposes, we concentrate on the ‘Search By Name’ use case, where a user uses the search function of the address book application to display a list of person entries. The

complete use cases and classes of the Address Book sample application are described in section 1.4, see Figure 19 and Figure 20.

Each person has 3 attributes, a name, an email address and a phone number. Each attribute can be retrieved by the client via `getName()`, `getEmail()` or `getPhone()`, respectively.

```
1  interface Person // ttl 5000
2  {
3          string getName(); // modifies none
4          string getEmail(); // modifies none
5          string getPhone(); // modifies none
6  };
7  typedef sequence<Person> PersonList;
8
9  interface Adrbook // ttl 5000
10 {
11         PersonList search( in string x ); // modifies none
12 };
```

Figure 79: Address Book XIDL Definition

Figure 79 presents the XIDL definition for the address book application. All person attributes have a TTL value of 5000 milliseconds. None of the specified methods have side effects on the server, denoted by a 'modifies none' comment for all methods. The Person's `get()` methods are all prefetchable, since they do not modify the server state and do not have any ingoing parameters. The AddressBook `search()` method cannot be prefetched or cached, since it has an ingoing parameter.

```
1  public static void doSearch(AddressBook aBook, String x)
2  {
3      Person[] personList = aBook.search(x);
4      for each Person p in personList
5      {
6          displayPerson( p.getName(), p.getEmail(), p.getPhone() );
7      }
8  }
```

Figure 80: AddressBook Client Pseudo Code

Figure 80 presents the pseudo code of the AddressBook evaluation test client. The code fragment shown here executes the 'Search By Name' test case by invoking the `search()` method of an `AddressBook` object and iterating over the resulting person list, calling `getName()`, `getEmail()` and `getPhone()` for each `Person` entry.

In our implementation of the 'Search By Name' test case, the `search()` method yields a list of 10 `Person` entries and each `Person` attribute value is a pre-computed string of a fixed length of 10 characters that the server stores in memory. Thus, the execution times of the test

runs are comparable and the execution times of the server side method implementations can be neglected. The test measurements reflect only CORBA communication performance, not any implementation issues like database access or LDAP server lookups for retrieving person data.

The test case involves 31 remote method calls in standard CORBA: one for the *search()* method and 30 for iterating over the resulting *Person* list. With DOC-CaP, the test case involves only one remote method call, as the result values for *getName()*, *getEmail()* and *getPhone()* for each *Person* entry are prefetched when *search()* is called.

We executed the ‘Search By Name’ test case several times to get stable average values. A detailed description of the test setup and the test method can be found in Appendix 9.1 and the evaluation result values can be found in Appendix 9.5.

Figure 81 presents the speedup factors for the Address Book evaluation, for various network delay and bandwidth configurations. The speedup is calculated as

$$speedup = \frac{Twait\_1}{Twait\_2}$$

where *Twait_1* is the client waiting time *Twait* using normal CORBA and *Twait_2* is the client waiting time *Twait* using DOC-CaP.

The x-axis shows the bandwidth, from 160 KB/s down to 5 KB/s. The y-axis shows the speedup factor. The network delay time has been set from 0 milliseconds to 160 milliseconds. The most important result is that the speedup factor depends mainly on the network delay time. The higher the network delay time, the greater is the speedup factor.

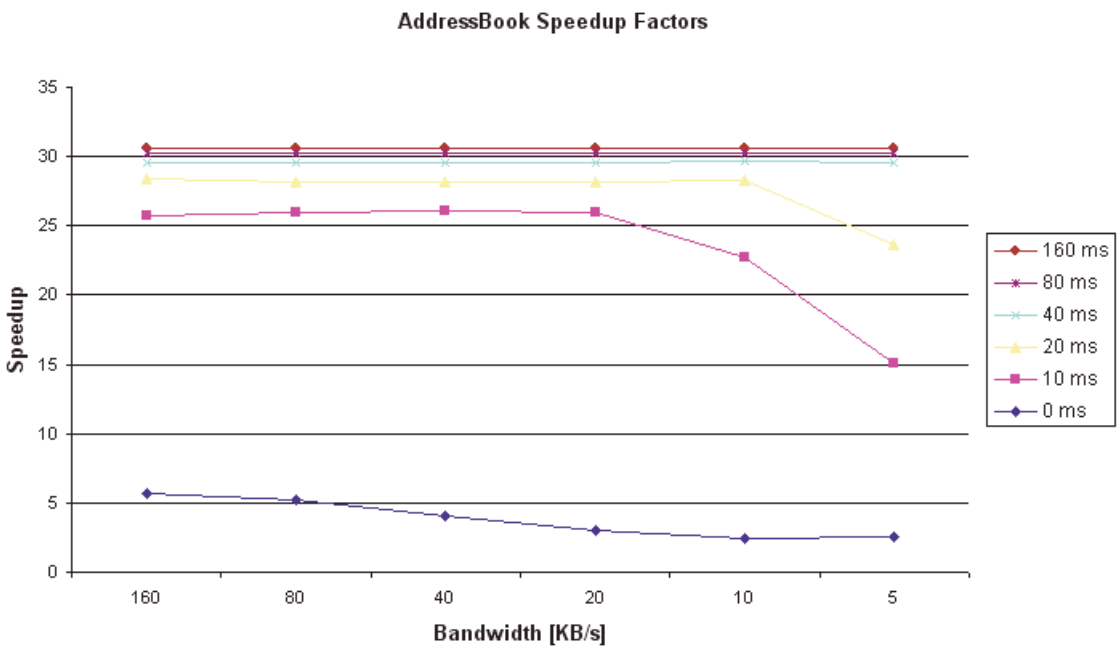


Figure 81: AddressBook Speedup Factors

By using the DOC-CaP approach, the performance of the AddressBook sample application can be increased significantly. The performance gain depends on a number of factors. First, the number of `search()` result length and the number of `Person` attributes determines the factor by which the number of remote method invocations can be reduced. Second, the network parameters (delay time and bandwidth) affect the performance speedup that can be gained by using DOC-CaP.

5.2. TPCW-W Benchmark

While the Address Book sample application has only two interfaces and the invocation pattern is simple, real-world applications typically do have more complex interfaces and invocation patterns. We investigate these applications by implementing an application based on the TPC-W Benchmark [65].

The TPC-W Benchmark specifies an E-Commerce workload that simulates the activities of a bookstore company website, like amazon.com for example. Emulated users can browse and order product items from the website. In the case of TPC-W the items are books. A user is emulated by a ‘Browser Emulator’ that simulates the same network traffic as a real customer would see using a Web browser. At the server side, a database stores information about

registered customers, books and orders. A Web server dynamically produces Web pages that are then sent to the client-side Browser Emulator.

The TPC-W specification describes in detail the 14 different Web pages of the website. The first page the user will see is the bookstore homepage. It includes the bookstore logo, promotional items, a list of best selling books, a list of new books, search pages, the users shopping cart, and order status pages. For each book there is a product page, which will give the user detailed information about that book. The user may order books by entering the order and shopping cart pages. The user can place an order, enter the quantity, or delete a book from the shopping cart. When the user wishes to buy, credit card information has to be entered and the order can be submitted. The system will present the user with an order confirmation page. At any later date the user can view the status of the last order.

The purpose of the TPC-W Benchmark is to provide a standardized basis for the performance evaluation of a given ‘System Under Test’. The System Under Test comprises all hardware and software components that are part of the application being simulated. This includes network connections, Web servers, application servers, database servers, etc. The outcome of a TPC-W benchmark test run is the number of emulated user interactions per seconds.

For the purpose of evaluating the DOC-CaP system, we have implemented the TPC-W entity model in an object-oriented way, using CORBA and Java. The TPC-W Entity Relationship Model is shown in Figure 82. Each attribute can be a character string, a number, a date value or a reference to another entity. The entity ITEM specifies two image attributes: I_THUMBNAIL and I_IMAGE. The TPC-W specification states that these attributes can be binary image data or references (filenames, URLs, etc.) to image data that is stored somewhere else in the system. In our implementation, the I_THUMBNAIL and I_IMAGE attributes are character strings and store image names instead of binary image data.

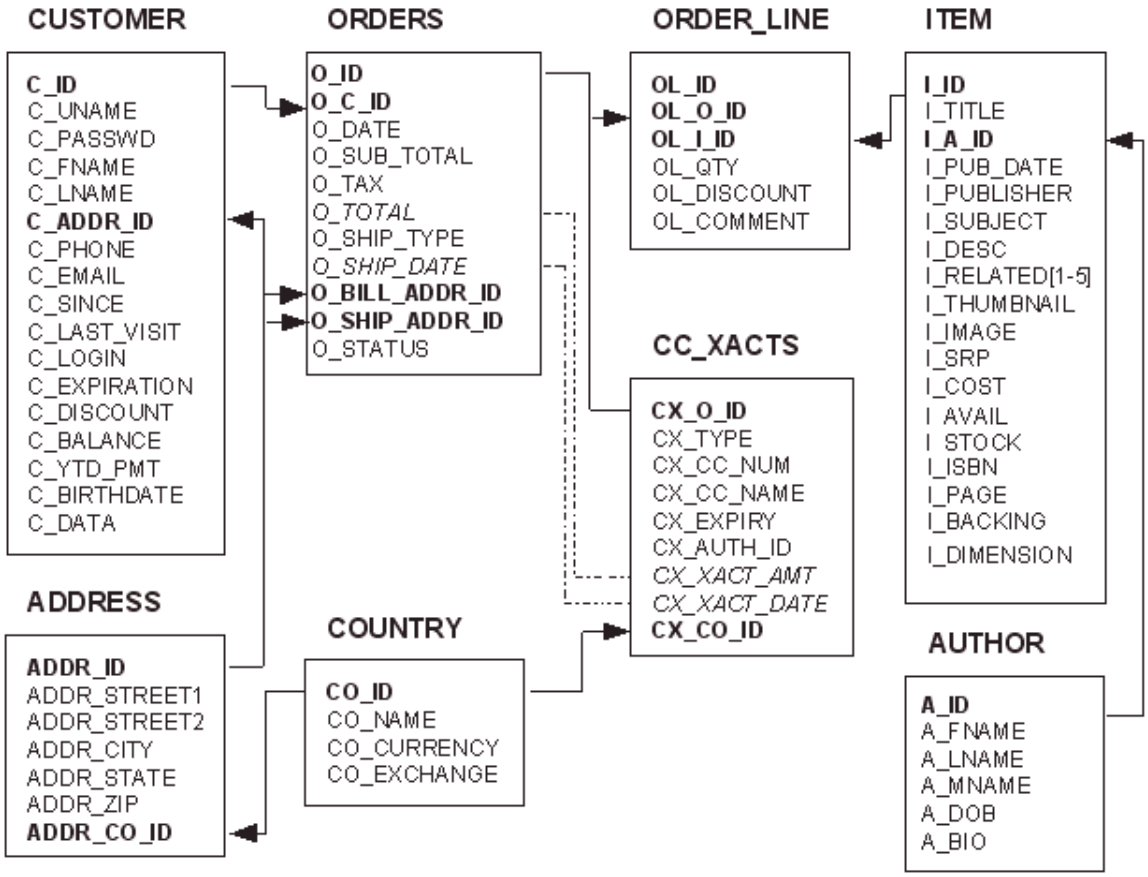


Figure 82: TPC-W Database Entities And Relationships

We defined a CORBA interface for each entity, and a get method for each string, number and date attribute. For each reference, we defined a get-method that returns a reference to the respective interface. The interface definitions contain additional methods that are used to add references or remove references. The TPC-W evaluation class diagram is shown in Figure 83 and the XIDL interface is shown in Appendix 9.6.

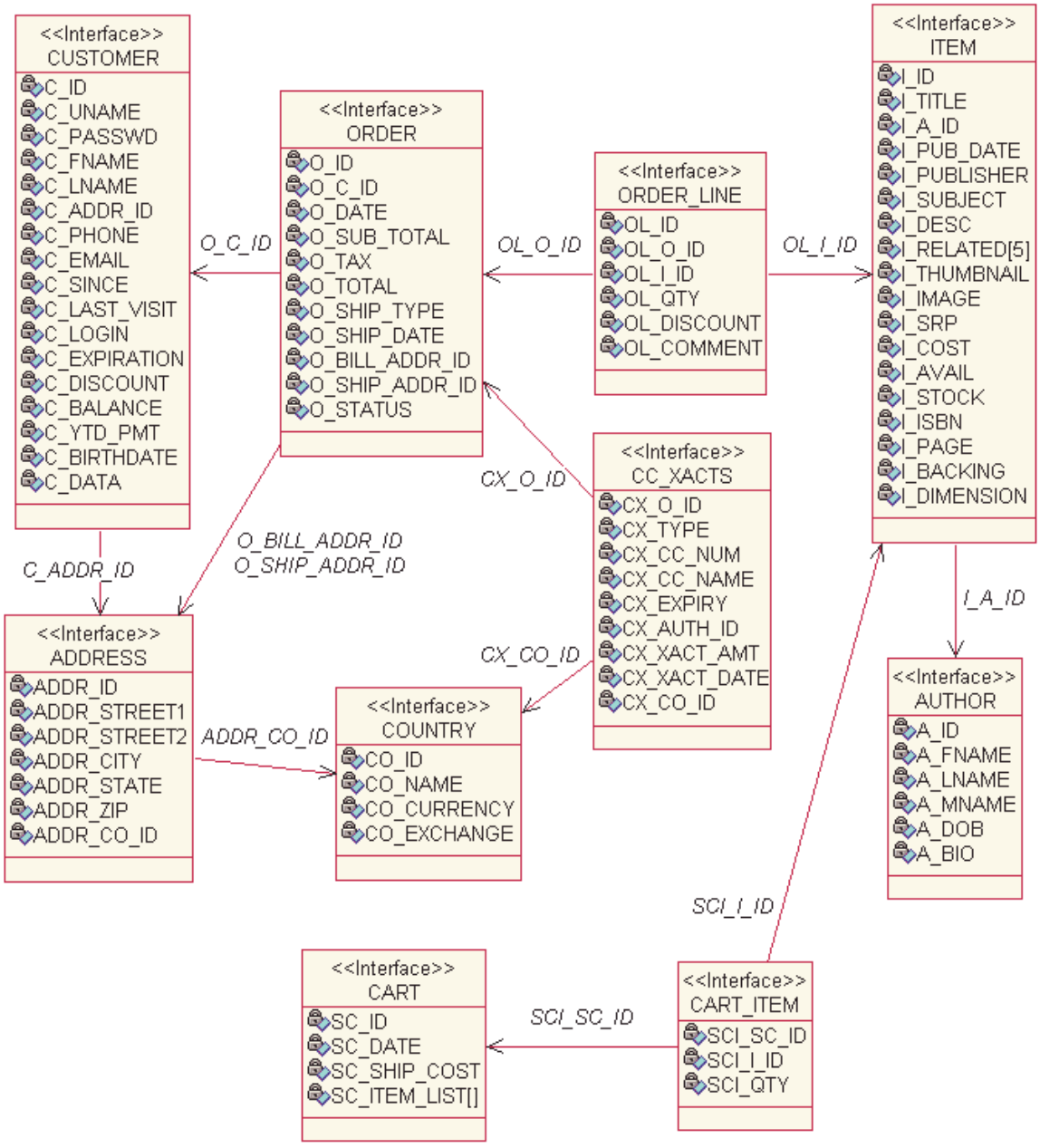


Figure 83: TPC-W Benchmark Class Diagram

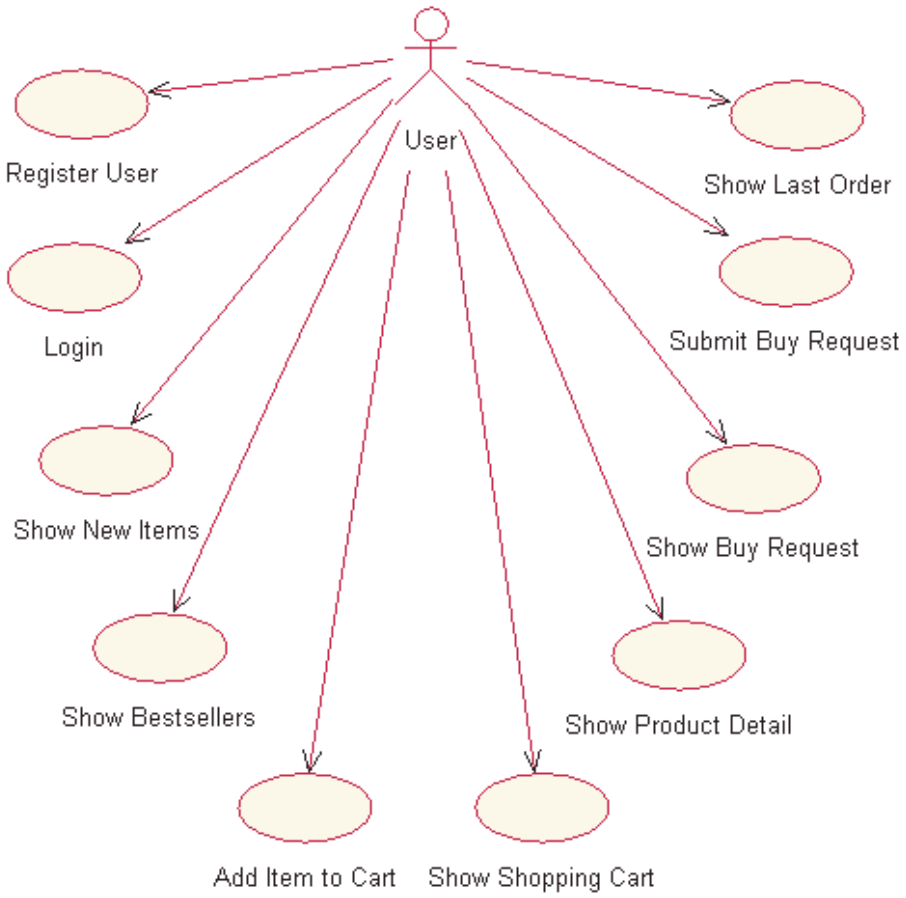


Figure 84: TPC-W Benchmark Use Case Diagram

The TPC-W suite specifies a number of use cases, which are shown in Figure 85. In our implementation, we use a fixed sequence of user transactions to obtain a basis for reproducible and comparable test results. A test run consists of the following user transactions:

1. *Register User*. The emulated user registers with the system by providing a username and a password. A new user account is created by the system and the user is ready for login.
2. *Login*. The emulated user logs in by providing a username and a password. The username and password must be known by the system, which means that a user registration has to be executed before. After successfully logging in, a welcome message is displayed and 5 randomly chosen promotional items are displayed.
3. *Show New Items*. The emulated user requests the list of new items to be displayed. The system displays 50 new items as well as 5 randomly chosen promotional items.

4. *Show Bestsellers*. The emulated user requests the list of bestseller items to be displayed. The system displays 50 bestseller items as well as 5 randomly chosen promotional items.
5. *Show Shopping Cart*. The emulated user requests the shopping cart to be displayed. At this time, the chopping cart is empty. The system displays the shopping cart page along with 5 randomly chosen promotional items.
6. *Show Product Detail*. The emulated user requests the product details of a specific item to be shown. In our implementation, the item to be displayed is identified by an index value that goes from 0 to the number of items stored in the server minus one.
7. *Add Item To Cart*. The emulated user adds the item to the shopping cart.
8. *Show Shopping Cart*. The emulated user requests the shopping cart to be displayed. At this time, the chopping cart contains 1 item. The system displays the updated shopping cart page along with 5 randomly chosen promotional items.
9. *Show Buy Request*. The emulated user requests a buy request page to be displayed. The system displays a page that contains the emulated user's billing and shipment information along with the price of the current shopping cart items.
10. *Submit Buy Request*. The emulated user fills out the shipment and credit card information and submits the buy request page. The system then displays a buy confirmation page, which contains the pricing information and a success message.
11. *Show Last Order*. The emulated user requests the last order page to be displayed.

The TPC-W test application consists of a TPC-W test server and a TPC-W test client that emulates a real user by initiating the sequence of user interactions, i.e. calling the respective methods in the server.

Upon startup, the TPC-W test server populates its database. It is important to note that we did not use a database management system, but programmed the test server to hold all data in memory. Thus, we can safely assume that the performance measurements are not perturbed by the time it takes for accessing an external database. We implemented the TPC-W test client as a test driver that automatically emulates a user and executes the use cases of the TPC-W benchmark. Therefore, we can also assume that the performance measurements are not affected by user think times.

The TPC-W test runs were conducted in two configurations, with normal CORBA and with DOC-CaP. For each test run, the network delay time was set to a value between 0 and 160 milliseconds. The client waiting time Twait of each test run was recorded and the result values are is shown in Figure 85. The exact test run result numbers are listed in Appendix 9.6.

As the network delay increases, so does the client waiting time Twait. Roughly one can say that in the case of normal CORBA, Twait is proportional to the network delay time. With DOC-CaP enabled, the Twait time values are significantly lower than without DOC-CaP. The speedup factors for the network delay times of 0 ms to 160 ms are shown in Figure 86.

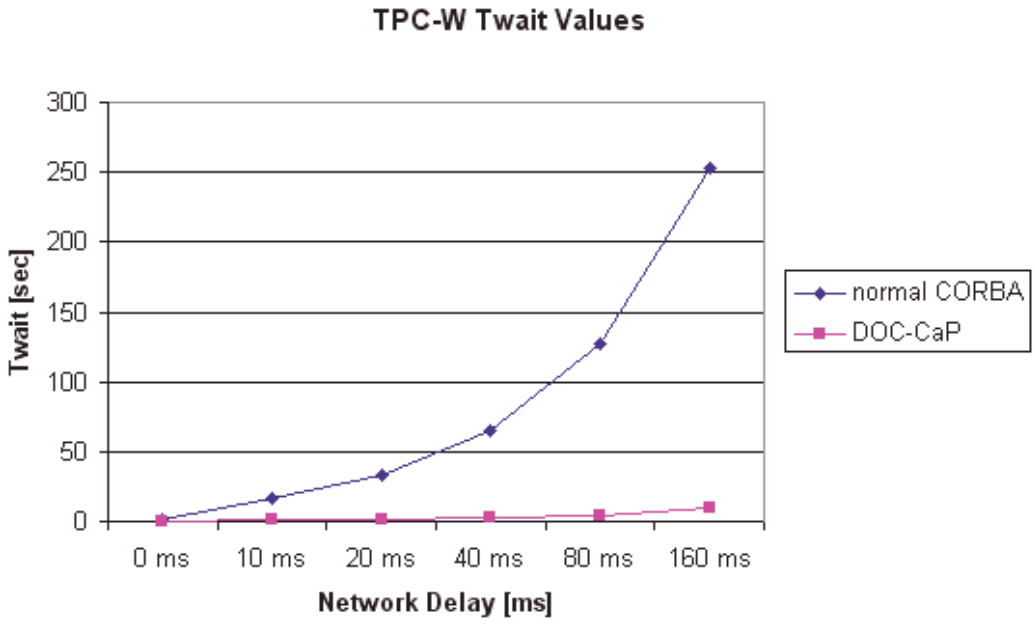


Figure 85: TPC-W Benchmark Test Run Duration (Twait)

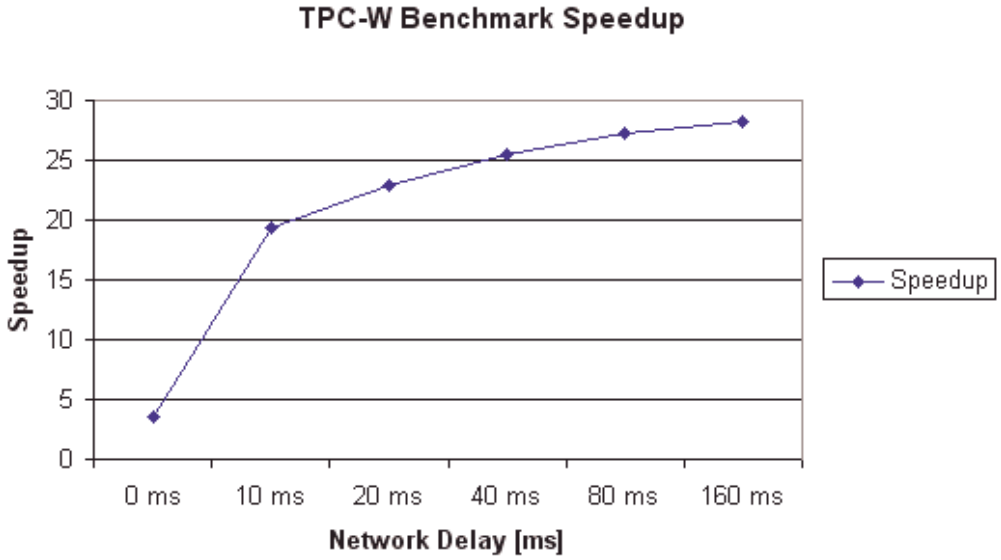


Figure 86: TPC-W Benchmark Speedup

5.3. AQUA: Automobile Quality Assurance Project

The AQUA system is part of a quality assurance system developed at an automobile company. After delivery of the first AQUA software release, users complained about the poor performance of the system on the corporate intranet WAN network. The reason for relatively low application performance was that the corporate WAN exhibited high network delays (5 to 20 milliseconds). A second release with Data Structures (see section 1.5 for an explanation of the 'Data Structures' approach) was implemented to increase user-perceived application performance.

Figure 87 shows the use case diagram for the AQUA system. The actor QAUser can create, list and edit incidents as well as access the incident overview page and the repair action list. A QAUser cannot show the QA summary page, as this is reserved for the actor QAManager. A QAManager is a QAUser and can as such execute all QAUser use cases. Additionally, a QAManager can invoke the QA summary page.

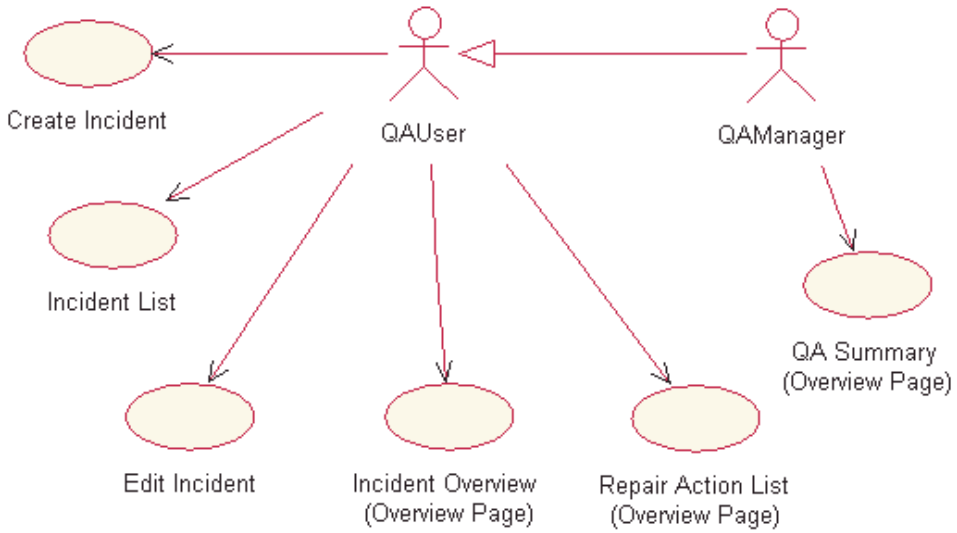


Figure 87: AQUA Use Case Diagram

The Use Cases of the AQUA system as described in the following:

- *Create Incident*. This test case starts with the Client Applet displaying a empty user interface where the user can enter data for a new incident. When the complete data is entered, the user presses the 'Save' button and the data is transferred to the Incident Server, where it is stored in the database.
- *Incident List*. The system displays the table of incidents, one incident per row. Each line has 10 columns that show the most important data items: The keyword, a short

description, the date when the incident was created, the name of the creator and so on. Moreover, each line contains a link to each of the overview pages for that incident.

- *Edit Incident*. The test case starts with the Client Applet displaying all incident data to the user. The user can then edit data that is already entered. Displaying the incident data involves the invocation of remote get-methods to retrieve the incident attributes. After the user has done editing the incident, a number of set-methods are called for transferring the data from the client to the server. Typically, only a small number of set methods are called, as the QA users make only small modifications to the incident data at a time.
- *Incident Overview*. The incident overview page contains detailed information about an incident. It lists the workers, repair personell, QA personell that are involved with this incident, a description, a short summary of the repair process, and so on.
- *Repair Action List*. The repair action list shows all repair actions that are executed in the past for this incident. Each repair action is displayed with a short description, the date of execution, a short description of the results, the person who was responsible for the repair, and so on.
- *QA Summary*. The QA summary page is intended for the QA managers to get a brief overview of an incident and its repair status. Besides QA-related incident attributes, it displays the list of workers and QA staff members who are involved in that incident. For each worker of staff member, the name, department and phone number is displayed.

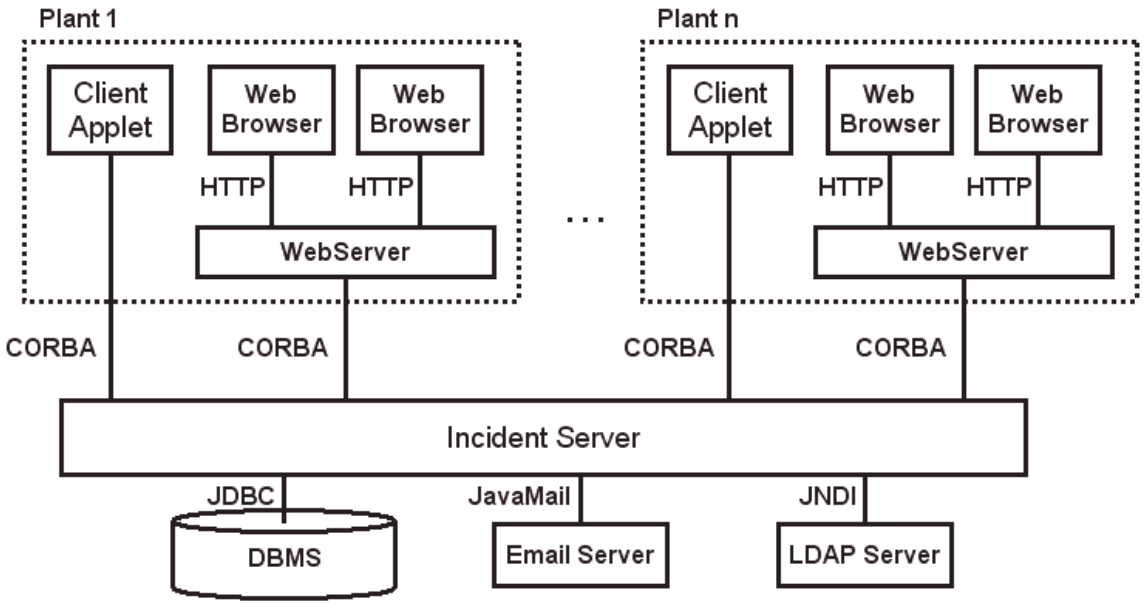


Figure 88: AQUA System Deployment

Figure 88 shows a deployment diagram of the AQUA system. The server application (Incident Server) is implemented as a Java application and manages database access, email notifications, user management and access control. The client application for creating and editing incidents is implemented with Java applet technology. The creation of the HTML overview pages is implemented with Java Servlet technology. The AQUA system is deployed on a Wide Area Network (WAN) where the Incident Server is residing in the central computing center, to which each plant of the automobile company is connected.

The Client Applets use the WAN connection to access the server functionality via its CORBA interface. Each plant owns its own WebServer that generates HTML overview pages using Java Servlet technology. The Java servlets that create the overview pages use CORBA to access the incident server and retrieve data that has to be displayed in the overview pages.

The AQUA system was designed and implemented according to the specification provided by the QA department of the automobile company. The functional requirements were specified as a GUI prototype that showed all input fields for an incident. In addition, the GUI prototype showed the fields that should be displayed in each of the overview pages. The non-functional requirements included the specification of the Java version to be used for implementation as well as the Web server and the database managements system.

The specification did not include any performance requirements. The customer decided that the performance of the system should be of minor importance at project start time. The idea was to build a first release of the system and use this release for performance tests. Then, having the performance test results, the customer wanted to decide whether the performance of the system is acceptable or performance optimizations should be built into the system.

In the first implementation of the AQUA project, an object model of the AQUA system was designed, using a straightforward approach of converting each input field into a class attribute. The object model had 18 classes: 7 classes main classes with an average of 13 attributes per class and 11 helper classes with an average of 2 attributes per class. After converting the object model into an IDL specification and generating the stub layer, the application was implemented and deployed on a Local Area Network in one plant of the automobile company.

The performance of the system was acceptable. Then the application was deployed on the company-wide intranet, using a WAN connection between the incident server and its clients. This network connection suffered from longer network delay times than the LAN connection that was used for performance assessment of the first prototype.

For increasing application performance, the ‘Data Structures’ approach described in section 1.5.2 was adopted. After re-implementing the Incident Server, the Client Applets and the client servlets to use data structures instead of fine-grained interfaces, the performance of the system was accepted by the customer.

In the following performance evaluation, we investigate how applying the DOC-CaP approach could speed up the performance of the AQUA system, as far as communication overhead is concerned. The AQUA server application, as it is deployed at the company, uses several external systems: An external database to store the application data, an LDAP server to get information about users of the system, and an email server to send notification emails to users of the system.

In our evaluation version of the AQUA server, we removed all accesses to these external systems. All application-specific data and user information is stored in memory, and the email notification service is disabled. The AQUA client is implemented as an automated test driver that executes the test cases (which are derived from the AQUA use cases, see Figure 87) of the AQUA performance evaluation without user interaction. It is important to note that the source code for this test drivers was derived directly from the AQUA client source code that is deployed at the company. No methods were added nor did we remove any method calls. This approach assures reliable performance data that reflects the performance of the real-world distributed application but does not depend on user think time or the execution speed of external systems[17].

We built a test driver that implemented the test cases ‘Create Incident’, ‘Edit Incident’, ‘Incident List’, ‘Incident Overview’, ‘Repair Action List’ and ‘QA Summary’. We compiled the test driver and the AQUA incident server in three configurations: normal CORBA with a straightforward implementation, normal CORBA with the Data Structures approach, and a DOC-CaP-based implementation. We have measured the client waiting time (*Twait*) for different network delay times ranging from 0 milliseconds to 160 milliseconds. The resulting *Twait* values were recorded and compared with each other, and a speedup factor was calculated that indicates the performance gain achieved by using the Data Structures approach and DOC-CaP. The test bed configuration is described in appendix 9.1. The test run result numbers are listed in appendix 9.7. Figure 89 and Figure 90 presents the test run speedup factors for Data Structures and DOC-CaP, for different test cases and network delay values. The speedup factor is calculated as

$$speedup = \frac{Twait\_1}{Twait\_2}$$

where *Twait_1* is the client waiting time for the straightforward configuration and *Twait_2* is the client waiting time for the Data Structures test or for the DOC-CaP test, respectively.

[17] It is important to note that the performance of the AQUA application depends on the size of the database. The more data is stored in the AQUA database, the longer it takes for e.g. a list of vehicle identification numbers to be transferred from the server to a client. Unfortunately, due to non-disclosure agreements, the exact numbers cannot be disclosed in this dissertation. We have therefore avoided to give any concrete numbers that would violate the non-disclosure agreements. However, we will present relative number where appropriate.

		Speedup Factor per Test Case (Data Structures)					
		Create Incident	Edit Incident	Incident List	Incident Overview	Repair Action List	QA Summary
Network Delay	0 ms	1,186	2,577	2,598	10,028	7,271	1,943
	5 ms	1,055	2,590	2,963	11,983	7,619	2,041
	10 ms	1,047	2,600	2,973	12,125	7,673	2,041
	20 ms	1,037	2,610	2,986	12,205	7,686	2,054
	40 ms	1,024	2,593	3,015	12,223	7,693	2,042
	80 ms	1,026	2,625	2,991	12,226	7,677	2,054
	160 ms	1,032	2,612	2,978	12,088	7,700	2,053

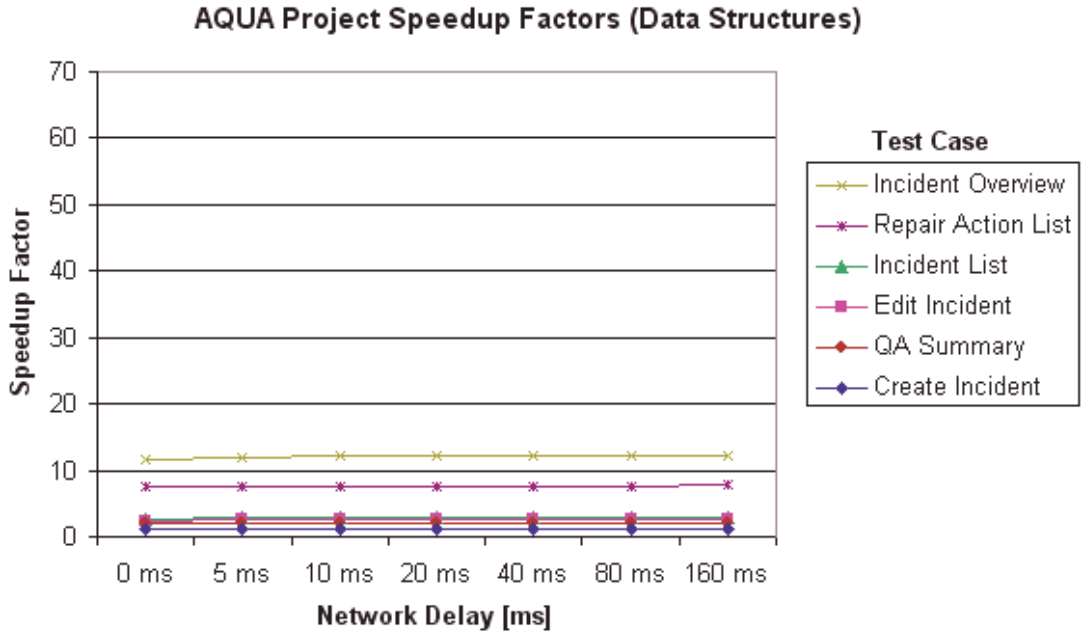


Figure 89: Speedup Factor per Test Case (Data Structures)

With Data Structures, the speedup factors for all test cases are greater than 1, which means that *Twait_2* is lower than *Twait_1* for all test cases. The 'Create Incident' test case involves many set-methods which are called one after another, so the performance gain is not high in this case. For the 'Incident Overview' and 'Repair Action List' test cases, the speedup factors are around 12 and 7, respectively. These test cases involve the invocation of many consecutive get-methods in the straightforward implementation and can benefit from using data structures instead of calling fine-grained interface methods. For the rest of the test cases, the speedup factor is around 2 to 3.

		Speedup Factor per Test Case (DOC-CaP)					
		Create Incident	Edit Incident	Incident List	Incident Overview	Repair Action List	QA Summary
Network Delay	0 ms	0,374	3,949	1,424	12,081	15,152	2,027
	5 ms	0,789	15,396	7,533	42,792	46,886	5,327
	10 ms	0,897	18,519	9,621	49,690	52,729	5,800
	20 ms	0,957	20,391	11,456	54,328	57,165	6,206
	40 ms	0,985	21,734	12,832	56,494	59,848	6,035
	80 ms	0,997	22,570	13,515	58,422	61,099	6,129
	160 ms	0,997	22,318	13,931	58,912	61,609	6,221

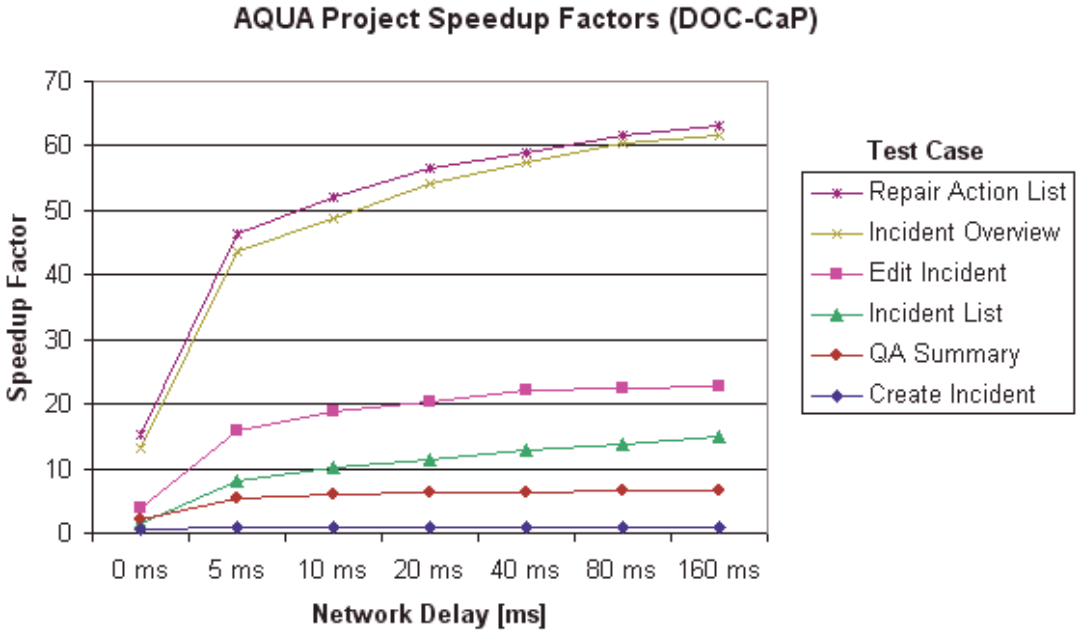


Figure 90: Speedup Factor per Test Case (DOC-CaP)

With DOC-CaP, the speedup factor for the 'Create Incident' test is less than 1, which means that *Twait_2* is greater than *Twait_1* in these cases. This is because the 'Create Incident' test involves many set-methods with side effects and are therefore not prefetchable or cachable. Consequently, each stub call involves one remote method call that has to be transferred to the server. Additionally, the DOC-CaP implementation overhead of creating and transferring a *MultiRequest*, as well as the overhead of the DOC-CaP statistic algorithm add to the client waiting time *Twait_2*.

As the AQUA test runs indicate, the benefit from using DOC-CaP is high when many consecutive remote methods are involved that do not have side effects on the server. This is the case in the 'Repair Action List' and 'Incident Overview' test cases, where remote method

calls are used to transfer data for an incident from the server to a client and all remote methods that are called are prefetchable. Moreover, the invocation pattern is repetitive, which means that the prefetching prediction algorithm can very accurately predict which methods will be called in the future. Consequently, the ‘Repair Action List’ and ‘Incident Overview’ test cases yield high speedup values of around 40 and more even for a relatively small network delay value of 5 milliseconds.

We assume that the speedup factor for the AQUA performance evaluation is determined by the number of remote method calls that can be saved by using Data Structures or DOC-CaP. To validate this assumption, we present the number of remote calls (RC) per test case and test configuration in Figure 91. The first column shows the name of the test cases. The next three columns show the number of remote (method) calls. The last two columns show the factor by which the number of remote calls is reduced by applying Data Structures or DOC-CaP instead of using the Straightforward approach.

	Remote Calls (RC)			RC Reduction Factor	
Test Case	Straight-forward	Data Structures	DOC-CaP	Data Structures	DOC-CaP
Create Incident	73	71	72	1,0	1,0
Edit Incident	47	18	2	2,6	23,5
Incident List	15	5	1	3,0	15,0
Incident Overview	184	15	3	12,3	61,3
Repair Action List	316	41	5	7,7	63,2
QA Summary	33	15	5	2,2	6,6

Figure 91: Number Of Remote Calls (RC) Per Test Case

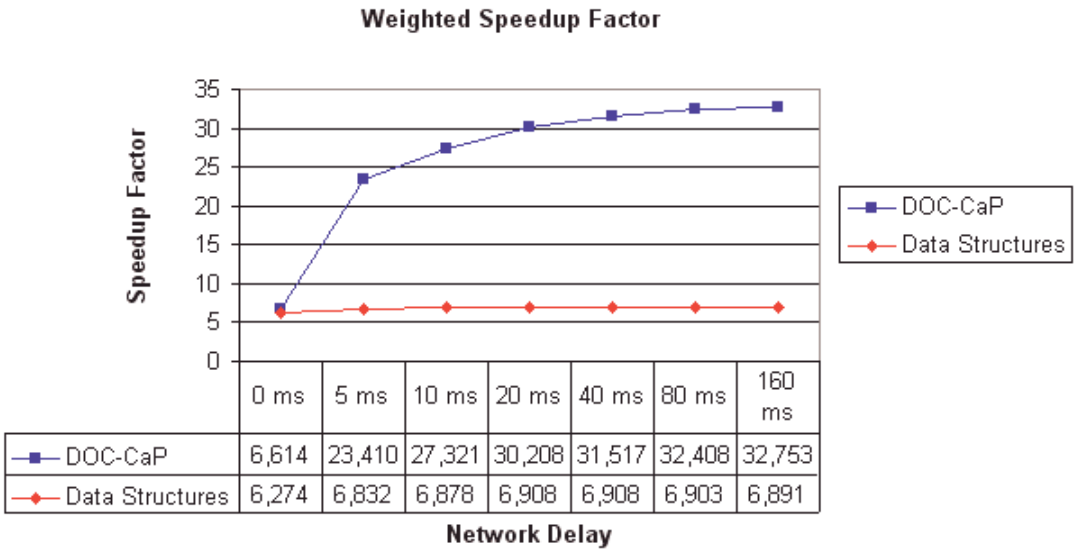
For example, The ‘Incident List’ test case involves 15 remote calls in the Straightforward configuration. With Data Structures, the number of remote calls is reduced to 5 (reduction factor is 3), whereas the DOC-CaP implementation sends 1 remote call to the server (giving a reduction factor of 15). By comparing the reduction factors in Figure 91 with the speedup factors in Figure 89 and Figure 90, one can see that the speedup factors converge against the reduction factor. Although the speedup factors do not reach the reduction factors, we can conclude that the speedup factor increases when the number of remote calls decreases.

In the AQUA system, not all use cases are executed the same number of times. For example, it is relatively seldom that a ‘Create Incident’ use case is executed, compared to the execution of a ‘Incident Overview’ use case. In order to be able to judge the benefit of using DOC-CaP in the AQUA project, we have calculated a ‘weighted’ speedup factor. It takes the client waiting time of each of the test cases and multiplies it with the number of how often that use case is executed in a certain time interval (e.g. one day). The relative weight for each use case is shown in Figure 92. Although we cannot disclose the exact numbers of how many incidents are created in a certain time interval, we can however publish relative numbers.

Use Case	Weight
Create Incident	1
Edit Incident	1,4
Incident List	64,8
Incident Overview	26,4
Repair Action List	23,4
QA Summary	13,6

Figure 92: Weight Per Use Case

If we apply the use case weight factors, we get weighted speedup factors for the Data Structures and the DOC-CaP configurations of the AQUA system as shown in Figure 93.



	0 ms	5 ms	10 ms	20 ms	40 ms	80 ms	160 ms
DOC-CaP	6,614	23,410	27,321	30,208	31,517	32,408	32,753
Data Structures	6,274	6,832	6,878	6,908	6,908	6,903	6,891

Figure 93: Weighted Speedup Factors

6. CONCLUSION AND FUTURE WORK

Distributed Object Computing (DOC) systems provide a comfortable way of implementing object-oriented communication through remote method calls. The mechanisms of transferring method parameters and method result values are transparent to the client and the server, and the developer does not need to know whether client and server are located on the same machine or across a computer network.

The goal of this dissertation was to deal with performance problems of Distributed Object Computing systems and to validate the hypothesis that caching and prefetching method result values can increase the performance of distributed applications.

After presenting current practice solutions, which require a high implementation effort, we have introduced a novel approach of automatically adding caching and prefetching functionality to the stub layer of distributed applications, thus ensuring client and server transparency with a minimal implementation effort. A CORBA-based implementation served as the basis for an experimental performance evaluation.

The results of our work are summarized in section 6.1. Remaining problems are described in section 6.2.

6.1. Results

Remote method calls are significantly slower than local method calls. Since calling a remote method involves parameter marshalling and network transfer, the client has to spend more time waiting for a remote method call than waiting for an in-process method call. We discussed current practice approaches that developers of distributed applications implement to overcome these performance losses. Current practice requires that developers must manually change the code to reduce the number of remote method calls to a minimum. The

performance speedup of distributed applications is directly related to the number of remote method calls that can be saved.

We introduced a new approach for increasing the performance of distributed applications by reducing the number of remote method calls. We have implemented this approach in the DOC-CaP Framework as a basis for adding caching and prefetching functionality to distributed applications. Since the DOC-CaP Framework is located at the stub layer of a distributed application, it is transparent to client- and server software. Thus, the DOC-CaP approach does not require a redesign or re-implementation of the distributed application, effectively keeping the implementation effort at a minimum. The only information that has to be provided to the DOC-CaP Framework is specified by declarative tags in XIDL, an extension to the interface definition language (IDL) specification. Therefore, the DOC-CaP approach is very well suited for improving the performance of newly developed applications as well as legacy applications.

For an experimental evaluation of our approach we provided an implementation of the DOC-CaP Framework in Java and CORBA and used three test applications as case studies. The first application is a distributed AddressBook sample application that stores and displays information of person objects like names, email addresses and phone numbers. For each test run, we recorded and analyzed the time for method marshalling and unmarshalling as well as the time for the network transfer of a marshalled method requests and responses. Additionally, we analyzed the overall time that a client has to wait for executing remote method calls.

The second application is TPC-W, a standard industry benchmark designed to evaluate the performance of E-Commerce systems. We implemented the benchmark business-object model in Java and CORBA. Additionally, we implemented a test driver client that emulates a user by executing the test cases defined by the TPC-W benchmark specification. The performance measurement results show the speedup factor that can be achieved by DOC-CaP, is significant.

The third application, a real-world industry application, was developed for a major automobile company as part of a quality assurance system in the company-wide intranet, which spans several subsidiaries of the company across several hundred kilometers. Our evaluation shows that DOC-CaP can reduce the overhead of remote method calls in this application by a factor of 20 to 30 in the average case and 50 to 60 in the best case, with a minimal implementation effort. In contrast, implementing the application with the data structure approach reduced the overhead of remote method calls by a factor of 6 in the average case and 12 in the best case, but only with an extraordinary high implementation effort . This test demonstrates the viability of our approach for distributed legacy applications.

6.2. Future Work

While the DOC-CaP Framework presented in this dissertation can speed up distributed application performance significantly and is applicable to a wide range of distributed applications, there are still problems that need to be addressed by further work.

The prefetching algorithm presented in this dissertation works well for predicting future method calls in Distributed Object Computing applications. The prefetching algorithm is based upon method call probabilities and a fixed threshold value. If the probability of a method call is greater than or equal to the threshold value, the method will be prefetched. If the probability of a method call is below the threshold value, the method will not be prefetched.
Once a method has been prefetched, the DOC-CaP system keeps track of how often this method is called later on, thus calculating the benefit of the prefetch. The cost of the prefetch can be derived from the time it takes to transfer the request and response of this method over the network plus the time it takes for the method implementation to execute.
If a client calls the method not at all, the benefit is zero and time has obviously been wasted by prefetching this method. If the prefetched method is called later on, the benefit can be calculated as the saved network delay time.
Based on the cost/benefit information, the prefetch threshold could be dynamically adjusted and we assume that by doing so, the prefetching accuracy could be improved.

The cache consistency approach used by the DOC-CaP framework is based on an expiration model with client invalidation. Each cache item is invalidated after its expiration time has elapsed. Additionally, whenever a client invokes a method that has side effects, the client cache is invalidated. In some cases it might actually be sufficient to invalidate only a small portion of the client cache, instead of the whole cache. For methods that have side effects only to attributes in the class in which the methods are defined, it would be sufficient to invalidate only the portion of the client cache that relates to instances of this class. We assume that such a finer-grained cache consistency would be lead to even higher speedup factors.

The problem of deciding whether a method can potentially have side effects is left to the developer of a distributed application. The developer uses the IDL specification to indicate for each method whether it has side effects or not. It is crucial for application correctness that the method side effect information given in the IDL description is correct.
For large-scale systems, keeping track of side effect information in the IDL specification is an additional task in system development and has to be carefully integrated in the software development process. Moreover, by providing side effect information in IDL, implementation details are disclosed in the interface specification which violates the principle of information hiding.

7. REFERENCES

[01] Informed Prefetching and Caching. Patterson, R.H. Gibson, G.A., Ginting, E., Stodolsky, D. and Zelenka, J. Proc. of the 15th Symposium of Operating Systems Principles, Copper Mountain Resort, CO, December 3-6, 1995, pp. 79-95.

[02] Informed Prefetching and Caching. Hugo Patterson. Carnegie Mellon Ph.D. Dissertation CMU-CS-97-204: December 1997.

[03] Object Management Group: Unified Modeling Language (UML), v1.4. (2001) formal/01-09-67.

[04] Andrew S. Tanenbaum. Computer Networks, Third edition. 1996 Prentice Hall.

[05] J.D. Day & H. Zimmermann, The OSI Reference Model, Proceedings of the IEEE, vol. 71, pp. 1334–1340, December 1983.

[06] International Organization for Standardization (ISO), http://www.iso.org

[07] Andrew D. Birrell, Bruce Jay Nelson. Implementing Remote Procedure Calls. ACM Transactions on Computer Systems, Volume 2 , Issue 1 (February 1984), pp 39-59.

[08] Frank E., III Redmond, DCOM: Microsoft Distributed Component Object Model, Hungry Minds Inc, 1997

[09] Object Management Group, Inc., CORBA Basics, http://www.omg.org/gettingstarted/corbafaq.htm

[10] Object Management Group, The Common Object Request Broker: Architecture and Specification 3.0, (2004) formal/04-03-01

[11] Object Management Group, CORBA Value Type Semantics, (2002) formal/02-06-09

[12] Object Management Group, CORBA Messaging, (2004) formal/04-03-09

[13] Object Management Group, Common Object Request Broker Portable Object Adapter, (2004), formal/04-03-17

[14] JacORB, The free Java implementation of the OMG's CORBA standard, http://www.jacorb.org

[15] JBoss The Professional Open Source Company, JBoss Application Server, http://www.jboss.com/products/jbossas

[16] SUN Microsystems, Inc., Java Technology, http://java.sun.com

[17] National Institute of Standards and Technology, NIST Net Home Page, http://snad.ncsl.nist.gov/itg/nistnet/

[18] Douglas C. Schmidt, Andy Gokhale, Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks, Proceedings of the IEEE 17th International Conference on Distributed Systems (ICDCS 97), May 27-30, 1997, Baltimore, USA.

[19] Michi Henning, Steve Vinoski, Advanced CORBA Programming with C++, Addison-Wesley Professional, 1999

[20] Russel Sandberg, David Goldberg, Steve Kleimann, Dan Walsh, and Bob Lyon: Design and Implementation of the Sun Network File System. In Proc. Summer 1985 USENIX Conference, pp. 119-130, Portland, OR, June 1985

[21] Michael N. Nelson, Brent B. Welch and John K. Ousterhout. Caching in the Sprite Network File System. ACM Transactions on Computer Systems (6)1 pp. 134-154, February 1988.

[22] Roy T. Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk Nielsen, Paul Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Internet Engineering Task Force, Network Working Group, June 1999. IETF RFC 2616.

[23] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM 21,7, July 1978, pp 558-565.

[24] Dan Duchamp. Prefetching hyperlinks. In Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS '99), Boulder, CO, October 1999.

[25] D. Kristol, L. Montulli, Network Working Group Request For Comments 2109, HTTP State Management Mechanism, February 1997

[26] Ken A. L. Coar, D. R. T. Robinson, The WWW Common Gateway Interface Version 1.1, Internet-Draft draft-coar-cgi-v11-03.html, http://cgi-spec.golux.com/draft-coar-cgi-v11-03.html

[27] Bruce Eckel, Thinking in C++ Volume 1, 2000, Prentice Hall, Upper Saddle River, New Jersey 07458

[28] Bjarne Stroustrup, The C++ Programming Language, 3rd edition, Addison-Wesley 1997

[29] Kwok Cheung Yeung, Paul H. J. Kelly, Optimizing Java RMI Programs by Communication Restructuring, Middleware 2003, Rio De Janeiro

[30] Intel Corporation, Intel Processor Spec Finder, http://processorfinder.intel.com/scripts/help3.asp

[31] Alan Jay Smith, Cache memories, ACM Computing Surveys, vol. 14, pp. 473-530, 1982

[32] Jeffrey Scott Vitter and P. Krishnan, Optimal Prefetching via Data Compression, Proceedings of the Annual IEEE Symposium on Foundations Of Computer Science (October 1991)

[33] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical Prefetching via Data Compression. In Proc. 1993 ACM-SIGMOD Conference on Management of Data, pages 257--266, May 1993.

[34] J. Ziv, A. Lempel, Compression of Individual Sequences via Variable-Rate Coding, IEEE Transactions on Information theory 24 (September 1978), pp. 530-536

[35] The Transaction Processing Performance Council, http://www.tpc.org

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, August 1994.

[37] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, B. B. Welch, The Sprite Network Operating System, Computer Magazine of the Computer Group News of the IEEE Computer Group Society; ACM CR 8905-0314, vol. 21, 1988.

[38] James Gwertzman and Margo I. Seltzer, World Wide Web Cache Consistency, USENIX Annual Technical Conference, pp. 141-152, 1996.

[39] A.H. Dutoit, O. Creighton, G. Klinker, R. Kobylinski, C. Vilsmeier, B. Bruegge Architectural Issues In Mobile Augmented Reality Systems: A Prototyping Case Study, The Eighth Asian Pacific Conference on Software Engineering (APSEC 2001) Macau SAR, China, December 4-7, 2001.

[40] G. Klinker, O. Creighton, A.H. Dutoit, R. Kobylinski, C. Vilsmeier, B. Bruegge Augmented maintenance of powerplants: A prototyping case study of a mobile AR system The Second IEEE and ACM International Symposium on Augmented Reality. New York, NY, October 29-30, 2001.

[41] B.Bruegge, C.Vilsmeier, Reducing CORBA Call Latency By Caching And Prefetching, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, 16-20 June 2003, MIDDLEWARE 2003 Work-In-Progress Paper, (IEEE Distributed Systems Online http://dsonline.computer.org/0306/f/brug.htm).

[42] Christoph Vilsmeier, Caching CORBA, Proceedings of the ACIS Fourth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'03). pp. 139-145 October 16-18, 2003, Lübeck, Germany. ACIS 2003, ISBN 0-9700776-7-X.

[43] Stefan Podlipnig and Laszlo Boeszoermenyi. A survey of Web cache replacement strategies. ACM Computing Surveys, 35(4):374–398, December 2003.

[44] A. Dan and D. Towsley, An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes, in Proceedings of the ACM SIGMETRICS, Denver, CO, 1990.

[45] I. Tatarinov, A. Rousskov, and V. Soloviev. Static Caching in Web Servers. In Proc. Sixth IEEE Intl. Conf. on Computer Communications and Networks (IC3N'97), Las Vegas, Nevada, USA, September 1997

[46] K. Chinen and S. Yamaguchi. An Interactive Prefetching Proxy Server for Improvement of WWW Latency. In Proceedings of The Seventh Annual Conference Of The Internet Society, INET 97, June 1997.

[47] Lazlo A. Belady. A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal, 5(2):79–101, 1966.

[48] Michi Henning, A New Approach to Object-Oriented Middleware, IEEE Internet Computing, vol. 08, no. 1, pp. 66-75, January/February, 2004.

[49] Thomas M. Kroeger and Darrell D. E. Long. The case for efficient file access pattern modeling. In Proceedings of the 1996 USENIX Technical Conference, January 1996.

[50] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification — Second Edition. Addison-Wesley, 2000.

[51] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies . Proceedings of ICS'90, Amsterdam, The Netherlands, 1:342--353, June 1990

[52] T. J. Mowbray, R. C. Malveau, CORBA Design Patterns, John Wiley & Sons, Inc., 1997, ISBN 0-471-15882-8.

[53] T. M. Kroeger and D. D. E. Long. Predicting file system actions from prior events. In Proceedings of the USENIX 1996 Annual Technical Conference, pages 319--328, January 1996.

[54] Emmanuel Cecchet, Julie Marguerite, Willy Zwaenepoel, Performance and scalability of EJB applications, Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 246-261, Seattle, Washington, USA, 2002.

[55] L.G.DeMichiel, L.U.Yalcinalp, S.Krishnan, Enterprise JavaBeans Specification Version 2.0., Sun Microsystems, Inc., Final release edn. (2001)

[56] Object Management Group: CORBA Portable Interceptor Specification. (2001) ptc/01-03-04, formal/02-05-18

[57] M. Aleksy, S.Kuhlins, Using Value Types to Improve Access to CORBA Objects, Proceedings of the ACIS 2nd International Conference on Software Engineering,

Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'01), S. 841-847, Mt. Pleasant, MI, USA, ACIS, 2001.

[58] Christoph Pohl and Alexander Schill, Client-side component caching, In 4th International Conference on Distributed Applications and Interoperable Systems (DAIS 2003), volume 2893 of Lecture Notes in Computer Science, pages 141-152, Paris, France, 19-21 November 2003. IFIP WG 6.1, Springer.

[59] Sun Microsystems Inc., Java 2 Enterprise Edition (J2EE), http://java.sun.com/j2ee

[60] Sun Microsystems Inc., Java Server Pages (JSP), http://java.sun.com/products/jsp

[61] Daniel Pfeifer, Hannes Jakschitsch, Method-Based Caching in Multi-Tiered Server Applications, In the Proceedings of OTM Confererated International Conferences CoopIS, DOA, and ODBASE 2003;Distributed Objects and Applications (DOA) 2003, LNCS 2888, pp.1312-1332, 2003. Catania, Sicily, Italy, November 2003.

[62] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In Proceedings of Middleware '00, pages 1-23, April 2000.

[63] R. Kordale, M. Ahamad and M. Devarakonda. Object Caching in a CORBA Compliant System. Proc. of Second Conference on Object-Oriented Technologies and Systems. June 1996.

[64] M. Aleksy, M. Schader, Design Alternatives in the Definition of IDL Interfaces, Proceedings of the 6th World Multiconference on Systems, Cybernetics and Informatics (SCI 2002). IIIS, 2002.

[65] The Transaction Processing Performance Council, TPC Banchmark W (Web Commerce) Specification, Version 1.8, Feb 19, 2002 http://www.tpc.org/tpc-w/default.asp

[66] Raj Jain, The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling, John Wiley & Sons, Inc, 1991

[67] WinPcap: The Windows Packet Capture Library, http://www.winpcap.org

[68] Windump: tcpdump for Windows, http://www.winpcap.org/windump/

[69] Jeffrey Richter, Advanced Windows, Third Edition, Microsoft Press, 1997

[70] Kay A Robbins, Steve Robbins, UNIX Systems Programming: Communication, Concurrency, and Threads, Prentice Hall PTR, 2003

[71] Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, 2nd Edition, Prentice Hall, 1988

[72] Mary Campione, Kathy Walrath, Alison Huml, The Java Tutorial Continued: The Rest of the JDK, Addison-Wesley Professional, 1998

[73] Mark Carson, Darrin Santay, NIST Net - A Linux-based Network Emulation Tool, ACM SIGCOMM Computer Communications Review Volume 33, Number 3: July 2003

[74] Gerald Brose, JacORB - a Java Object Request Broker, Technical Report B-97-02, Freie Universität Berlin, April 1997

[75] Object Management Group, CORBA Abstract Interface Semantics, formal/02-06-10

[76] Themistoklis Palpanas, Alberto Mendelzon, Web Prefetching Using Partial Match Prediction, Proceedings of the 4th International Web Caching Workshop, San Diego, California, March 31 - April 2, 1999

[77] I. Zukerman, D. W. Albrecht, A. E. Nicholson, Predicting users' requests on the WWW, Proceedings of the Seventh International Conference on User Modeling (UM99), 1999

[78] T. Berners-Lee, R. Fielding, H. Frystyk, Hypertext Transfer Protocol - HTTP/1.0, RFC 1945, HTTP Working Group, May 1996

[79] Balachander Krishnamurthy, Jeffrey C. Mogul, David M. Kristol, Key Differences between HTTP/1.0 and HTTP/1.1, Proceedings of the Eight International World Wide Web Conference, May 1999

[80] Pei Cao, Chengjie Liu, Maintaining Strong Cache Consistency in the World Wide Web, IEEE Transactions on Computers, vol. 47, pp.445-457, 1998

[81] Mikhail Mikhailov, Craig E. Wills: Evaluating a new approach to strong web cache consistency with snapshots of collected content. Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20-24 May 2003

[82] Balachander Krishnamurthy, Craig E. Willis, Piggyback Cache Validation for Proxy Caches in the World-Wide Web, Proceedings of the 1997 Web Cache Workshop, 1997

[83] Sheng Liang, Java Native Interface: Programmer's Guide and Specification, Sun Microsystems, Addison Wesley, 1999

[84] Information Sciences Institute, University of Southern California, Transmission Control Protocol, DARPA Internet Program Protocol Specification , September 1981, RFC 793

[85] Natalija Krivokapic, Synchronization in a Distributed Object System, Datenbanksysteme in Buro, Technik und Wissenschaft, pp. 332-341, 1997

[86] Object Management Group OMG, CORBAservices: Common Object Services Specification (COSS), OMG, second edition, March 1996

8. LIST OF FIGURES

9. APPENDIX

9.1. Test Bed Configuration

In this section we describe the test bed that we used for evaluating the performance of the test applications presented in our work. We describe the hardware and software configuration as well as the performance measurement setup.

9.1.1. Hardware and Software Configuration

The test bed for all performance tests conducted in our work consists of three computers connected by an Ethernet network: A client, a server and a router, which acts as a gateway between client and server. The testbed setup is shown in Figure 94.

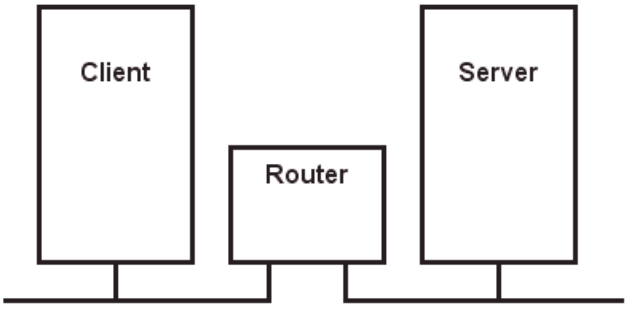


Figure 94: Testbed Hardware Setup

Client
The client machine is a Laptop (Dell Inspiron 8100, 384 MB, Intel Pentium III mobile 800 MHz) running MS Windows XP, Service Pack 1. The network card is a 10/100MBit Ethernet Card (FASTline PCMCIA 10/100 MBit Fast-Ethernetadapter).

Server
The server machine is PC (265 MB, Intel Pentium II 600 MHz) running MS Windows 2000 Professional, Service Pack 4. The network card is a 10/100MBit Ethernet Card (SMC EtherPower II 10/100 PCI Ethernet Adapter).

Router
The router is a Linux-based PC configured as a TCP/IP router and running the 'NIST Net' TCP/IP network emulator (see [17] and [73]). The NIST Net package allows emulating packet delay and bandwidth limitation. The router machine hosts two 10/100MBit network cards (3Com 3c905CX-TX-M PCI)

TCP/IP Trace Utility
The WinPcap/Windump package (see [67] and [68]) was used to capture network packets at the client and server machine. We installed WinPcap/Windump on both the client machine and the server machine and configured the utility so that it resides between the network transport layer and the application program and captures any ingoing or outgoing TCP/IP packets and stores them in a capture file. The capture file is analysed later (off-line) and network roundtrip times are extracted. Dedicated test runs showed that the capturing process does not slow down network traffic measurably.

CORBA implementation
The CORBA system installed at the client and server machine is the JacORB 2.0 [14] CORBA implementation, which was developed at Freie Universität Berlin and is distributed under an open-source license, see [74] for a description of JacORB. The product and its source code are free, and it is widely used in the industry, in research projects and in open source projects, for example in the JBoss [14] application server. JacORB 2.0 provides a CORBA 2.3 compliant IDL compiler to produce client- and server-stubs, Objects By Value [11] and Asynchronous Messaging Interface [12].

Programming Language
The client and server software as well as the JacORB CORBA implementation is written in Java [15]. The Java Runtime Environment for executing the test runs is Java 1.4.0, Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.0-b92), Java HotSpot(TM) Client VM (build 1.4.0-b92, mixed mode).

CPU and network load
All background processes that are not part of the operating systems of the client, server or router machine are disabled. The network connection is not used by any other computers, which is the basis for reproducible test results.

9.1.2. Performance Measurement Setup

The primary goal of the performance measurements presented in the following sections is to evaluate the performance of CORBA-based and DOC-CaP-based distributed applications under different network delay values, different network bandwidths, different amounts of data to be transferred, and so on. The most important performance criteria for the user of a distributed application is the perceived application performance, which is - among other factors - affected by the time that is needed by a client stub to perform a remote method call. This time, which is called *Twait* throughout this dissertation, is the time that elapses between the invocation of a stub method and the return of that stub method.

Since our test applications are implemented in Java and Java does not provide a high-resolution timer, we have implemented a Java Extension in C (using the Java Native Interface JNI, see [83]) that provides a high-resolution timer using the Windows *QueryPerformanceCounter* and *QueryPerformanceFrequency* system calls. The *QueryPerformanceCounter* system call returns the current value of the high-resolution clock and the *QueryPerformanceFrequency* system call returns the frequency with which the clock is incremented. The frequency depends on the installed hardware and is 3579545 on the client machine used in our test runs, resulting in a timer resolution of around 0.3 microseconds. The *QueryPerformanceCounter* system call itself takes about 0.02 milliseconds.

The *Twait* values for stub calls are measured by placing a high-resolution timer query directly before a stub call and another timer query immediately after the stub call. The difference of both values is taken as the execution time of the stub call, which is *Twait*.

In our test runs, the client waiting time *Twait* is composed of several factors. The two most important factors are the network time *Tnet*, which is the time for transferring remote method requests and responses between client and server, and the marshalling time *Tmar*, which is the time for marshalling and unmarshalling remote method requests and responses at the client and server side.

When a client calls a stub method, the stub first converts all method parameters into a byte array and creates a remote method request. Then it sends the method request, along with the marshalled method parameters, across the network to the server. The basic control flow of a remote method call is discussed in section 1.2 and shown again in Figure 95, annotated with time markers. Since the JacORB CORBA implementation used in our test runs uses TCP sockets for network communication, a TCP/IP connection between client and server is used to transfer the method request. For a description of TCP/IP network communication details we refer to [84]. For the understanding of our test measurements, it is important to note that the TCP/IP protocol does not send the method request to the server at once (as presented in

Figure 95). Simplified, it splits the byte array into small TCP packets and transfers them over the network one by one. After a certain number of packets have been sent, the client TCP waits for an acknowledgement from the server. Having received the acknowledgement, it continues sending TCP packets. This procedure is repeated until all data is sent to the server. The same procedure is repeated for the remote method response, which is again split into several TCP packets and sent back to the client, the server TCP waiting for acknowledgement messages from the client. The size of the TCP packets depends on the configuration of the operating system and is 1500 bytes in our test runs. The number of packets that can be sent before waiting for an acknowledgement message, the so-called TCP window size, is up to the TCP/IP implementation. Most implementations – among them the TCP/IP implementation used in our test runs – dynamically adjust the TCP window size to maximize network throughput.

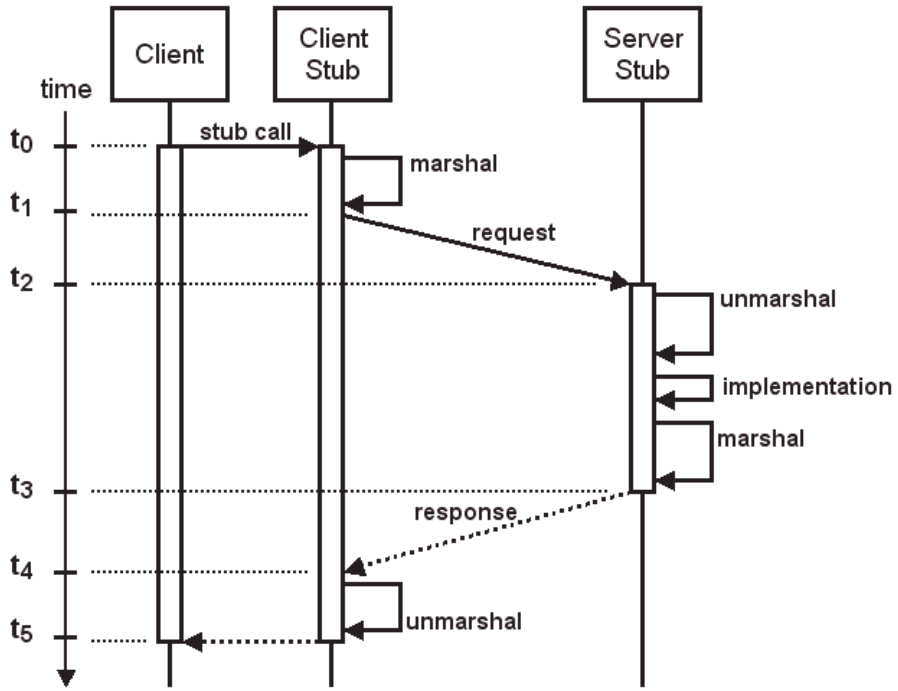


Figure 95: Performance Measurement Timestamps

We have measured the network time of a remote method call by installing the WinPcap/Windump package (version 3.8.3, see [67] and [68]) on both client and server machine and tracing all network traffic that belonged to the application under test. For measuring the network time, we analyzed the Windump traces on the client machine and calculated the time between the first outgoing TCP packet of the method request (t_1 in Figure 95) and the last incoming TCP packet of the method response (t_4 in Figure 95). This time is the network time T_{net} plus the server-side marshalling time plus the execution time of the

server-side method implementation. Since the server-side method implementations return pre-computed values, the execution time of the server-side method implementation can be neglected for all test runs and the server-side marshalling time can be calculated as the time that elapses between the last incoming TCP packet of the method request (t_2 in Figure 95) and the fist outgoing TPC packet of the method response (t_3 in Figure 95).

The formulas for calculating *Twait*, *Tnet* and *Tmar* are presented in the following:

$$Twait = (t_5 - t_0)$$
$$Tnet = (t_4 - t_1) - (t_3 - t_2)$$
$$Tmar = Twait - Tnet = (t_5 - t_0) - (t_4 - t_1) + (t_3 - t_2)$$

The test runs result numbers presented in the following sections are average numbers. Each test run was executed several times (short-running test runs were executed more often than long-running test runs) and the result numbers of all test runs were logged in trace files. Later we analyzed these trace files (using off-line trace analyzer scripts) and extracted the timing information shown in Figure 95. We cut the top 10 percent and the bottom 10 percent of all numbers and calculated the average values of the rest of the performance numbers. By applying this procedure, outliers that result from java garbage collection, operating system background activities, and so on, are ignored. This principle of ignoring outliers is described in [66].

As described in section 9.1.1 the client machine is not directly connected to the server machine. Instead, the TCP/IP traffic is going through a router. We have installed the NIST Net package (version 2.0.12) on the router, which allows for emulating network delay and bandwidth limitations in a controlled manner. It is important to note that the existence of the NIST Net network emulation is transparent to both the client and the server.

9.2. Distributed Object Computing Performance

The purpose of the test runs 'Distributed Object Computing Performance' is to show how network parameters and marshalling time affect the client waiting time. The test scenarios are described in section 1.3: The test server implements one single method `getName()`. In all test runs, the implementation of `getName()` returns a pre-computed value. The implementation time `Timpl` is below a measurable value and can be neglected.

Varying Network Delays

The first series of test runs shows how the client waiting time `Twait` of a remote method call depends on the network delay time `Tdelay`. The test scenario for this test run is described in section 1.3. The test results are shown in Table 1, Table 2 and Table 3, a graphical representation of the `Twait` values is presented in Figure 96: The x-axis represents the length of the `getName()` result value, from 0 to 32000 characters. The y-axis represents the time for `Twait` in milliseconds. The diagram contains 6 data series, one for each configured network delay time. In this test run, no bandwidth limitation is configured on the router machine, which means that the full speed of the network could be utilized.

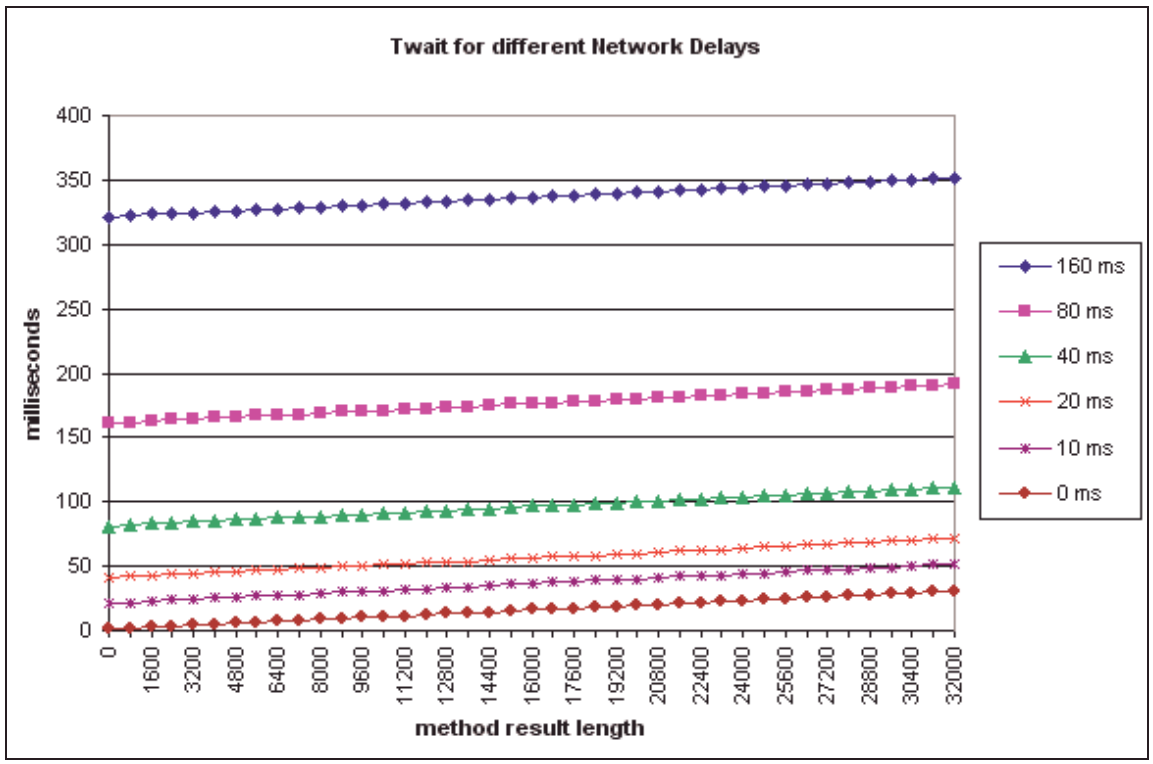


Figure 96: Twait For Different Network Delays

As one can see from the test results, *Tmar* and *Tnet* grow as the method result value grows. As a consequence, the client waiting time *Twait* depends on the length of the method result value that has to be transferred over the network. As we can see from the test result values, even if the method result length is 0, the network time *Tnet* and the marshalling time *Tmar* are still above zero. This is because even when there is no method result value to transmit, there are still *Request* and *Response* objects to be created, marshalled and transferred over the network. A discussion of the test result numbers can be found in section 1.3.

		Network Delay Time [ms]					
		160 ms	80 ms	40 ms	20 ms	10 ms	0 ms
Method Result Length	0	321,43	161,06	81,25	41,17	21,20	1,14
	800	322,19	161,94	82,09	41,95	22,00	1,96
	1600	323,21	163,04	83,08	43,14	23,13	3,00
	2400	323,91	163,65	83,76	43,77	23,77	3,70
	3200	324,64	164,44	84,59	44,54	24,57	4,45
	4000	325,35	165,11	85,20	45,22	25,18	5,12
	4800	326,08	165,86	86,01	45,97	26,00	5,90
	5600	326,84	166,68	86,72	46,56	26,66	6,53
	6400	327,59	167,36	87,48	47,43	27,43	7,35
	7200	328,20	168,05	88,22	48,08	28,09	7,98
	8000	329,08	168,76	88,96	48,96	28,95	8,78
	8800	329,80	169,64	89,82	49,72	29,73	9,56
	9600	330,58	170,34	90,39	50,35	30,27	10,22
	10400	331,33	171,00	91,26	51,10	31,11	11,06
	11200	331,99	171,73	91,96	51,85	31,87	11,33
	12000	332,70	172,59	92,71	52,67	32,66	12,22
	12800	333,54	173,31	93,35	53,35	33,35	12,95
	13600	334,30	174,01	94,41	53,96	34,06	13,62
	14400	334,92	174,71	94,84	54,77	34,77	14,38
	15200	335,66	175,76	95,80	55,66	35,82	15,37
	16000	336,41	176,40	96,58	56,33	36,45	16,02
	16800	337,37	177,17	97,33	57,10	37,30	16,82
	17600	338,21	178,03	98,01	57,87	38,15	17,36
	18400	338,78	178,57	98,82	58,41	39,06	18,17
	19200	339,45	179,40	99,52	59,65	39,22	18,92
	20000	340,26	180,15	100,37	59,96	39,82	19,49
	20800	341,02	180,77	101,11	60,93	40,81	20,42
	21600	341,83	181,68	101,69	61,75	42,17	21,12
	22400	342,60	182,45	102,55	62,47	42,10	22,04
	23200	343,27	182,98	103,19	63,11	43,10	22,60
	24000	344,06	183,79	103,99	63,98	43,93	23,46
	24800	344,89	184,62	104,76	64,75	44,64	24,20
	25600	345,61	185,34	105,41	65,41	45,74	24,79
	26400	346,34	186,15	106,22	66,19	46,56	25,65
	27200	346,92	186,79	106,94	66,85	47,20	26,44
	28000	347,77	187,65	107,74	67,76	47,77	27,07
	28800	348,62	188,74	108,57	68,48	48,71	27,83
	29600	349,38	189,10	109,29	69,36	49,43	28,58
	30400	350,05	189,98	109,92	69,99	50,14	29,39
	31200	350,84	190,73	110,66	70,96	50,95	29,93
	32000	351,56	191,34	111,42	71,67	51,69	30,63

Table 1: Twait For Different Network Delays

		Network Delay Time [ms]					
		160 ms	80 ms	40 ms	20 ms	10 ms	0 ms
Method Result Length	0	320,62	160,36	80,58	40,52	20,53	0,39
	800	321,38	161,18	81,29	41,22	21,22	1,13
	1600	322,13	161,98	82,03	42,11	22,09	1,90
	2400	322,67	162,49	82,63	42,63	22,60	2,47
	3200	323,31	163,15	83,35	43,29	23,26	3,10
	4000	323,88	163,69	83,83	43,80	23,77	3,67
	4800	324,49	164,35	84,52	44,48	24,48	4,30
	5600	325,15	164,97	85,08	44,98	24,98	4,87
	6400	325,72	165,53	85,74	45,72	25,66	5,53
	7200	326,29	166,17	86,37	46,24	26,26	6,08
	8000	326,89	166,76	86,91	46,90	26,89	6,72
	8800	327,64	167,43	87,65	47,57	27,56	7,37
	9600	328,18	168,01	88,16	48,10	28,04	7,93
	10400	328,75	168,65	88,84	48,80	28,76	8,57
	11200	329,41	169,23	89,40	49,30	29,21	9,00
	12000	330,03	169,87	90,06	50,01	30,00	9,70
	12800	330,62	170,50	90,58	50,59	30,53	10,20
	13600	331,29	171,09	91,26	51,20	31,15	10,87
	14400	331,79	171,67	91,85	51,83	31,77	11,47
	15200	332,55	172,40	92,45	52,38	32,45	12,12
	16000	333,08	172,87	93,07	53,01	33,00	12,63
	16800	333,75	173,57	93,76	53,57	33,62	13,32
	17600	334,33	174,16	94,31	54,34	34,40	13,92
	18400	334,86	174,75	94,97	54,89	34,78	14,47
	19200	335,56	175,51	95,59	55,57	35,38	15,16
	20000	336,17	175,95	96,17	56,03	35,92	15,66
	20800	336,79	176,54	96,76	56,82	36,68	16,32
	21600	337,29	177,22	97,34	57,38	37,33	16,91
	22400	338,02	177,87	98,02	58,02	37,71	17,52
	23200	338,52	178,38	98,56	58,50	38,32	18,06
	24000	339,16	179,01	99,23	59,21	39,14	18,74
	24800	339,83	179,73	99,82	59,86	39,71	19,36
	25600	340,49	180,24	100,41	60,42	40,43	19,92
	26400	341,06	180,96	101,09	61,08	40,95	20,55
	27200	341,63	181,46	101,66	61,59	41,58	21,12
	28000	342,34	182,14	102,30	62,32	42,24	21,75
	28800	342,90	182,71	102,80	62,81	42,78	22,30
	29600	343,52	183,38	103,46	63,45	43,46	22,90
	30400	344,07	183,91	104,09	64,02	43,99	23,50
	31200	344,73	184,55	104,67	64,69	44,68	24,07
	32000	345,32	185,13	105,29	65,24	45,21	24,64

Table 2: Tnet For Different Network Delays

		Network Delay Time [ms]					
		160 ms	80 ms	40 ms	20 ms	10 ms	0 ms
Method Result Length	0	0,82	0,70	0,67	0,65	0,68	0,74
	800	0,82	0,77	0,80	0,73	0,77	0,83
	1600	1,08	1,06	1,06	1,04	1,04	1,10
	2400	1,24	1,16	1,13	1,14	1,17	1,23
	3200	1,33	1,29	1,24	1,24	1,31	1,35
	4000	1,47	1,42	1,37	1,42	1,42	1,45
	4800	1,60	1,51	1,49	1,49	1,53	1,59
	5600	1,69	1,70	1,64	1,58	1,68	1,66
	6400	1,88	1,83	1,74	1,72	1,77	1,82
	7200	1,91	1,88	1,84	1,84	1,83	1,90
	8000	2,19	2,01	2,05	2,06	2,05	2,06
	8800	2,15	2,20	2,17	2,15	2,18	2,19
	9600	2,39	2,33	2,24	2,25	2,23	2,29
	10400	2,58	2,35	2,43	2,31	2,35	2,50
	11200	2,58	2,51	2,56	2,56	2,66	2,33
	12000	2,67	2,72	2,64	2,66	2,66	2,52
	12800	2,92	2,80	2,77	2,77	2,82	2,75
	13600	3,01	2,92	3,14	2,76	2,91	2,75
	14400	3,13	3,04	2,98	2,94	3,01	2,91
	15200	3,11	3,36	3,35	3,28	3,37	3,25
	16000	3,33	3,54	3,51	3,32	3,46	3,39
	16800	3,62	3,60	3,57	3,53	3,68	3,50
	17600	3,88	3,87	3,70	3,54	3,75	3,43
	18400	3,91	3,82	3,85	3,51	4,28	3,70
	19200	3,89	3,88	3,92	4,09	3,84	3,76
	20000	4,09	4,20	4,20	3,93	3,90	3,83
	20800	4,23	4,23	4,35	4,11	4,13	4,10
	21600	4,54	4,46	4,35	4,37	4,84	4,22
	22400	4,58	4,58	4,53	4,45	4,38	4,52
	23200	4,76	4,60	4,63	4,61	4,78	4,54
	24000	4,90	4,79	4,76	4,77	4,79	4,72
	24800	5,06	4,89	4,94	4,89	4,92	4,84
	25600	5,12	5,10	5,00	4,99	5,31	4,87
	26400	5,28	5,20	5,13	5,12	5,62	5,10
	27200	5,29	5,33	5,29	5,26	5,62	5,32
	28000	5,43	5,51	5,43	5,43	5,53	5,33
	28800	5,72	6,03	5,77	5,67	5,93	5,53
	29600	5,87	5,72	5,83	5,91	5,97	5,67
	30400	5,98	6,07	5,83	5,97	6,15	5,90
	31200	6,10	6,18	5,99	6,27	6,27	5,87
	32000	6,23	6,21	6,13	6,43	6,47	5,99

Table 3: Tmar For Different Network Delays

Varying Network Bandwidth

The second test runs illustrates how the client waiting time *Twait* depends on the network bandwidth, see Table 4. Again, the marshalling time *Tmar* depends only on the length of the method result. It does not depend on the bandwidth of the network connection. The network time Tnet depends on the bandwidth and on the method result length. Figure 97 presents the *Twait* timing results for 6 bandwidth settings, going from 50 KByte/s to 1600 KByte/s. For all bandwidth settings, the network delay time is set to zero.

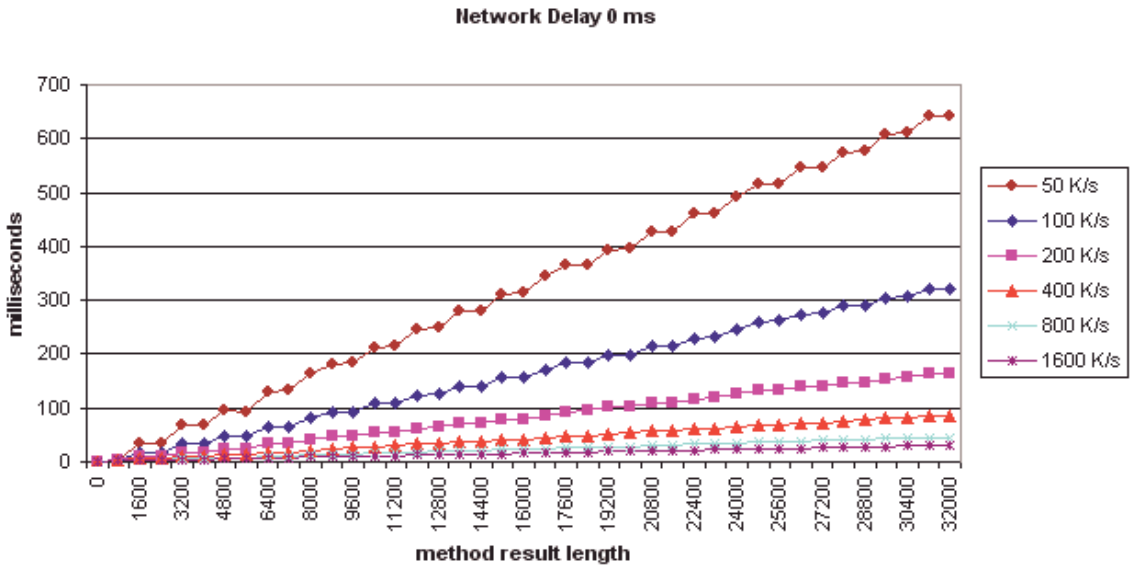


Figure 97: Twait For Different Bandwidths

The bandwidth limitation test run shows that the bandwidth affects the growth of the data series. The growth does – in the long run – not depend on the network delay time, whereas the offset values of the data series do not depend on the configured bandwidth limitation.
In Figure 97, one can observe that the test series are not linear but develop a sawtooth pattern. This behavior can be noticed across all bandwidth configurations. To explain this 'sawtooth' effect we have to illustrate how the NIST Net package, which is used on the router machine to emulate network delay and bandwidth limitations is working.
As opposed to real networks, the NIST Net does not implement bandwidth limitations on a bits-per-second basis but on a packet-per-second basis. That is, each time an IP packet is received by the NIST Net package, it is held back until a certain time interval (depending on the configured bandwidth) has elapsed. After this time interval has elapsed, the IP-packet is sent out. In the test runs, the method result value increment is 800 characters, which translates to 800 bytes at the TCP/IP level. Since the IP packet size is greater than 800 bytes in the test

run configuration, an additional IP-packet has to be created only every other 800-byte increments, which means that the hold-back time of the IP-packets increases only every other 800-byte-unit. Therefore the test run result values produce a sawtooth pattern.

		Network Bandwidth [K/s]					
		50 K/s	100 K/s	200 K/s	400 K/s	800 K/s	1600 K/s
Method Result Length	0	1,52	1,09	1,09	1,14	1,10	1,10
	800	2,59	1,99	2,00	1,97	1,92	1,92
	1600	33,99	16,90	9,17	5,55	3,59	2,97
	2400	33,88	17,55	10,07	6,25	4,42	3,64
	3200	66,70	33,26	17,00	9,41	5,66	4,40
	4000	66,90	33,04	18,01	10,40	6,57	5,08
	4800	94,88	47,15	24,78	13,59	7,84	5,85
	5600	93,29	48,18	25,56	14,25	8,49	6,51
	6400	130,81	64,17	32,85	17,70	10,10	7,25
	7200	132,71	66,45	33,60	18,49	10,78	7,93
	8000	163,97	81,08	40,54	21,55	12,14	8,72
	8800	182,52	92,57	47,52	25,03	13,74	9,51
	9600	183,44	93,40	48,49	25,87	14,53	9,91
	10400	212,87	107,71	55,23	28,95	15,78	10,70
	11200	213,73	108,57	56,06	29,64	16,54	11,30
	12000	246,49	123,36	63,05	33,17	18,13	12,16
	12800	249,11	125,23	64,62	33,86	18,83	12,88
	13600	279,70	138,93	70,86	37,05	20,19	13,53
	14400	280,66	140,41	71,84	37,83	20,82	14,39
	15200	312,29	155,50	78,71	41,30	22,50	14,99
	16000	313,53	156,67	79,55	41,99	23,25	15,71
	16800	343,73	171,87	86,45	45,14	24,56	16,79
	17600	363,67	183,60	93,64	48,63	25,88	17,42
	18400	364,50	184,71	94,57	49,44	26,77	18,07
	19200	394,37	198,80	101,36	52,56	27,96	18,89
	20000	395,51	199,69	102,11	53,40	28,85	19,54
	20800	427,66	214,36	109,31	56,79	30,60	20,37
	21600	427,69	215,36	110,13	57,55	31,24	21,05
	22400	460,21	229,96	117,02	60,66	32,67	21,87
	23200	461,60	230,85	117,98	61,48	33,31	22,53
	24000	493,09	246,16	124,99	64,92	34,89	23,39
	24800	514,48	259,39	131,91	68,08	36,23	24,08
	25600	515,30	261,78	132,72	68,79	36,92	24,69
	26400	544,89	274,87	139,82	72,27	38,52	25,48
	27200	545,76	275,81	140,57	72,97	39,35	26,24
	28000	575,23	289,88	147,48	76,24	40,65	26,86
	28800	577,69	291,13	148,24	77,11	41,56	27,84
	29600	607,47	305,58	155,36	80,60	43,06	28,42
	30400	609,55	306,50	156,23	81,22	43,88	29,14
	31200	640,92	320,84	163,25	84,41	45,18	29,93
	32000	642,22	321,50	163,86	85,17	45,89	30,68

Table 4: Twait For Different Bandwidths

The Tmar values of this test run are the same as in Table 3 and therefore not shown here.

9.3. Address Book Sample Performance

The following test runs results measure the performance of the AddressBook sample application as described in section 1.4. The test bed configuration is as described in Appendix 9.1.

Test Run 1, Varying Network Parameters

This test runs presents the performance of the Address Book sample application under different network conditions. The parameters of the test run are listed in Table 5. The measurement results for Twait, Tnet and Tmar are listed in Table 6, Table 7 and Table 8. A graphical representation of Twait (which is the same as the cycle time) is shown in Figure 98.

Number Of Search Result Entries	Np=10
Number Of Attributes Per Person	Na=3
Number Of Remote Method Calls Per Test Cycle	Nn=31
Method Result Length	l=10
Network Delay Time	Tdelay=0…160 ms
Network Bandwidth	b=50..1600 Kbyte/s

Table 5: Test Run 1 Parameters

		Bandwidth [Kb/s]					
		1600 Kb/s	800 Kb/s	400 Kb/s	200 Kb/s	100 Kb/s	50 Kb/s
Delay	0 ms	31	30	30	33	45	88
	5 ms	345	345	345	344	344	350
	10 ms	655	655	655	655	655	655
	20 ms	1277	1276	1276	1276	1276	1276
	40 ms	2519	2519	2519	2519	2519	2519
	80 ms	4996	4996	4996	4996	4996	4996
	160 ms	9963	9963	9964	9964	9963	9963

Table 6: Test Run 1, Twait Result Values

		Bandwidth [Kb/s]					
		1600 Kb/s	800 Kb/s	400 Kb/s	200 Kb/s	100 Kb/s	50 Kb/s
Delay	0 ms	14	14	14	14	28	70
	5 ms	327	328	328	328	328	334
	10 ms	638	638	638	638	638	638
	20 ms	1258	1259	1259	1259	1259	1259
	40 ms	2500	2500	2500	2500	2500	2500
	80 ms	4976	4975	4976	4976	4975	4976
	160 ms	9940	9940	9940	9940	9940	9940

Table 7: Test Run 1, Tnet Result Values

		Bandwidth [Kb/s]					
		1600 Kb/s	800 Kb/s	400 Kb/s	200 Kb/s	100 Kb/s	50 Kb/s
Delay	0 ms	17	17	16	19	16	18
	5 ms	17	17	17	16	16	16
	10 ms	18	17	17	17	17	17
	20 ms	18	18	18	18	18	17
	40 ms	19	19	19	19	19	18
	80 ms	21	20	20	20	20	20
	160 ms	23	23	23	23	23	23

Table 8: Test Run 1, Tmar Result Values

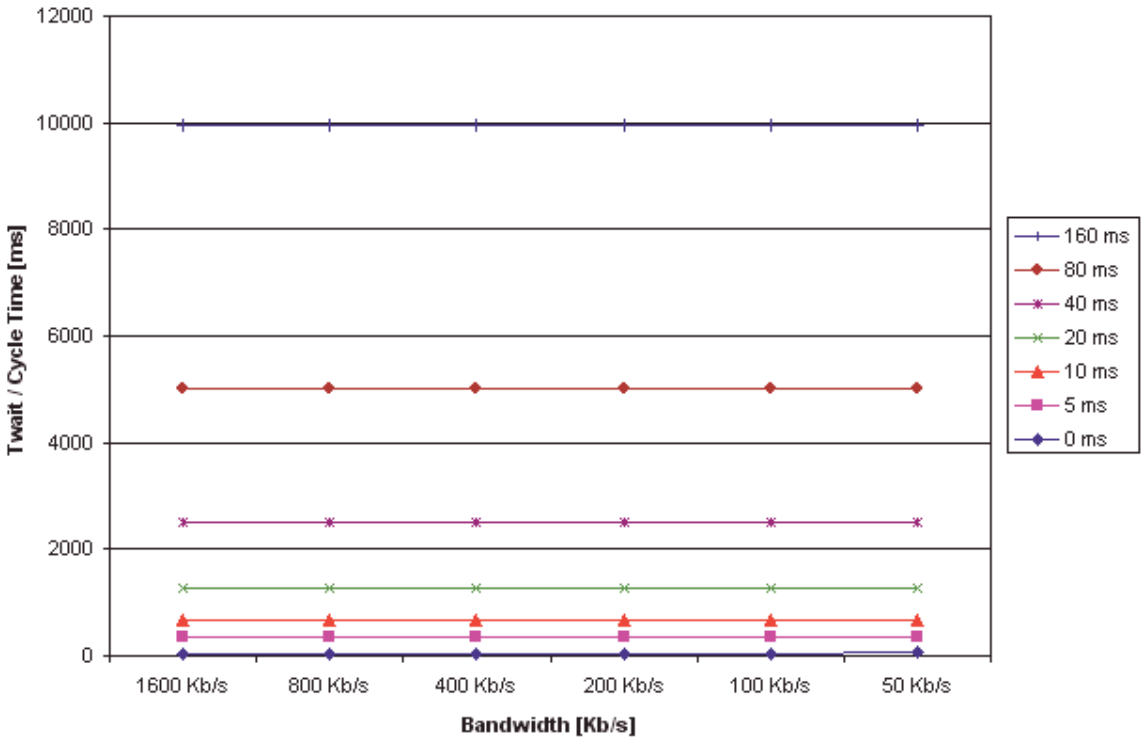


Figure 98: Test Run1, Twait / Cycle Time Result Diagram

Test Run 2, Varying Number Of Attributes Per Person

This test run evaluates the performance of the AddressBook sample application with different numbers of attributes per Person. The parameters of the test run are listed in Table 9. The measurement results for Twait are listed in Table 10. A graphical representation of Twait (which is the same as the cycle time) is shown in Figure 99.

The number of remote method calls is given as:

```
Nm = 1 + Np * Na
```

Where
Nm is the number of remote method calls per test cycle
Np is the length of the search() result list, the number of matching Person entries
Na is the number of attributes per Person.

Number Of Search Result Entries	Np=10
Number Of Attributes Per Person	Na=0…3
Number Of Remote Method Calls Per Test Cycle	Nm=1+10*Na (=0...31)
Method Result Length	l=10
Network Delay Time	Tdelay=0…40 ms
Network Bandwidth	No Limitation (full Ethernet speed)

Table 9: Test Run 2 Parameters

		Number Of Attributes Per Person (Na)			
		Na = 0	Na = 1	Na = 2	Na = 3
		Nm = 1	Nm = 11	Nm = 21	Nm = 31
Delay	0 ms	4	13	23	32
	10 ms	24	235	446	657
	20 ms	44	456	867	1278
	40 ms	84	896	1708	2520

Table 10: Test Run 2, Twait / Test Cycle Time Result Values [ms]

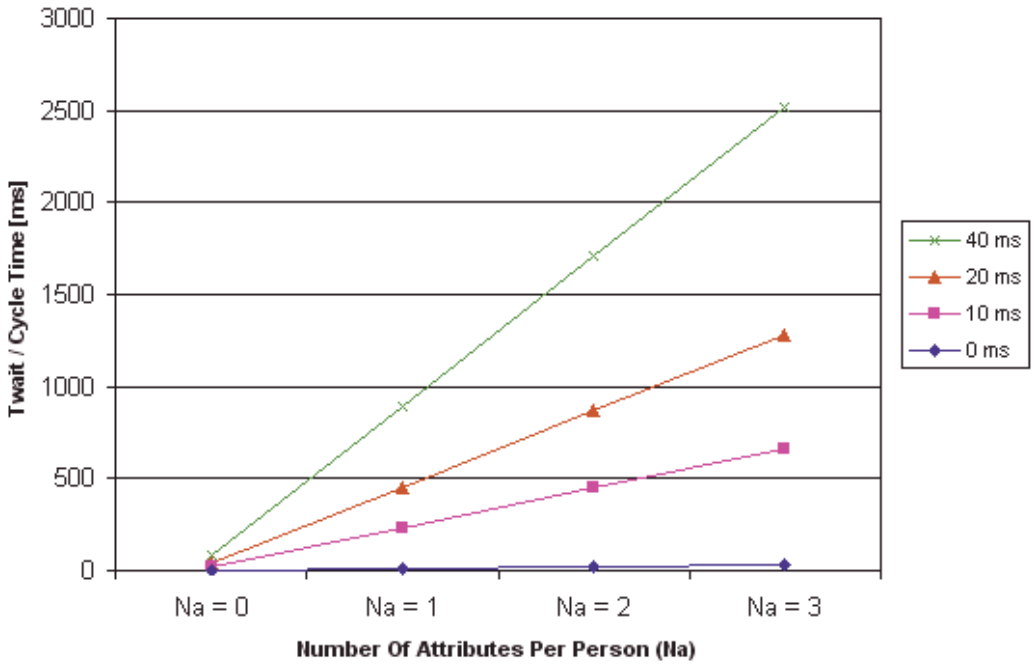


Figure 99: Test Run 2, Twait / Cycle Time Result Diagram

The main result of this test run is that the cycle time is a linear function of the number of remote method calls. The growth of the function is determined by the network delay time, which could have been expected, because each remote method call needs a network roundtrip,

and the roundtrip time is twice the configured network delay time. The bandwidth was not limited, so the full ethernet speed (see Appendix 9.1) could be utilized.

9.4. Current Practice Performance

The following test runs results present the performance of the Address Book sample application as described in section 1.4. The test bed configuration is as described in Appendix 9.1. The Address Book sample application was implemented in several different ways, as described in section 1.5:

1. Straightforward Implementation. See section 1.4.4.
2. Fat Operations. See section 1.5.1.
3. Data Structures. See section 1.5.2.
4. Objects By Value. See section 1.5.3.
5. Asynchronous Method Calls. See section 1.5.4.

The network delay was set from 0 to 160 ms, the network bandwidth was not limited.

Number Of Search Result Entries	Np=10
Number Of Attributes Per Person	Na=3
Number Of Remote Method Calls Per Test Cycle	Nm, see test run descriptions below.
Attribute Length	l=10 characters
Implementation Execution Time	Timpl=0 ms
Network Delay	0 ms ... 160 ms
Network Bandwidth	No Limitation (full Ethernet speed)

Table 11: Test Run Parameters

In each test run, the client was calling the `search()` method and iterating over the result list. We measured for each remote method call the client waiting time, the network time and the marshalling time. All times were then summed up and are presented in the data tables (Table 12 to Table 15) as *Twait*, *Tnet* and *Tmar*, rounded to the nearest millisecond value. The implementation times *Timpl* of all method implementation can be considered as 0 ms, all result values are pre-computed and stored in memory.

	Straightforward Implementation (Nm=31)			
		Twait [ms]	Tnet [ms]	Tmar [ms]
Network Delay	0 ms	32	14	18
	10 ms	657	638	19
	20 ms	1276	1259	17
	40 ms	2520	2502	18
	80 ms	4996	4975	19
	160 ms	9961	9942	19

Table 12: Straightforward Implementation Result Values

	Fat Operations (Nm=11)

		Twait [ms]	Tnet [ms]	Tmar [ms]
Network Delay	0 ms	15	6	9
	10 ms	236	228	8
	20 ms	457	448	9
	40 ms	899	889	10
	80 ms	1776	1768	8
	160 ms	3536	3528	8

Table 13: Fat Operations Result Values

	Data Structures (Nm=1)			
		Twait [ms]	Tnet [ms]	Tmar [ms]
Network Delay	0 ms	2	1	1
	10 ms	22	21	1
	20 ms	42	41	1
	40 ms	82	81	1
	80 ms	162	161	1
	160 ms	322	321	1

Table 14: Data Structures Result Values

	Objects By Value (Nm=1)			
		Twait [ms]	Tnet [ms]	Tmar [ms]
Network Delay	0 ms	2	1	1
	10 ms	22	21	1
	20 ms	42	41	1
	40 ms	83	81	2
	80 ms	162	161	1
	160 ms	322	321	1

Table 15: Objects By Value Result Values

	Asynchronous Method Calls (Nm=31)			
		Twait [ms]	Tnet [ms]	Tmar [ms]
Network Delay	0 ms	154	6	148
	10 ms	198	45	154
	20 ms	301	85	216
	40 ms	434	165	270
	80 ms	776	324	452
	160 ms	1397	712	684

Table 16: Asynchronous Method Calls Result Values

The result values for Twait, Tnet and Tmar are shown in Figure 100, Figure 101 and Figure 102.

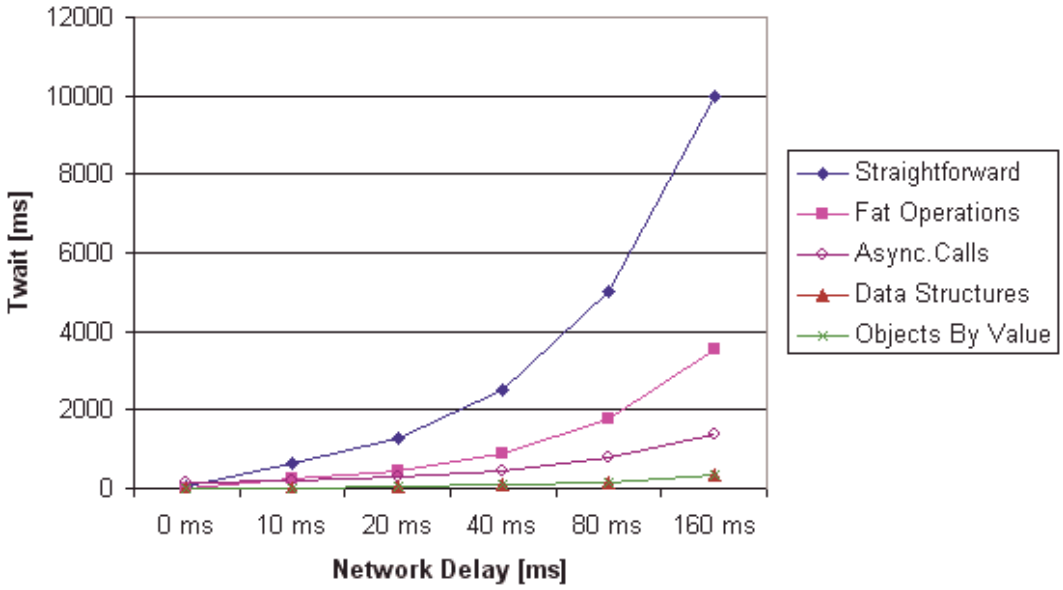


Figure 100: Test Run "Current Practice" / Twait Values

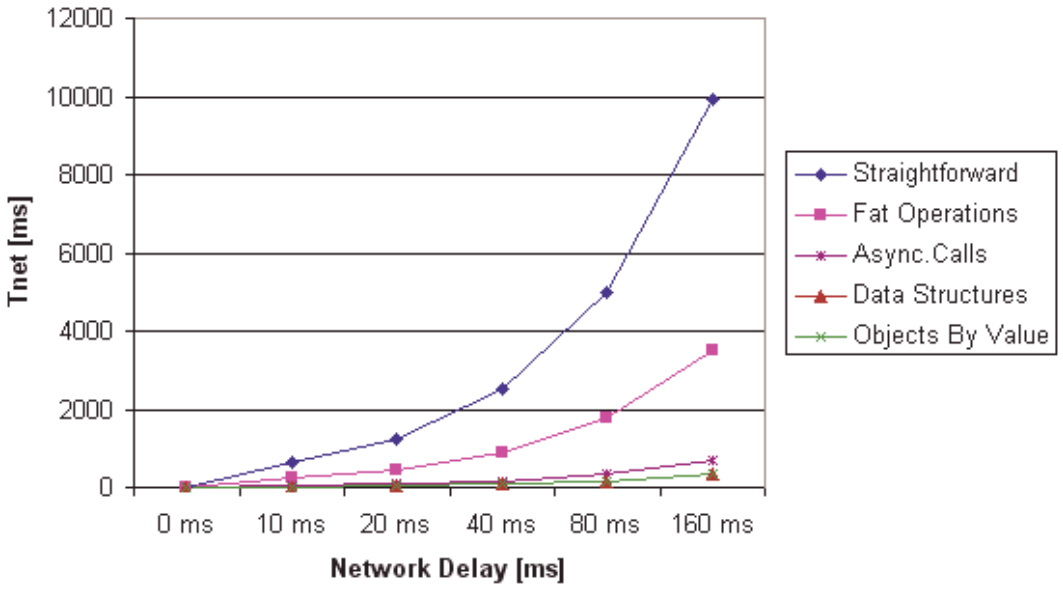


Figure 101: Test Run "Current Practice" / Tnet Values

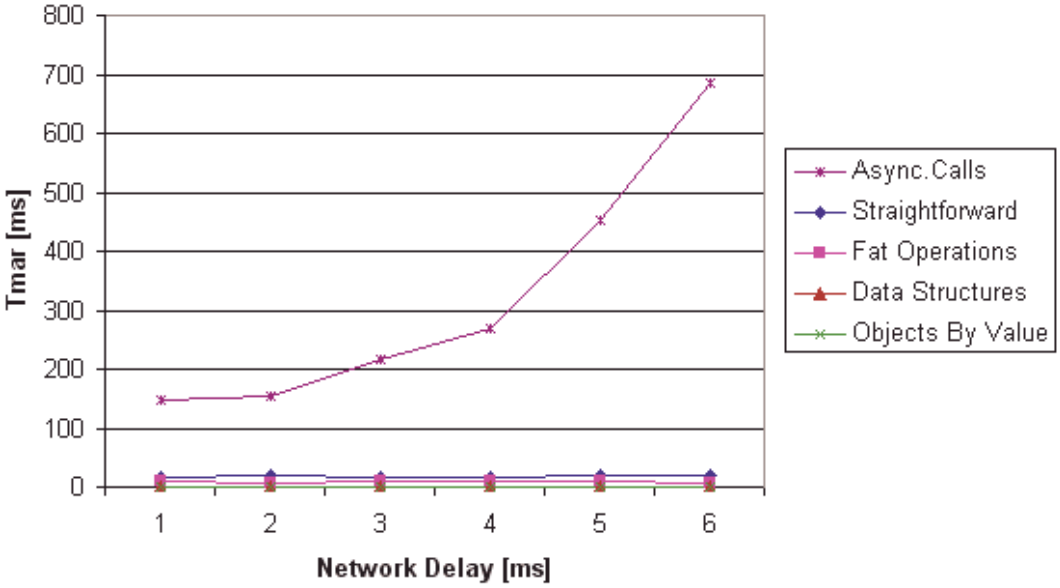


Figure 102: Test Run “Current Practice” / Tmar Values

9.5. Address Book Evaluation

The following tables and diagrams present the Address Book evaluation test result values, as described in section 5.1. The DOC-CaP approach was applied to the address book sample application (see section 1.4.1 for a description of the sample application) and evaluated using the testbed setup described in section 9.1.

Number Of Search Result Entries	Np=10
Number Of Attributes Per Person	Na=3
Method Result Length	l=10
Network Delay Time	0 ms … 160 ms
Network Bandwidth	5 KB/s … 160 KB/s

		Bandwidth					
		1600 KB/s	800 KB/s	400 KB/s	200 KB/s	100 KB/s	50 KB/s
Delay Time	0 ms	6961	2605	7584	8240	15786	31149
	10 ms	27178	22176	25208	22163	25859	40811
	20 ms	46911	42191	45341	42216	42108	50956
	40 ms	87026	82202	85342	82250	82215	82197
	80 ms	166865	162028	165233	162082	162054	162051
	160 ms	326173	322222	325448	322109	322238	322223

Table 17: AddressBook Evaluation, Twait Result Values

		Bandwidth					
		1600 KB/s	800 KB/s	400 KB/s	200 KB/s	100 KB/s	50 KB/s
Delay Time	0 ms	2002	2605	4544	8240	15786	31149
	10 ms	22153	22176	22162	22163	25859	40811
	20 ms	42090	42191	42194	42216	42108	50956
	40 ms	82197	82202	82199	82250	82215	82197
	80 ms	162012	162028	162048	162082	162054	162051
	160 ms	322212	322222	322179	322109	322238	322223

Table 18: AddressBook Evaluation, Tnet Result Values

Delay Time \ Bandwidth	1600 KB/s	800 KB/s	400 KB/s	200 KB/s	100 KB/s	50 KB/s
0 ms	3438	3321	3040	3076	3000	3435
10 ms	3312	3073	3046	3050	3020	3047
20 ms	2954	3109	3147	3079	3048	3037
40 ms	3243	3115	3143	3141	3041	3103
80 ms	3232	3207	3185	3204	3155	3165
160 ms	3401	3398	3269	3290	3293	3329

Table 19: AddressBook Evaluation, Tmar Result Values

Delay Time \ Bandwidth	1600 KB/s	800 KB/s	400 KB/s	200 KB/s	100 KB/s	50 KB/s
0 ms	6	5	4	3	2	3
10 ms	26	26	26	26	23	15
20 ms	28	28	28	28	28	24
40 ms	30	30	30	30	30	30
80 ms	30	30	30	30	30	30
160 ms	31	31	31	31	31	31

Table 20: AddressBook Evaluation, Twait Speedup Factors

9.6. TPC-W Evaluation

TPC-W XIDL Definition

```
typedef long long Date;

interface CCXact // ttl 5000
{
        string getTYPE(); // modifies none
        long getNUM(); // modifies none
        string getNAME(); // modifies none
        Date getEXPIRY(); // modifies none
        string getAUTH_ID(); // modifies none
        float getXACT_AMT(); // modifies none
        Date getXACT_DATE(); // modifies none
};


interface Country  // ttl 5000
{
        long getID(); // modifies none
        string getNAME(); // modifies none
        float getEXCHANGE(); // modifies none
        string getCURRENCY(); // modifies none
};
typedef sequence<Country> CountryList;


interface Address  // ttl 5000
{
        string getSTREET1(); // modifies none
        string getSTREET2(); // modifies none
        string getCITY(); // modifies none
        string getSTATE(); // modifies none
        string getZIP(); // modifies none
        Country getCOUNTRY(); // modifies none
};


interface Author // ttl 5000
{
        long getID(); // modifies none
        string getFNAME(); // modifies none
        string getLNAME(); // modifies none
        string getMNAME(); // modifies none
        Date getDOB(); // modifies none
        string getBIO(); // modifies none
};


interface Item // ttl 5000
{
        long getID(); // modifies none
        string getTITLE(); // modifies none
        Author getAUTHOR(); // modifies none
        Date getPU_DATE(); // modifies none
        string getPUBLISHER(); // modifies none
        string getSUBJECT(); // modifies none
        string getDESC(); // modifies none
        Item getRELATED1(); // modifies none
        Item getRELATED2(); // modifies none
        Item getRELATED3(); // modifies none
        Item getRELATED4(); // modifies none
        Item getRELATED5(); // modifies none
        string getTHUMBNAIL_NAME(); // modifies none
        string getIMAGE_NAME(); // modifies none
```

```
        float getSRP();         // modifies none
        float getCOST(); // modifies none
        Date getAVAIL(); // modifies none
        long getSTOCK(); // modifies none
        string getISBN(); // modifies none
        long getPAGE(); // modifies none
        string getBACKING(); // modifies none
        string getDIMENSIONS(); // modifies none
};
typedef sequence<Item> ItemList;


interface CartItem // ttl 5000
{
        Item getITEM(); // modifies none
        long getQTY(); // modifies none
        void setQTY( in long qty ); // modifies world
};
typedef sequence<CartItem>CartItemList;


interface Cart // ttl 5000
{
        Date getDATE(); // modifies none
        float getSHIP_COST(); // modifies none
        CartItemList getCartItems(); // modifies none
        void addItem( in long itemId, in long qty ); // modifies world
        void removeItem( in long itemId ); // modifies world
};


interface OrderLine  // ttl 5000
{
        Item getItem(); // modifies none
        long getQTY(); // modifies none
        string getCOMMENTS(); // modifies none
};
typedef sequence<OrderLine>OrderLineList;


interface Order  // ttl 5000
{
        long getID(); // modifies none
        Date getDATE(); // modifies none
        string getSHIP_TYPE(); // modifies none
        Date getSHIP_DATE(); // modifies none
        Address getSHIP_ADDR(); // modifies none
        string getSTATUS(); // modifies none
        OrderLineList getOrderLines(); // modifies none
        CCXact getCCXact(); // modifies none
};


interface Customer // ttl 5000
{
        long getID(); // modifies none
        string getUNAME(); // modifies none
        string getPASSWD(); // modifies none
        string getFNAME(); // modifies none
        string getLNAME(); // modifies none
        Address getADDRESS(); // modifies none
        string getPHONE(); // modifies none
        string getEMAIL(); // modifies none
        Date getSINCE(); // modifies none
        Date getLAST_VISIT(); // modifies none
        Date getLOGIN(); // modifies none
        Date getEXPIRATION(); // modifies none
        float getDISCOUNT(); // modifies none
        float getBALANCE(); // modifies none
        float getYTD_LIMIT(); // modifies none
        Date getBIRTHDATE(); // modifies none
```

```
        string getDATA(); // modifies none

        Cart getCart(); // modifies none
        Order placeOrder ( in string shipStreet1, in string shipStreet2,
                in string shipCity, in string shipState, in string shipZip,
                in long countryId, in string ccType, in string ccName,
                in long ccNumber, in Date ccExpiration,
                in string shipType ); // modifies world
        Order getLastOrder(); // modifies none
};


interface TPCWServer // ttl 5000
{
        void populate(); // modifies world
        void reset(); // modifies world
        CountryList getCountries(); // modifies none
        Customer login( in string uname, in string passwd ); // modifies world
        void registerCustomer ( in string UNAME, in string PASSWD,
                in string FNAME, in string LNAME, in string STREET1,
                in string STREET2, in string CITY, in string STATE,
                in string ZIP, in long COUNTRY_ID, in string PHONE,
                in string EMAIL, in Date BIRTHDATE, in string DATA ); // modifies world
        ItemList getPromoItems(); // modifies none
        ItemList getNewItems(); // modifies none
        ItemList getBestsellers(); // modifies none
        Item findItem( in long id );  // modifies none
        ItemList search( in string field, in string expr ); // modifies none
};
```

TPC-W Evaluation Result Values

		Twait [s]		
		CORBA	DOC-CaP	Speedup
Network Delay	0 ms	1	0	3
	10 ms	17	1	19
	20 ms	33	1	23
	40 ms	64	3	26
	80 ms	127	5	27
	160 ms	254	9	28

Table 21: TPC-W Evaluation, Twait Result Values

9.7. Automobile Quality Assurance Project Evaluation

		Test Configuration Straight-Forward			Test Configuration Data Structures			Test Configuration DOC-CaP		
Test Case	Delay	Twait [s]	StdDev		Twait [s]	StdDev		Twait [s]	StdDev	
Create Incident	0 ms	0,175	0,017	(10%)	0,148	0,013	(9%)	0,468	0,021	(4%)
	5 ms	0,926	0,026	(3%)	0,878	0,022	(3%)	1,173	0,028	(2%)
	10 ms	1,661	0,028	(2%)	1,586	0,015	(1%)	1,851	0,045	(2%)
	20 ms	3,126	0,026	(1%)	3,014	0,021	(1%)	3,266	0,059	(2%)
	40 ms	6,054	0,027	(0%)	5,913	0,027	(0%)	6,148	0,043	(1%)
	80 ms	11,931	0,077	(1%)	11,628	0,110	(1%)	11,969	0,091	(1%)
	160 ms	23,906	0,383	(2%)	23,172	0,368	(2%)	23,972	0,611	(3%)
Edit Incident	0 ms	0,049	0,004	(9%)	0,019	0,000	(1%)	0,012	0,001	(8%)
	5 ms	0,525	0,003	(1%)	0,203	0,001	(0%)	0,034	0,001	(4%)
	10 ms	0,998	0,005	(1%)	0,384	0,001	(0%)	0,054	0,002	(3%)
	20 ms	1,941	0,005	(0%)	0,743	0,000	(0%)	0,095	0,002	(2%)
	40 ms	3,827	0,007	(0%)	1,476	0,008	(1%)	0,176	0,003	(2%)
	80 ms	7,649	0,146	(2%)	2,914	0,004	(0%)	0,339	0,007	(2%)
	160 ms	15,205	0,089	(1%)	5,821	0,036	(1%)	0,681	0,016	(2%)
Incident List	0 ms	0,014	0,000	(1%)	0,005	0,000	(1%)	0,010	0,001	(7%)
	5 ms	0,167	0,000	(0%)	0,056	0,000	(0%)	0,022	0,002	(9%)
	10 ms	0,317	0,000	(0%)	0,107	0,000	(0%)	0,033	0,003	(8%)
	20 ms	0,618	0,000	(0%)	0,207	0,000	(0%)	0,054	0,003	(5%)
	40 ms	1,229	0,010	(1%)	0,407	0,000	(0%)	0,096	0,004	(4%)
	80 ms	2,417	0,001	(0%)	0,808	0,000	(0%)	0,179	0,009	(5%)
	160 ms	4,845	0,027	(1%)	1,627	0,028	(2%)	0,348	0,010	(3%)
Incident Overview	0 ms	0,193	0,011	(6%)	0,019	0,000	(3%)	0,016	0,002	(12%)
	5 ms	2,064	0,014	(1%)	0,172	0,001	(1%)	0,048	0,002	(4%)
	10 ms	3,907	0,011	(0%)	0,322	0,000	(0%)	0,079	0,002	(3%)
	20 ms	7,633	0,046	(1%)	0,625	0,002	(0%)	0,141	0,002	(1%)
	40 ms	14,977	0,011	(0%)	1,225	0,001	(0%)	0,265	0,014	(5%)
	80 ms	29,685	0,033	(0%)	2,428	0,002	(0%)	0,508	0,012	(2%)
	160 ms	59,333	0,139	(0%)	4,908	0,121	(2%)	1,007	0,056	(6%)
Repair Action List	0 ms	0,378	0,029	(8%)	0,052	0,004	(7%)	0,025	0,002	(9%)
	5 ms	3,580	0,025	(1%)	0,470	0,005	(1%)	0,076	0,002	(3%)
	10 ms	6,752	0,034	(1%)	0,880	0,003	(0%)	0,128	0,009	(7%)
	20 ms	13,081	0,041	(0%)	1,702	0,003	(0%)	0,229	0,008	(3%)
	40 ms	25,748	0,037	(0%)	3,347	0,004	(0%)	0,430	0,039	(9%)
	80 ms	51,045	0,151	(0%)	6,649	0,037	(1%)	0,835	0,094	(11%)
	160 ms	101,842	0,206	(0%)	13,227	0,023	(0%)	1,653	0,064	(4%)
QA Summary	0 ms	0,032	0,001	(4%)	0,017	0,000	(1%)	0,016	0,002	(11%)
	5 ms	0,367	0,001	(0%)	0,180	0,000	(0%)	0,069	0,003	(4%)
	10 ms	0,698	0,001	(0%)	0,342	0,003	(1%)	0,120	0,003	(2%)
	20 ms	1,358	0,000	(0%)	0,661	0,000	(0%)	0,219	0,002	(1%)
	40 ms	2,681	0,000	(0%)	1,313	0,010	(1%)	0,444	0,028	(6%)
	80 ms	5,320	0,007	(0%)	2,590	0,010	(0%)	0,868	0,014	(2%)
	160 ms	10,625	0,021	(0%)	5,176	0,028	(1%)	1,708	0,115	(7%)

Table 22: AQUA Evaluation Result

Table 22 shows the result values of the AQUA evaluation test runs. Each test case is executed with a variable network delay in the range of 0 to 160 milliseconds. (Note that a network delay of 0 milliseconds is a theoretical value.) Each test case was executed 40 times and the outliers were removed by removing the 10% largest and 10% smallest values, a method described in [66]. The table shows the average client waiting time (column *"Twait"*) in seconds for each test case and each test configuration. The table shows also the standard deviation as an absolute value in seconds (column *"StdDev"*) and a percentage value, which is calculated as the absolute standard deviation divided by the *Twait* average value.