Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

František Kovařík

Supporting multiplatform applications with YA-RPC

Katedra softwarového inženýrství

Vedoucí diplomové práce: *RNDr. Petr Hnětynka, Ph.D.*

Studijní program: *Informatika*

Studijní obor: *Softwarové systémy*

Praha, 2011

Na tomto místě bych rád poděkoval Doc. RNDr. Petru Hnětynkovi, Ph.D. za jeho cenné rady, připomínky a neskonalou vstřícnost při vedení diplomové práce.

Stejné díky patří Mgr. Janu Čurnovi za poskytnutí možnosti se tímto tématem zabývat, za poskytnutí podkladů, následně všech připomínek a neustálého povzbuzování.

Rovněž patří velké díky mé rodině za veškerou podporu při studiu a tvorbu potřebného zázemí. Závěrem nemohu zapomenout poděkovat také své přítelkyni za její trpělivost a stálou podporu.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 15. 4. 2011

František Kovařík

Abstrakt

Název práce: *Supporting multiplatform applications with YA-RPC*

Autor: *František Kovařík*

Katedra: *Katedra softwarového inženýrství*

Vedoucí diplomové práce: *RNDr. Petr Hnětynka, Ph.D.*

E-mail vedoucího: *hnetynka@d3s.mff.cuni.cz*

Abstrakt:

Během posledních tří desetiletí se vzdálené volání procedur (RPC) stalo oblíbeným způsobem pro komunikaci mezi počítači a procesy, využívaným v mnoha různých distribuovaných systémech. I přes velké množství RPC protokolů a implementací vyvinutých během těchto let, ani jeden systém nenabízí veškeré podstatné funkce a zároveň uživatelsky přívětivé rozhraní pro programování aplikací.

V této práci představíme Yet Another Remote Procedure Call – YaRpc – specifikaci flexibilního a uživatelsky přívětivého middleware, který nabízí pokročilé funkce jako např. rozšiřitelnost a nezávislost na transportním či komunikačním protokolu, zpětná volání a konfigurovatelnou obsluhu volání metod. Následně popíšeme YaRpc Native Protocol (YNP), což je nový, minimalistický a výkonný RPC protokol s bohatou množinou funkcí. Dále představíme nativní implementaci YaRpc standardu a YNP protokolu v Javě a .NET Frameworku, a porovnáme jejich použití vůči protokolům jako je Java RMI, .NET Remoting a SOAP webové služby.

Klíčová slova: *YaRpc, vzdálené volání procedur, distribuované systémy*

Abstract

Title: *Supporting multiplatform applications with YA-RPC*
Author: *František Kovařík*
Department: *Department of Software Engineering*
Supervisor: *RNDr. Petr Hnětynka, Ph.D.*
Supervisor's e-mail address: *hnetynka@d3s.mff.cuni.cz*

Abstract:

Over the last three decades, Remote Procedure Call (RPC) has become a popular inter-computer and inter-process communication paradigm widely used by a variety of interconnected computer systems. Even though a number of RPC protocols and implementations evolved over those years, no single system offers a significant set of features, while providing an easy-to-use application programming interface.

In this thesis, we present Yet Another Remote Procedure Call – YaRpc, a specification of a flexible and programmer friendly middleware that offers advanced features such as pluggable transports and protocols, callbacks, and configurable method dispatch. Additionally, we define YaRpc Native Protocol (YNP), a new light-weight high-performance RPC protocol with a rich set of features. We provide a native implementation of both YaRpc middleware and YNP protocol for Java and .NET Framework, and compare its usability with Java RMI, .NET Remoting and SOAP web services.

Keywords: *YaRpc, remote procedure call, distributed system*

Table of Contents

Introduction 1
1. State of the Art 3
  1.1. History of RPC 3
  1.2. How RPC Works 5
  1.3. Why RPC 6
  1.4. Review of Existing Relevant RPC Technologies 7
    1.4.1. CORBA 7
    1.4.2. .NET Remoting 9
    1.4.3. Java Remote Method Invocation 10
    1.4.4. SOAP/WSDL 12
    1.4.5. XML-RPC 14
    1.4.6. REST 15
  1.5. Emerging RPC Protocols 15
    1.5.1. Etch 16
    1.5.2. Protocol Buffers 17
    1.5.3. Thrift 18
  1.6. RPC Protocols Comparison 19
2. Design 20
  2.1. Core Services 24
    2.1.1. YaRpcManager 24
    2.1.2. IYaRpcConnection 26
    2.1.3. YaRpcRequest 27
    2.1.4. YaRpcResponse 28
    2.1.5. IYaRpcTransport 29
    2.1.6. IYaRpcRemotable 31
    2.1.7. IYaRpcWorker 33
  2.2. Protocol Plug-ins 33
  2.3. Serialization 35
    2.3.1. YaRpcSmartStream 35

2.3.2. YaRpcWriter 37
2.3.3. YaRpcReader 38
2.3.4. Z-Encoding..39
2.4. Utility Services 40
2.4.1. YaRpcStub 40
2.4.2. YaRpcProxy 41
2.4.3. YaRpcConnectionBase 42
2.5. YaRpc Native Protocol (YNP) 42
2.5.1. Message Format 43
2.5.2. Serialization..45
2.5.3. Asymmetric Serialization..47
2.5.4. Nested Method Calls 47
3. Implementation 50
3.1. .NET Framework 2.0..50
3.2. Java 6..53
4. Evaluation 59
4.1.1. Java RMI..59
4.1.2. .NET Remoting 61
4.1.3. SOAP Web Services with C#..64
4.1.4. SOAP Web Services with Java 65
4.1.5. YaRpc with YNP in C#..67
5. Conclusion & Future Work.. 69
References & Annexes.. 71
List of Tables.. 73
List of Figures 74
Definitions and Abbreviations 76

Introduction

The history of Remote Procedure Call (RPC) can be tracked as far as 1976, when a first official RPC proposal has been described in RFC 707 [Rfc707]. In simple terms, RPC enables software developers to use an elementary programming construct – a procedure call – to perform a task as complex as communication with a remote computer system over a network link. Since 1976, dozens of different RPC communication protocols have been proposed, hundreds of implementations of those protocols developed, and many thousands applications created and deployed in all areas of the industry.

The RPC is intrinsically a client-server communication paradigm; one side, a client, issues a procedure call, which is dispatched by another side – a server, while possibly returning a response back to the client. The popularity of RPC in general stems from the simplicity of its use. A single procedure call on a client side encapsulates a sequence of tasks which the programmer otherwise would have to perform, such a serialization of the procedure parameters, formatting and submission of the message over the network using a specific *protocol*, receiving a response message from the server, and deserialization of the response in order to return a value from the procedure. Similarly, on the server side, the RPC infrastructure is responsible for receiving incoming network messages, deserialization of them into procedure call requests, mapping the requests to procedures registered by a programmer of the server system (aka *stubs*), and serialization and submission of the procedure's return value back to the client.

The existing RPC protocols vary in a number of factors, such as the representation of messages (text or binary), transport layer (TCP, UDP, HTTP, HTTPS…), supported data types (simple or complex), text encoding (ASCII, UTF-8…), byte ordering (little-endian or big-endian), encryption, or procedure invocation mode (unidirectional, bidirectional). A single RPC protocol typically has a number of implementations for various programming languages and platforms, offering different sets of features, runtime performance, and varying levels of mutual compatibility. Even

though the number of alternatives seems to be quite high, making a right selection from the zoo of RPC protocols and implementations for a particular application is not a simple task. Very often the constraints imposed by RPC protocols, or (un)availability or (in)appropriateness of a particular RPC implementation for a particular programming environment or operating system, forces programmers to change design of their applications, or even dictates a selection of another programming environment. Moreover, the implementations of feature-rich protocols tend to be too heavy-weight, with everything but easy-to-use tools and application programmable interface (API).

The main contribution of this thesis is to demonstrate that it is possible to support feature-rich high-performance RPC protocols with an easy-to-use API. We present *Yet Another Remote Procedure Call (YaRpc)*, a flexible framework that aids in the development of communicating applications. YaRpc is not just another RPC protocol; it is rather an extensible middleware that allows application developers to select a combination of a transport layer (e.g., TCP, UDP, HTTP, HTTPS, shared-memory) and communication protocol (e.g., SOAP, XML-RPC, CORBA) of their preference for a particular task. As a proof of concept, we provide two native implementations of the YaRpc specification, for Java and .NET Framework platforms. Additionally, we define the *YaRpc Native Protocol (YNP)*, a new light-weight high-performance binary RPC protocol that provides a large number of features such as server-initiated callbacks, customizable serialization and nested method calls. The core of the YaRpc framework is protocol-independent, and provides low-level services to particular protocol implementations, such as advanced dispatch threading control, high-performance and customizable serialization, asynchronous method invocation and dispatch, or interface definition based on reflection.

The text of the thesis is organized as follows: Section 1 describes basic concepts of RPC and reviews the state-of-the-art protocols and implementations. Section 2 describes the architecture of the YaRpc middleware and justifies the design decisions behind it. Section 3 then details particularities of Java and .NET Framework implementations. In Section 4, we compare usage of YaRpc with YNP protocol to Java RMI, .NET Remoting and SOAP web services. Section 5 concludes the main findings of this research and outlines the future work.

1. State of the Art

This section provides an overview of existing Remote Procedure Call (RPC) protocols and describes advantages and disadvantages of the popular ones. We provide a brief introduction to the fundamentals of RPC operation, followed by an analysis of some well established protocols as well as recently emerged ones. In the conclusion of this section, we show differences between the described protocols to demonstrate the specific values the new proposed middleware and protocol described and implemented within this work – the YaRpc and YaRpc Native Protocol (YNP) – brings into the equation.

1.1. History of RPC

RPC stands for Remote Procedure Call. As such it does not refer to any particular product, protocol or a single standard but rather describes a common design or architectural pattern in the distributed computing universe. The quintessential operation behind this term is the ability to execute functions/procedures on remote computers via some network connection or eventually in other address spaces on the same computer. An implementation of such a mechanism has to address various specific needs and aspects such as transfer of the computational context of an invoked procedure, interoperability considerations etc. in order to make the implementation suitable for a specific desired application. RPC thus makes it possible to physically (or architecturally) separate entities contributing to the same computational task, linking it closely to the field of distributed systems. RPC has evolved significantly over time keeping up with advent of new concepts and paradigms in areas like programming, computing or application architecture.

At the beginning of the computer era the most common way of computing was bound to mainframe nodes accessed via terminals. The boom of personal computers brought the need of gradually more sophisticated user interfaces and the application model changed. While previous programs needed a mainframe for their entire operation, the new model of client/server architecture allowed for transferring part of the execution

to users' desktops and thus drove improvement of user interfaces – effectively forming a two-tier architecture.

While the two-tier model is still being used by many applications, the requirements for efficient software development, manageability, scalability, interoperability and robustness motivated the efforts to create new more sophisticated platforms by introducing additional application layers (Multi Tier Architectures), breaking applications into components (Component Architectures), facilitating applications by thoughtful development and subsequent composition of services (Service Oriented Architectures) or indeed composition of predefined canonical hosted services as in the case of Cloud Computing. All of these concepts are built around ways how to efficiently communicate between the tiers, services or components and how to distribute and manage these services across various computational facilities for a greater flexibility – making a suitable RPC mechanism a must.

Another important mover in RPC evolution was the nearly universal adoption of Object Oriented Design paradigms across the computer society. Instead of only distinguishing between the layers, this model enables to represent atomic application abstractions as objects. Thus an object or a set of objects may represent an application layer, provide a service interface or form a single system component. Again, as these objects can be distributed, there clearly need to be means to remotely discover and invoke other objects' methods. The inter-object RPC mechanism is often referred to as Remote Method Invocation (RMI), or Remote Method Call.

First RPC techniques were investigated as early as 1976 by B. J. Nelson at XEROX PARC [Bir83]. Sun Microsystems popularized the technique with its SunSoft's ONC (Open Network Computing) remote procedure calls (SunRPCs) [Rfc1831], which was one of the first commercial implementations of RPC. In addition, source code for ONC RPC has been available since 1988 as RFC 1057 [Rfc1057]. For example, millions of systems running Network File System (NFS) use the ONC RPC libraries. In 1989 the Object Management Group (OMG) organization has been founded. OMG defined general system architecture for object-oriented applications based on the specification of the interface, and the standard has been published as Object Management Architecture (OMA).

1.2. How RPC Works

In this section we discuss fundamentals of the RPC operation. Figure 1 depicts the basic difference between a local procedure call and a remote procedure call.

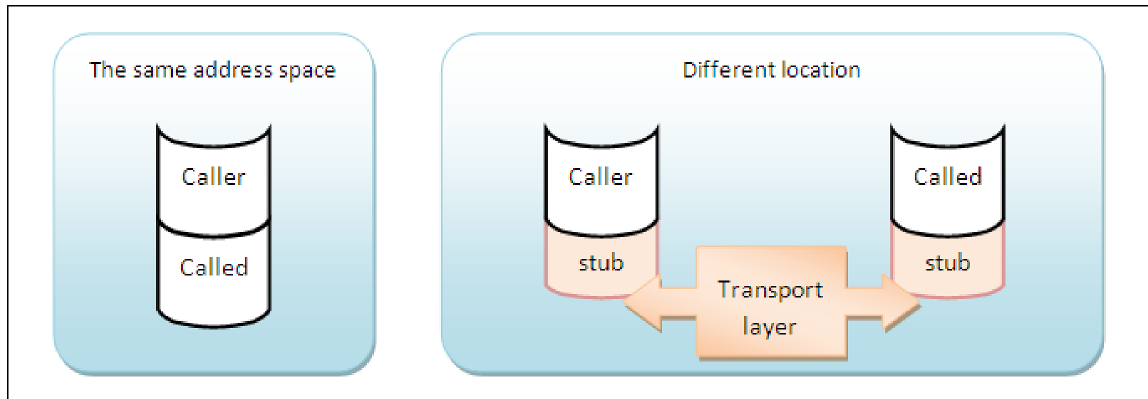

Figure 1 Local procedure call vs. remote procedure call

In a local procedure call, the execution context can easily be shared between the caller and the called code or simply passed through simple memory or register operations. In a remote procedure call, however, the context has to be explicitly transferred to called execution environment and the resulting context after the operation execution then needs to be pushed back to the client – potentially across the network. Such a transfer is usually described in the following steps, which effectively represent the basic RPC algorithm:

1. Serialization: encoding of input parameters and identification of the procedure into a pre-defined format for transfer.
2. Transfer: transfer of the serialized data to the remote server via a network transport.
3. Deserialization: decoding of the input parameters and identification of the called procedure.
4. Dispatch: call of the local procedure.
5. Serialization: encoding of the output parameters from the called procedure for transfer.
6. Transfer: transfer of the serialized data back to the client via a network transport.

7. Deserialization: decoding of the output parameters, and return of the values to the caller.

The format of the network messages, and particularities of the communication mechanism, is defined by an RPC *protocol*. Serialization (sometimes called marshalling), deserialization (unmarshalling) and data transfer routines are usually wrapped into objects denoted as *stub* objects (proxy objects, skeletons – the terminology is somewhat scattered). The way this helper code is generated – and indeed the time when it happens, the effort required by the developer to make it happen (level of automation), the level of control or finesse with which the two interconnected systems can interact, the various levels of support for various programming languages and architectures – is specific for specific RPC protocols and their implementations.

There is no "best" RPC mechanism – there are many typical tradeoffs one should consider when choosing the right RPC protocol and implementation for a particular application. The protocol with most features might not be the easiest to learn and use efficiently and comfortably, the most robust protocol might not be the fastest etc. Surprisingly enough, in many situation, all the major protocols seem suboptimal when considering all the options modern programming languages can provide – which is essentially the basic idea behind this thesis.

1.3. Why RPC

The typical advantages of adopting some RPC mechanism for inter-component communication are:

1. Execution location transparency – the called object may reside on a different computer without this fact being of any concern to the programmer.
2. Programming to interfaces – it is a good practice to hide the implementation behind a well-defined interface, and RPC mechanisms usually do this.
3. Platform independence – many RPC mechanisms provide implementations for various platforms, enabling efficient means of mixing different architectures into a single distributed system. This might provide a welcome freedom in integrating

legacy or third-party components while keeping the design of a system clean and simple.

4. Scalability/distribution potential – with execution location transparency it is easy to see how a system can be effectively distributed over a number of nodes. This makes it possible to increase the aggregate computational power of the system by increasing the number of nodes and potentially also exploiting routing of requests across nodes to balance the load of individual nodes.

1.4. Review of Existing Relevant RPC Technologies

As noted earlier, there is large number of RPC protocols and implementations – way more than we can cover within the scope of this work. However, we argue that we cover the most relevant RPC protocols for the purposes of this thesis and thus that the foundations for a further work are sound.

1.4.1. CORBA

CORBA, which stands for Common Object Request Broker Architecture, has emerged in 1991 and is maintained and developed within the OMG consortium [Omg04]. At the core of the architecture is the Object Request Broker (ORB) specification defining the basic rules for marshalling and unmarshalling phases of RPC, and the Interface Description Language (IDL) which allows for defining component interfaces in a platform/implementation independent way. Based upon IDL definitions the ORB implementation (specific for a given target platform) can generate the stub/proxy code for that particular platform, which can be easily integrated into existing application logic. A component providing such CORBA powered interfaces can then be called from other CORBA-enabled components regardless of their individual architectures and the fact that they communicate over the network. The ORB provides all necessary functionality for this to happen in a transparent manner.

In 1995, an important milestone was established with the CORBA Specification version 2.0. The rules for communication between ORB servers were made part of the standard itself, which further boosted the interoperability potential. The abstract inter-ORB protocol is called General Inter-ORB Protocol (GIOP). The most

fundamental implementation of GIOP protocol is based on TCP/IP network stack and is referred to as Internet Inter-Orb Protocol (IIOP).

During the further development, a number of extensions and generalizations has been defined, including other GIOP implementations and bridges. A basic CORBA architecture is depicted in Figure 2.

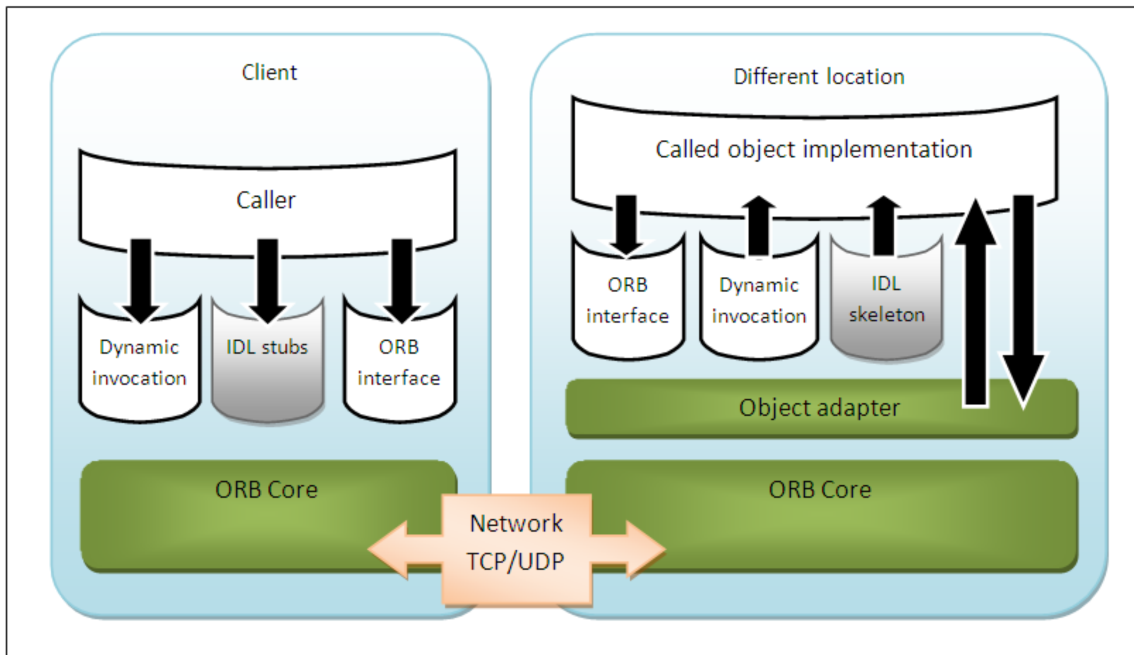

Figure 2 CORBA architecture

In order to make a request, the client can use the Dynamic Invocation interface (which is a uniform interface to call any CORBA component at the price of more cumbersome coding and more space for possible error). For more static integrations, a stub based on the target IDL definition can be generated in advance - this makes the calling trivial, but requires the stub-generation intermediate phase and thus makes the calling process less dynamic. The object implementation then receives the request through its own IDL generated skeleton (or stub, proxy…) or through a dynamic skeleton in symmetrical fashion.

CORBA is considered a very powerful and fully fledged remoting platform that provides its robust function set at the expense of relative un-ease of use and heavyweight footprint; this comes hardly as a surprise considering the fact that CORBA specification has more than 1000 pages.

1.4.2. .NET Remoting

Microsoft .NET Remoting is a Microsoft application programming interface (API) for inter-process communication. It was released in 2002 with the 1.0 version of .NET Framework [Net07]. Microsoft .NET Remoting is not exactly an RPC protocol, but it provides the same features like other RPC protocols do. It offers a comprehensive functionality including support for the transfer of objects by value or reference, operating conditions, life cycle, etc.

As .NET Framework is built around objects in a similar way Java is, it is objects that can be remotely accessed in .NET Remoting. An object, denoted as a remote reference from within the client application, can then be remotely accessed in a very easy and transparent way just like a normal object is. The stub code generation is hidden from the programmer and is performed automatically and dynamically for objects registered as client or remote objects with the system. All remote objects must be registered with the remoting system (environment) – at which point the environment generates necessary glue code. Instead of IDL the system uses reflection and code level annotations and obtains all the properties necessary for stub generation from the remote object definition itself, which makes the process much simpler. Figure 3 illustrates the basic operation of .NET Remoting platform.

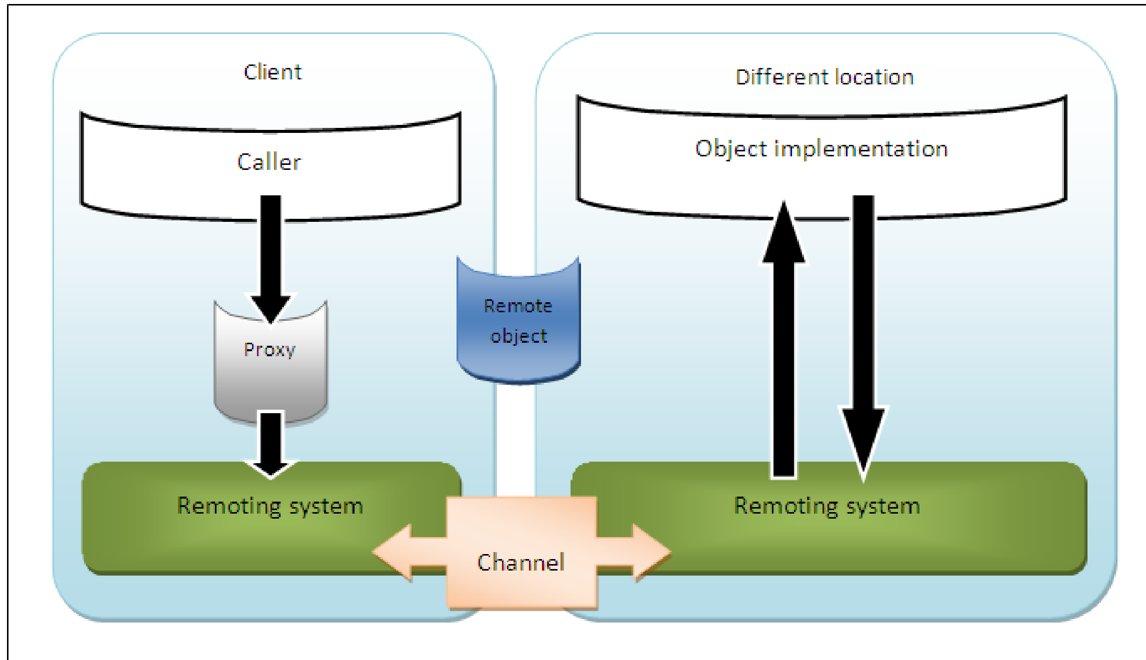

Figure 3 Microsoft .NET Remoting architecture

When writing a client application using a remote object, the developer requests a remote reference to that object from the remoting system and then uses this reference in the same way as if it was the object itself used locally. The reference is in fact a wrapper of a proxy class that is responsible for communication with the remote object.

The system allows for customization of communication channels with ad-hoc implementation of communication protocols – or a choice of typical standardized solutions such as TCP/IP (more efficient due to binary transport, but with firewalls problems) or HTTP (a little slower, but without firewalls problem) or a named pipe. .NET Remoting is very simple and powerful technology with one major disadvantage: it is only available in programs written in Microsoft .NET Framework.

1.4.3. Java Remote Method Invocation

Java Remote Method Invocation (RMI) is a robust and effective way to build a distributed application in a situation where all the participating programs are written in Java [Java06]. Interfaces and classes that are responsible for the RMI are distributed within the *java.rmi* package. RMI first appeared in JDK 1.1 and used the JRMP (Java Remote Method Protocol) protocol, which is an efficient binary protocol. Later RMI

adopted the RMI-IIOP which made it out-of-the-box compatible with CORBA powered systems.

The proxy code in RMI stack is generated by the RMI compiler - the *rmic*. Generated classes contain unique identification of the server callable objects, marshalling and unmarshalling code and a lot of other functionality. To use the remote object in RMI, the object has to be registered within the RMI registry. From Java version 2, it is not necessary to use skeleton, because they are dynamically generated based on reflection. Java RMI architecture is depicted in Figure 4.

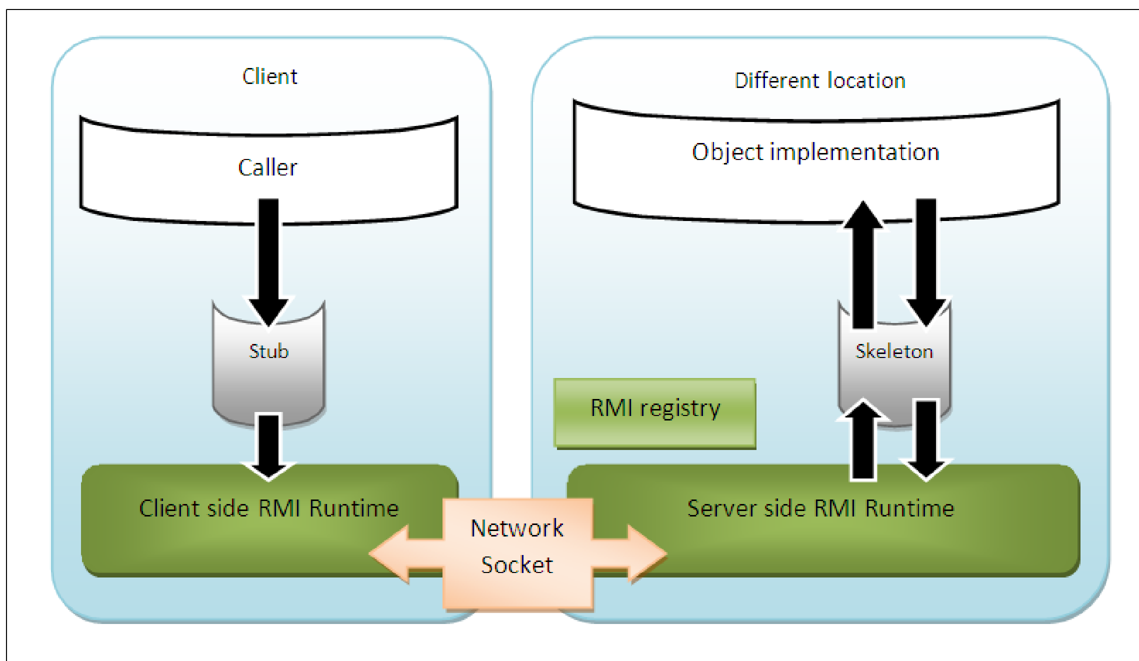

Figure 4 Java RMI architecture

The basic procedure a client uses to communicate with a server is as follows:

1. The client obtains an instance of the stub class. The stub class is automatically pre-generated from the target server class and implements all the methods that the server class implements.
2. The client calls a method on the stub. The method call is actually the same method call the client would make on the server object if both objects resided in the same Java Virtual Machine (JVM).

3. Internally, the stub either creates a socket connection to the skeleton on the server or reuses a pre-existing connection. It marshalls all the information associated to the method call and sends this information over the socket connection to the skeleton.
4. The skeleton unmarshalls the data and makes the method call on the actual server object. It gets a return value back from the actual server object, marshalls the return value, and sends it back over the wire to the stub.
5. The stub unmarshalls the return value and returns it to the client, and the control goes back to the client.

Today, with Java 6 SE, the classical pantheon of features is present to compete with other remoting solutions – including fully fledged implementations of remoting stacks based on various protocols, dynamic proxy generation based on annotations and reflection etc. Thus Java RMI is very a powerful remoting stack with a one major disadvantage: to use it, we have to use Java.

1.4.4. SOAP/WSDL

SOAP, which stands for Simple Object Access Protocol, is a protocol for exchanging XML-based messages over the network, mainly via Hypertext Transport Protocol (HTTP) [Web04]. The SOAP format is a basic layer of communication between web services and provides an environment for creating more complex communication. The SOAP format is based on a previous protocol XML-RPC. XML-RPC is a protocol that provides a set of rules how to use an already functioning and standardized technology for the needs of the RPC.

WSDL stands for Web Services Description Language. WSDL is a language powered by a set of specifications that allows for XML-structured descriptions/specifications of web services that can be used in similar fashion as IDL in CORBA. Based upon a service WSDL advertised descriptor, a client can decide how to structure the parameters in a SOAP/XML message so that the service understands the request. WSDL is then used for proxy code generation with many tools available for this task. Even more comfort is available when the actual WSDL is generated from object façade by further automation based on reflection or annotations. Such functionality is

well defined within the J2EE stack specification and is provided by most of the standard J2EE containers. This makes deployment of a service end-point extremely easy.

SOAP and WSDL are well-established standards which are widely used in practice. Complementary to the semantic aspect of service provision is the service directory and discovery functionality and the service contract description functions. Web service APIs as final products of enterprises are more and more common and thus a need for accepted fitting standards in above mentioned areas is growing. UDDI (Universal Description, Discovery and Integration) protocol provides a standardized semantics for service directory facilitation, service discovery and service contract specification. WSIL (Web Services Inspection Language) protocol provides standardized means to aggregate information about a web service in an efficient wrapper – this includes both service WSDL functional specification as well as the UDDI meta-data bound in a uniform presentation façade. The standard also defines ways how to query and process this information. Web services architecture is depicted in Figure 5.

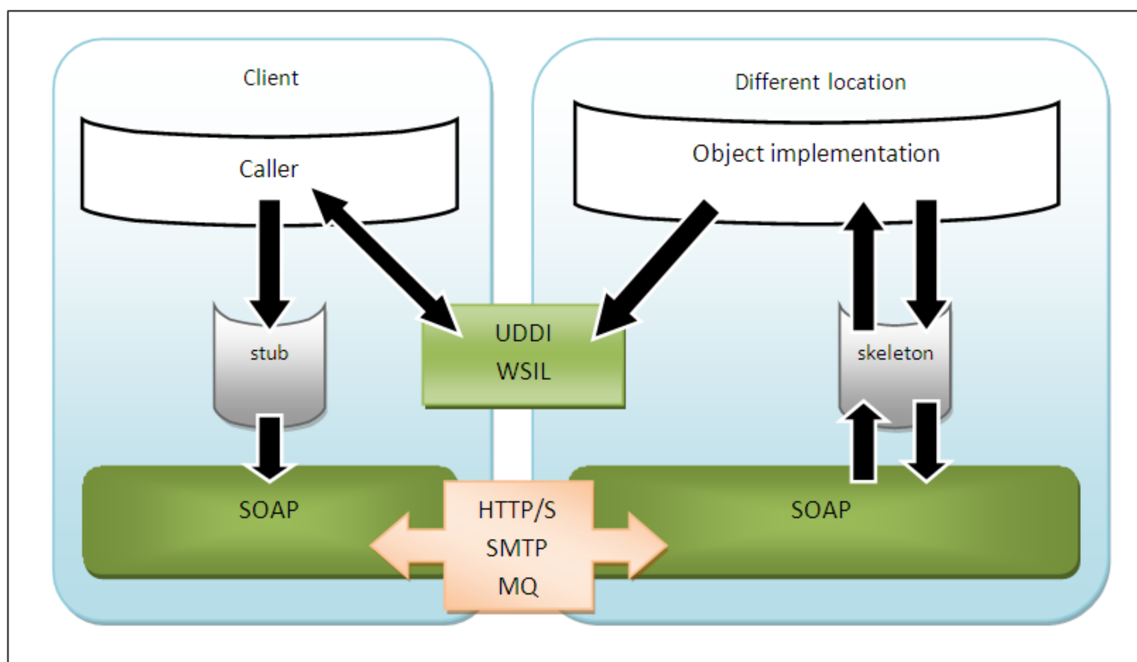

Figure 5 Web services architecture

Web services are de-facto a standard for cross-enterprise integration and service front provision. Therefore, the support for easy web service creation is one of the central pieces in every enterprise development platform or architecture, such as .NET, J2EE, all

the major IDEs etc. Web services provide well supported platform-independent and fully fledged RPC/communication stack.

On the other hand, one can argue whether the inherent complexity of web services is appropriate for all possible uses. Web services are not simply a data transport layer and not even a simple one at that. Probably the unanimous effort to support web services from all competing parties and the hype effect of repeating a single word too many times are to blame. Additionally, the XML-based communication protocol is too inefficient to be used by high-performance distributed systems.

1.4.5. XML-RPC

XML-RPC is a protocol which allows users to easily perform remote procedure calls [Xml99]. XML-RPC is a widely supported RPC mechanism with supporting libraries available for most common programming languages and some automation tools available for main enterprise application architectures. The data transferred during RPC calls is encapsulated using the XML language (eXtensible Markup Language) and transmitted through the HTTP protocol. In general, it can be said that XML-RPC is a far less complex than web services and while lacking some features necessary for full fledged enterprise SOA development, it is fitting for many simpler applications – and fitting better than web services. Currently the project is ended, but the XML-RPC protocol has been used as a draft for a newer SOAP protocol. XML-RPC architecture is depicted by Figure 6.

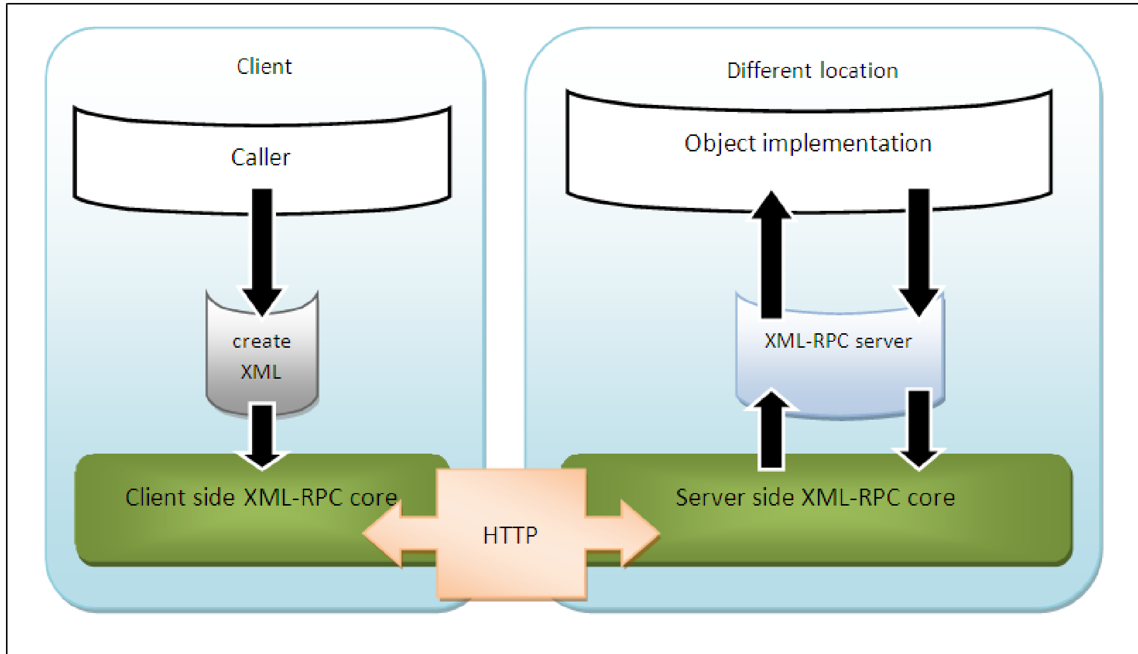

Figure 6 XML-RPC architecture

1.4.6. REST

The list of web-related RPC protocols would not be complete without mentioning REST [Rest07]. REST uses URL mappings and GET parameters to pass the procedure call parameters to the server and is one of the rapidly emerging technologies especially in application development for heterogeneous devices like iPhones, Android phones and web browsers. REST is a very simple and very easy to learn and use, which makes it ideal for numerous embedded or uncomplicated usages. REST is also de-facto standard in modern Model View Controller (MVC) web-development platforms – including Ruby on Rails, Groovy on Grails and many others. Obviously, REST is not an appropriate solution for high-performance distributed applications.

1.5. Emerging RPC Protocols

In this section we present some of the newer and very young protocols that extended the RPC family in recent years. These protocols provide new features or new techniques for more user-friendly programming. Advent of these protocols is mainly motivated by the boom in the internet and service market, inadequacy of existing RPC protocols for a particular use, and specific needs for application development and its

simplicity. These motivations do not different significantly from the motivations behind this thesis.

1.5.1. Etch

Etch is a relatively new protocol that was announced in May 2008 by Cisco Systems [Etch08]. Cisco presents Etch as an open source, cross-platform framework for building high-performance network services. It is intended to supplement SOAP and CORBA as methods of communicating between networked pieces of software, especially where there is an emphasis on portability, transport independence, small size, and high performance. There is a stable implementation for Java and C# programming languages with Ruby, Python and C support planned in the near future. Etch is available under Apache license in Apache Incubator. Etch architecture is illustrated by Figure 7.

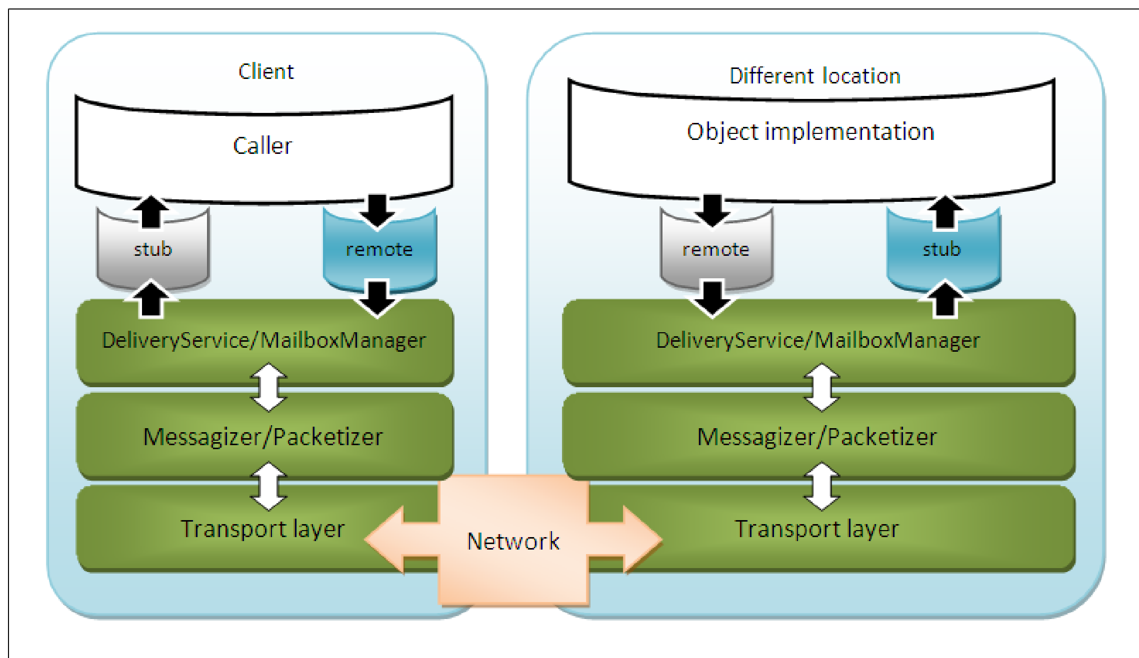

Figure 7 Etch architecture

Architecture of Etch is based on fully symmetric, flexible protocol stack for client and server. Stubs and skeleton are automatically generated by Java or .NET reflection, and a stub on client side corresponds with remote skeleton on a server side, and analogously, a remote skeleton on the client side corresponds to the stub on the server side. The Deliver Service and Mailbox Manager provide a layer for accepting and

managing incoming and outgoing requests. In a lower layer, the Messagizer or Packetizer deal with composing and parsing the actual transported messages, passing or getting the underlying byte arrays from the bottom level Transport Layer responsible for addressing, and actual network communication. The underlying communication channel may be TCP/IP or other protocol. The Etch stack natively supports Secure Socket Layer (SSL) for encryption.

Etch protocol tries to be simpler, faster and more flexible than existing competing systems that are either too slow, hard to use, bloated, and/or proprietary. The integration into development process is done on the lines of today commonly accepted scaffolding paradigm with a simple to use and integrate command-line utility generating necessary helping code, project fragments and other artifacts necessary for smooth and comfortable operation. Even though Etch is far simpler to use than other protocols, we believe that the simplicity of use can further be improved.

1.5.2. Protocol Buffers

Protocol Buffers, developed by Google, are not an RPC mechanism on its own [Gpb08]. Protocol Buffers is rather a way of encoding structured data in an efficient yet extensible format for the purposes of RPC-like communication. Google uses Protocol Buffers for almost all of its internal RPC protocols and file formats.

The main aspects of Protocol Buffers are language-neutrality, platform-neutrality and extensibility in the way structured data can be serialized. Compared to its main competitor in the RPC arena – the XML – Protocol Buffers format is simpler to parse and manipulate, resulting data is significantly smaller, the basic operations like simplistic traversing or searching is much faster and achievable by simpler means. All of this is achieved by creating a format that is less general and less ambiguous than XML, and thus easier to parse systematically, orderly and still correctly. Data access classes generated along the way of parsing the data stored in this format were developed with ease of programmatic use in mind – a strong and reasonable selling point these days.

The Protocol Buffers protocol is a valuable alternative to more sophisticated formats and may help to significantly save space for storage of large amounts of data, to speed-up transmission of data over the internet and thus accelerate the execution of RPC. Implementations of the protocol are available in C++, Java and Python – a typical

Google portfolio. Alongside JSON, it is a compelling alternative to structure complex data transferred between components or applications.

1.5.3. Thrift

Thrift is an easy-to-use cross-platform and cross-language framework for developing service infrastructures [Ft07]. The protocol was originally developed at Facebook before it went open source in April 2007 and entered the Apache Incubator in May, 2008.

Thrift offers scalability, efficiency and freedom of choice of language and platform. It provides support for C++, Java, Python, PHP, C# and other languages. Service data types and service interfaces can be defined in a simple definition file and the file is then compiled by a provided compiler to generate necessary proxy classes and management source code for facilitating the service client and/or server facades. Thrift architecture is illustrated in Figure 8.

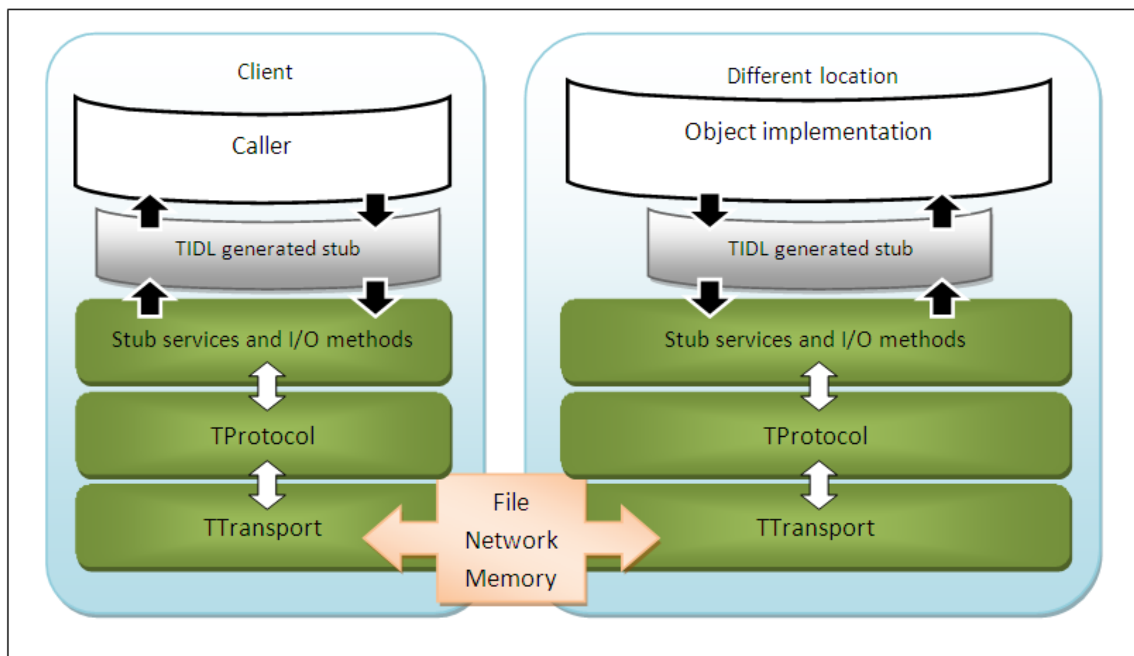

Figure 8 Thrift architecture

Service interfaces are specified using a dedicated language – the Thrift Interface Description Language (TIDL), which is being processed by the Thrift Preprocessor Utility. Multiple options for a transport layer facilitator are available.

1.6. RPC Protocols Comparison

Table 1 summarizes the features of the RPC technologies discussed, in order to compare their strengths and weaknesses.

Protocol / Feature	CORBA [Omg04]	.Net Remoting [Net07]	Java RMI [Java06]	Web Services [Web04]	XML-RPC [Xml99]	Etch [Etch08]	Protocol Buffers [Gpb08]	Thrift [Ft07]
RPC	✓	✓	✓	✓	✓	✓		✓
IDL needed	✓					✓	✓	✓
Transports	TCP, SSL, HTTP(S), bridges	TCP, HTTP(S)	TCP, HTTP(S)	HTTP(S)	HTTP(S)	TCP,UDP, SSL, HTTP(S), …	TCP, UDP, files, memory …	TCP, UDP, files, memory, zlib, …
Protocols	GIOP	.NET Remoting	JRMP, RMI-IIOP	SOAP	XML-RPC	Binary, XML, …	Binary, XML, JSON,…	TBinary, TCompact, TJSON, ...
Free for languages	Java, C/C++, Python, …	C#	Java	C#, Java, Python, C/C++, …	C#, Java, Python, C/C++, …	C#, Java, C	C#, Java, Python, C/C++, …	C#, Java, Python, C/C++, …
Callbacks	✓					✓		
Asynchronous method dispatch								✓
High performance	✓	✓	✓			✓	✓	✓
High scalability						✓	✓	✓

Table 1 RPC technologies comparison

2. Design

As discussed in Section 1, even the most widely accepted and used RPC technologies have serious issues in terms of a general usability and ease-of-use – at least for certain very general purposes. Features and particularities of the available RPC protocols and implementations often force developers of distributed applications to make non-trivial design decisions, such as to trade off high performance to code portability, choose a heavy-weight technology in order to support a single specific feature, resort to a paid RPC implementation in order to support a specific protocol on a specific platform, or to make non-elegant workarounds such as firewalls and/or Network Address Translation (NAT) reconfiguration on client systems in case callbacks are needed and a desired protocol does not support them.

In this thesis, we demonstrate that there is another way, and that it is possible to provide a portable RPC middleware with a wide range of features, yet fast, lightweight, and simple to use. YaRpc is not just a new RPC protocol or implementation; it is rather a specification of a toolkit and an application programming interface (API) for developers of distributed applications. The design goals of the YaRpc middleware can be summarized as follows:

1. Plug-in architecture and modularity in terms of network transport protocols, i.e. a possibility to use a number of transport layers such as TCP/IP, UDP, SSL, named pipes, etc.
2. Plug-in architecture and modularity in terms of communication protocols, i.e. a possibility to integrate existing protocols such as XML-RPC, SOAP, Java RMI, .NET Remoting, CORBA, etc.
3. Availability for both Microsoft .NET and Java platforms as an open source, or under some public license, with a possibility to extend to other platforms.
4. Simple and user-friendly rules for the definition of a communication interface between client and server. Interface description language (IDL), or external utilities for a manual proxy code generation are time consuming and not easy-to-use, and thus should be avoided whenever possible. We rather rely on

code reflection and want to achieve a high-level of user transparency in both development and operational process.

5. Bi-directional remote calls invoked by both client and server, for protocols that support it. This means that we want to make YaRpc a prepared solution for technologies of callbacks without any constraints on firewalls or NAT.
6. Asynchronous remote calls and server method dispatch. Both issuing and dispatch of a remote method calls should be able to take advantage of underlying operating system asynchronous I/O capabilities, and execute in background while not blocking the calling or dispatching thread, respectively. After the issue or dispatch of a remote method call is finished, the RPC mechanism initiates response processing via a callback.
7. High performance, fast and robust.
8. There is a need for a new RPC protocol, the YaRpc Native Protocol (YNP), that can take the best out of the YaRpc middleware, and existing RPC protocols. The protocol should be fast, light-weight and offer a comprehensive set of features, most prominently callbacks.

The ultimate goal of our effort was to provide an easy-to-use feature-rich RPC infrastructure that would be an interesting alternative to existing RPC systems and potentially even adopted by an open-source community for a further development. In order to achieve this goal, the framework needs to provide an attractive set of features that existing frameworks do not or cannot provide, and be available to a larger software developer community.

The starting point of our work was an early prototype of YaRpc, which was just a monolithic RPC protocol with an implementation for .NET Framework only. This prototype has been developed for one particular application – the DistributedMI distributed application [Jc07], and although it already contained some interesting features (e.g., nested call execution), it was far from being a generally usable RPC middleware. Another problem of the prototype was a fixation to a new unknown protocol, which altogether with a user-unfriendly API made it very unlikely that the prototype might ever be adopted by the desired target audience.

By a careful review of the YaRpc prototype, it became apparent that only a small subset of the code was actually related to the communication protocol itself, and the most of the code was dedicated to service routines that might be used by other protocols as well (e.g., serialization, remotable object registration, network transport routines). This led to an idea that the framework can be designed in an extensible fashion that would allow developers to plug-in additional protocols, and support implementations of these protocols by a rich lower-level API. Once the framework provides a sufficient number of protocol plug-ins, while still keeping a clean and easy-to-use API, it will have a potential to be widely adopted. Also, the provision of quality core services will be enough incentive for developers not to create new RPC systems from a scratch, but preferably implement their protocols as YaRpc plug-ins. It is necessary to make switching of the actual communication protocols absolutely seamless, without requiring users to significantly change their code.

Another important design goal was high runtime performance. Traditional threading models used by distributed applications, such as *select/poll* loops, do not scale well for large number of parallel network connections [Bhkk06]. Fortunately, modern operating systems provide new networking primitives such as I/O Completion Ports in Microsoft Windows, which can even scale to tenths thousands of parallel network connections [Jon02]. The RPC system should be able to take advantage of the operating system capabilities, and thus enable developers to create high-performance applications with a little effort. In practice this means that all slow operations, such as connection to a remote host or execution of a remote call, are always supported using asynchronous (non-blocking) methods; the caller issues a request to begin an asynchronous operation which gets executed in the background, and receives a notification that the operation has been finished using a callback, possibly called in another thread. This allows the original thread to perform some more interesting tasks than waiting. For a convenience, the synchronous (blocking) equivalents of the asynchronous methods are also provided, wherever appropriate.

From a programmer's point of view, the YaRpc specification only defines a set of interfaces and classes that should be provided by a particular implementation of the middleware. As such, the YaRpc can be implemented virtually in every modern

object-oriented programming language; however, a functionality that utilizes special features of a language, such as runtime code reflection, might be limited.

The object-oriented paradigm is adopted not only by the YaRpc specification, but also by the abstraction of the remote procedure calls. The remote procedure calls are assumed to always reference an object that will dispatch the calls. For this reason, in the later text we use the term *remote method call* instead, without any confusion. Note that the assumption of existence of an object that dispatches the remote calls does not force users or protocols to also adopt the object-oriented semantics; standard static procedure calls can easily be "emulated" by particular protocols implementations, if they choose so. An overall view of the YaRpc architecture is available in Figure 9.

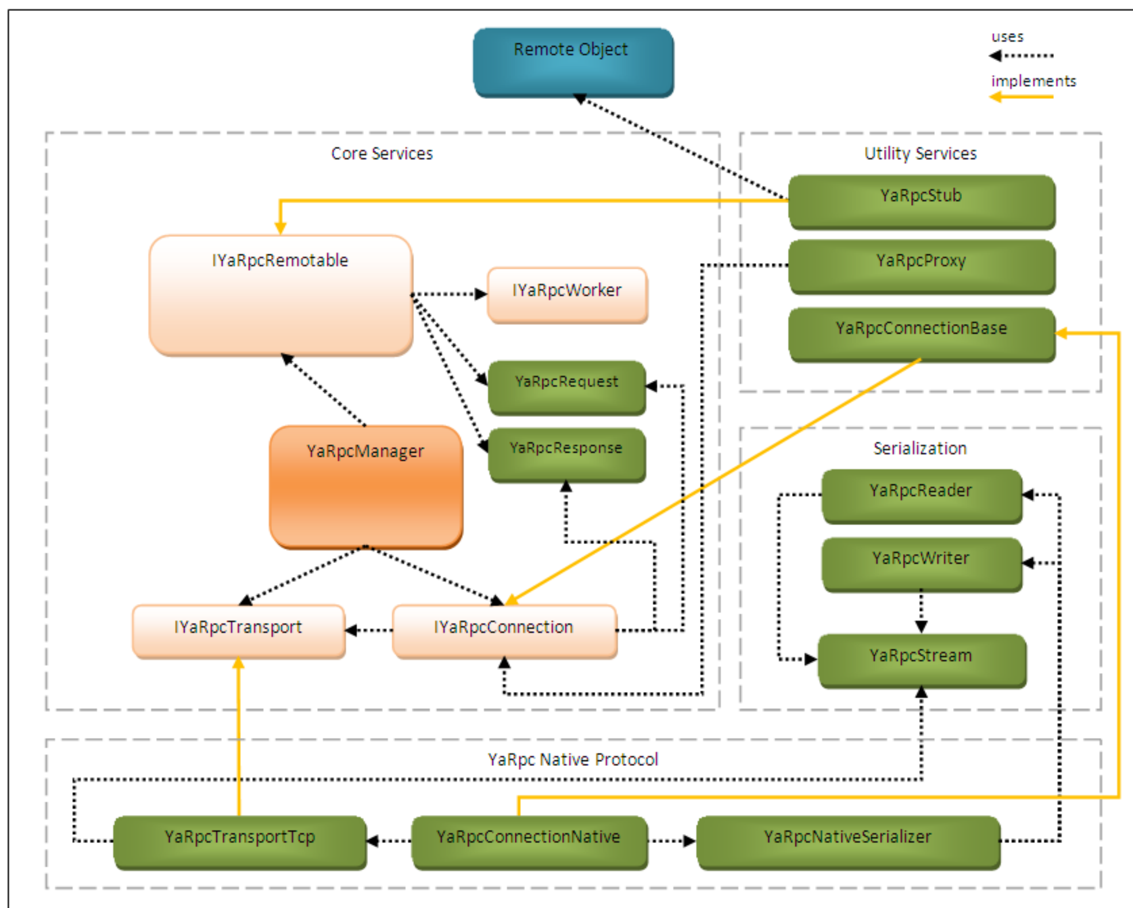

Figure 9 YaRpc architecture

2.1. Core Services

The Core Services part of the YaRpc specification defines the elementary services that are to be provided by each YaRpc implementation. The specification also defines a set of appropriate interfaces to guide "external" implementations of objects used by the middleware, such as protocol plug-ins.

2.1.1. YaRpcManager

The main entry point to the YaRpc infrastructure is the *YaRpcManager*, a class which contains static methods to manage aspects such as global configuration, protocol plug-ins, remotable object registration, opening of remote connections for clients and listeners for servers. Table 2 lists the methods of the *YaRpcManager* class.

Method	**Description**
`RegisterProtocol`	Registers a new protocol plug-in to the YaRpc middleware, under a specific name. The protocol plug-in is a combination of transport layer type implementation (*IYaRpcTransport* interface) and connection type implementation (*IYaRpcConnection* interface).
`TryGetRegisteredProtocol`	Determines whether a particular protocol plug-in is registered.
`UnregisterProtocol`	Unregisters a specific protocol plug-in.
`Connect`	Opens a connection to a remote host, and returns an instance of the *IYaRpcConnection* interface.
`BeginConnect`	Asynchronously connects to a remote host. This method is a non-blocking equivalent of the BeginConnect method.
`EndConnect`	Finishes an asynchronous connection to a remote host.
`StartListening`	Starts listening for incoming connections at a specific local end-point, using a specific protocol. A user-provided callback might be executed whenever a new connection is accepted.

`StopListening`	Stops listening for incoming connections on a specific local end-point.
`RegisterObject`	Registers a remotable object (implementing the *IYaRpcRemotable* interface) to receive the remote method calls, under a specific object key.
`IsObjectRegisterted`	Determines whether a specific remotable object is registered.
`GetObjectKeys`	Gets all object keys a remotable object is registered under.
`GetRegisteredObject`	Gets a registered remotable object based on its key.
`UnregisterObject`	Unregisters a remotable object.
`NewGuid`	A helper method to genereate a new globally unique identifier (GUID).
`NewObjectKey`	A helper method to generate a new object key.
`HideRemoteExceptionDetails`	Configuration parameter: indicates whether local exception details should be forwarded to a remote side, which might be a security issue.
`MaxReceiveMessageSize`	Configuration parameter: maximum size of any incoming message, in order to help prevent overloading of the server, or denial of service (DoS) attacks.
`MaxRecursionDepth`	Configuration parameter: maximum depth of recursion for nested method calls.
`BlockIncomingRequests`	Configuration parameter: indicates whether all incoming method calls should be blocked, to increase the level of security for pure clients.
`NotifyConnectionClosed`	An utility method that has to be called whenever an implementation of IYaRpcConnection determines that the connection has been closed.

Table 2 *YaRpcManager* class methods

2.1.2. IYaRpcConnection

The *IYaRpcConnection* interface represents an open connection to a remote host, or a process. It provides all the necessary methods to manage the connection and to issue a remote method call on that connection. The implementations of the *IYaRpcConnection* interface are responsible for serialization of the method parameters, and deserialization of the return values, and as such an *IYaRpcConnection* implementation in represents a particular RPC protocol. The methods of the *IYaRpcConnection* interface are described in detail in Table 3.

Method	**Description**
`State`	Gets the state of the connection (*None*, *Intializing*, *Open*, *Closed*).
`BeginIntialize`	Asynchronously initializes a new connection, over a provided communication channel (specified using the *IYaRpcTransport* interface). The initialization is protocol- specific, and might involve a network hand-shake between the two endpoints.
`EndInitialize`	Finishes asynchronous initialization of the connection, invoked using the *BeginInitialize* method.
`IsServer`	Gets a value indicating that this connection was accepted from the remote side, i.e. the local side acts as a server.
`Tag`	Gets or sets a user-provided tag associated with the connection.
`RemoteCall`	Synchronously issues a remote method call, specified using an instance of the *YaRpcRequest* class. After the remote call is finished, the method returns an instance of the *YaRpcResponse* class with the call results.
`BeginRemoteCall`	An asynchronous version of the *RemoteCall* method.
`EndRemoteCall`	Finishes an asynchronous method call, invoked using the *BeginRemoteCall* method.
`KeepAlive`	Can be called by the YaRpc middleware or a user to prevent the connection from timing-out.
`Ping`	Measures the duration of a single network roundtrip.
`Close`	Closes the connection.

`ConnectionClosed`	Registers a user callback which will be called when the connection was closed.

Table 3 *IYaRpcConnection* interface methods

Even though an implementation of the *IYaRpcConnection* interface will usually be associated with a physical network link (e.g., a TCP socket), it is not the same thing. The *IYaRpcConnection* serves merely as an abstraction that captures a state of communication between two end-points, and does need to be associated with a particular network link at all (e.g., consider an in-process connection). In the YaRpc middleware, the network link is considered as a separate entity, and is handled by the *IYaRpcTransport* interface, which is described in the Section 2.1.5.

Implementations of the *IYaRpcConnection* interface are also responsible for dispatching of the remote method calls received from a remote end-point of the connection. Typically, this will be the case for accepted incoming connections on a server (see the *IsServer* property), but it might also apply to client connections in case of a bidirectional RPC protocol. In order to dispatch the method call, the *IYaRpcConnection* implementation needs to use methods provided by the *YaRpcManager* class to obtain a reference to the dispatching object (aka *remotable* object), and invoke the appropriate method of the *IYaRpcRemotable* interface - see Section 2.1.6 for details.

2.1.3. YaRpcRequest

The *YaRpcRequest* class is a simple store for all the necessary information about a remote method call that a client is issuing to a server. An *IYaRpcConnection* implementation accepts a *YaRpcRequest* instance in the *RemoteCall* method (or *BeginRemoteCall*), and serializes the request to a message in a protocol-specific way for the transmission over the network link. Table 4 lists the properties of the *YaRpcRequest* class.

Property	**Description**
`ObjectKey`	The key under which the remotable object is registered on the server.

`MethodKey`	The name of the remote method to be executed.
`Inputs`	An array of input parameters for the remote method.
`InputsDefinition`	A list containing the name and type of each input parameter.
`OutputsDefinition`	A list containing the name and type of each return value from the remote method.
`Timeout`	Time after which the remote method call will be considered as failed.
`Flags`	Additional flags affecting the issue and execution of the remote call.
`Context`	A protocol-specific hash-map with additional information about the request.

Table 4 *YaRpcRequest* class properties

It might be surprising that the *YaRpcRequest* class also contains definition of the output values (see *OutputsDefinition* property), but in fact, this information is necessary for some implementations of *IYaRpcConnection* interface in order to correctly deserialize the returned message, especially in case of binary RPC protocols that do not transmit meta-data describing the type of return values. Fortunately, YaRpc implementations in high-level programming language such as Java or C# can use code reflection to automatically extract all the necessary information in the run-time from a user-provided communication interface, so that the users do not need to use or even know about the *YaRpcRequest* class at all. For more details, see the *YaRpcProxy* class in Section 2.4.2.

2.1.4. YaRpcResponse

The *YaRpcResponse* class has a symmetric role to the *YaRpcRequest* class – it contains information about a finished remote method call, such as the return values or an exception that has been thrown. All the properties of the *YaRpcResponse* class are listed in Table 5.

Property	**Description**
`Outputs`	The array of return values of a remote method call.
`Exception`	An exception thrown during the execution of the remote method

	call.
`Context`	A protocol-specific hash-map with additional information about the response.

Table 5 *YaRpcResponse* class properties

Although some languages only allow methods to return a single value (e.g., Java), the YaRpc middleware supports remote methods with multiple output values. In languages that support output or by-reference parameters, the output values can be mapped to those parameters. Again, users usually do not need to deal with the *YaRpcResponse* class directly, and can use convenience classes to issue and dispatch remote method calls with a little programming effort. See *YaRpcProxy* in Section 2.4.2 for more details.

2.1.5. IYaRpcTransport

Even though RPC protocols are usually associated with a specific network transport layer (e.g., SOAP is associated with HTTP or HTTPS) and in numerous RPC implementations the protocol and transport are tied closely together, the protocol and the transport layer can be considered as two separate entities. The RPC protocol, here represented using the *IYaRpcConnection* interface, is responsible for serialization of requests and responses to network messages, and maintaining a logical state between two communication end-points, whilst the functionality of a network transport layer, here encapsulated in the *IYaRpcTransport* interface, is responsible for establishing a physical link between two end-points, and sending and receiving messages over that link. In a wider picture, such a distinction between the protocol and transport allows us to reuse software components in various protocol implementations. For example, a HTTP implementation of the *IYaRpcTransport* interface can be used by both SOAP and XML-RPC implementation of the *IYaRpcConnection* interface. All the following network transport layers can easily be represented using the *IYaRpcTransport* interface: UDP, TCP, SSL, HTTP, HTTPS, named pipes, shared-memory, etc. Table 6 lists the methods of *IYaRpcTransport* interface.

Method	Description
`State`	Gets the state of the transport link (*Uninitialized, Connecting, Connected, Listening, Closed*).
`IsServer`	Indicates whether the network link has been initiated by the remote end-point (i.e. that the current end-point represents a server).
`Tag`	Gets or sets a user-provided tag associated with the transport.
`BeginConnect`	Asynchronously initiates a connection to a remote end-point, specified using a transport-specific address.
`EndConnect`	Finishes the asynchronous connection initiated previously using the *BeginConnect* method.
`BeginSend`	Asynchronously begins sending a message over the link. The message is stored in a *YaRpcSmartStream* class instance.
`EndSend`	Finishes an asynchronous sending operation previously initiated using the *BeginSend* method.
`BeginReceive`	Asynchronously begins receiving a message over the link. The received message is stored in a provided *YaRpcSmartStream* class instance.
`EndReceive`	Finishes an asynchronous receive operation previously initiated by the *BeginReceive* method.
`Close`	Closes the network link and releases associated resources.
`StartListening`	Starts listening for incoming connections, on a local end-point specified by a transport-specific address. Whenever an incoming connection is accepted, a user-provided callback is executed.
`StopListening`	Registers a user callback which will be called when the connection was closed.

Table 6 *IYaRpcTransport* interface methods

An implementation of *IYaRpcTransport* works in two distinct modes of operation: *connection* and *listening*. In the connection mode, the transport represents and open network link between two end-points that can be used to send or receive data. In the listening mode, the transport is listening for incoming connections in a network transport layer-specific way, and whenever a connection is accepted, a new instance

of an appropriate *IYaRpcTransport* implementation is created which represents the newly established link.

Clearly, the methods provided by the *IYaRpcTransport* interface resemble the Berkeley sockets interface [Ws97], and in many situations an *IYaRpcTransport* implementation will be just a thin wrapper above a socket (e.g., for UDP and TCP transports). However, for a special transport layers, such as shared-memory or named pipes, the *IYaRpcTransport* implementations will need to provide a more comprehensive functionality.

The *IYaRpcTransport* implementations should preferably use asynchronous capabilities of the underlying operating systems for their own asynchronous operations, e.g. I/O Completion Ports in Microsoft Windows, in order to reduce performance bottlenecks caused by poor threading model [Jon02]. However, if such asynchronous capabilities are not available on a particular platform, the implementations have to emulate them by spawning a thread to execute an underlying blocking I/O operation.

2.1.6. IYaRpcRemotable

The *IYaRpcRemotable* interface represents an object that is able to execute remote method calls (aka *remotable* object). Typically, a developer of a server application will register a bunch of remotable objects to the YaRpc infrastructure (using appropriate methods of the *YaRpcManager* class – see Section 2.1.1), and a protocol implementation will call their methods during a dispatch of a received remote method call (in an implementation of the *IYaRpcConnection* interface – see Section 2.1.2). The methods of the *IYaRpcRemotable* interface are described in Table 7.

Method	**Description**
`ObjectKey`	Gets an object key under which the remotable object is registered to the YaRpcManager. This property is necessary for RPC implementations that can serialize a reference to a remotable object.
`GetMethodDefinition`	Gets the definition of a particular method given its name. Among other things, the definition contains a list of the method input parameters and return values, including their names and

	types.
`ExecuteRequest`	Synchronously executes a specific remote method call, using a list of parameters provided.
`BeginExecuteRequest`	Begins asynchronous execution of a remote method call. This method is a non-blocking variant of the *ExecuteRequest* method.
`EndExecuteRequest`	Finishes the asynchronous execution of a remote method call initiated previously using the *BeginExecuteRequest* method.

Table 7 *IYaRpcRemotable* interface methods

A remotable object has a role similar to the *stub* (sometimes called *skeleton*), a piece of code generated from an interface description language (IDL) in traditional RPC protocols such as CORBA or DCOM [Dcom98]. The *IYaRpcRemotable* interface does, however, provide more flexibility to RPC implementations compared to the stubs – implementations in high-level programming languages such as Java or C# can take advantage of code reflection and provide an appropriate implementation of the *IYaRpcRemotable* interface automatically in the run-time, while an implementation in a low-level language such as C/C++ can still use IDL to automatically generate the stubs that will implement the *IYaRpcRemotable* interface, in the development time. See the description of the *YaRpcStub* class in Section 2.4.1 for more details on how to use code reflection to aid creation of remotable objects.

Remotable objects need to provide a definition of the remotable methods they support, including the name and type of its parameters and return values, using the *GetMethodDefintion* method. The information provided might (or might not) be used by a particular RPC protocol implementation for serialization and deserialization of requests and responses, and to execute the requests in a right way. Among other things, the *GetMethodDefintion* method definition provides a worker object that should be used to execute the request. This functionality enables server applications to only use one thread to serve all requests on a specific remotable object, for instance. For more details on the execution workers, please see the description of the *IYaRpcWorker* interface in Section 2.1.7.

The YaRpc infrastructure enables remotable objects to execute the remote method call request either synchronously or asynchronously, based on their preference.

The information whether a request should be executed synchronously or asynchronously is also provided by the *GetMethodDefintion* method. Asynchronous execution of request is especially useful due to runtime performance reasons. For example, a server accessing files on a disk might prefer to provide asynchronous remote methods so that the dispatching thread will not be blocked while waiting for the file access to finish and thus the thread can perform some more interesting tasks. The fact, that remotable objects can provide asynchronous remotable methods, has important implications for the scalability of server applications. In YaRpc, this behavior is called *asynchronous method dispatch*.

The asynchronous method dispatch can be also advantageous to high-performance server applications that only forward method calls between a number of clients. When a remote call is received, the server's remotable object can asynchronously issue a remote call to another client, and this initiation is likely to be executed before a context switch occurs (possibly on an I/O Completion Port thread), thus providing an enormous scalability potential.

2.1.7. IYaRpcWorker

The *IYaRpcWorker* interface represents a worker object that can "physically" execute a remote method call request. For example, the requests can be executed on a thread pool, or on a single-thread queue. The *IYaRpcWorker* interface gives *IYaRpcRemotable* objects a flexibility to define when and where each dispatched request should be executed. Table 8 shows a single method of the *IYaRpcWorker* interface.

Method	**Description**
`QueueWorkItem`	Schedules a method delegate for an execution, with an implementation-specific semantics.

Table 8 *IYaRpcWorker* interface methods

2.2. Protocol Plug-ins

Software developers are encouraged to implementation plug-ins in order to add support of RPC protocols of their choice to the YaRpc middleware. A protocol plug-in is a combination of an implementation of the *IYaRpcConnection* interface (see Section 2.1.2) and an implementation of the *IYaRpcTransport* interface (see Section 2.1.5).

A developer registers such a combination to the YaRpc infrastructure using the *YaRpcManager.RegisterProtocol* method, under a specific protocol name. When a user is connecting to a remote end-point using the *YaRpcManager.Connect* method (or the *BeginConnect* method), he or she has to provide an URL of the remote end-point. The appropriate protocol is selected based on the URL scheme prefix.

For example, in order to support web-services, a user might register the *YaRpcConnectionSoap* and *YaRpcTransportHttp* types under "`soap`" scheme prefix. Whenever a connection attempt is made to an URL such as:

```
soap://www.hostname.com/protocol/specific/part
```

then the YaRpc infrastructure will automatically perform the following steps:

1. Identify the connection-transport pair based on the "`soap`" scheme prefix.
2. Create a new instance of the *YaRpcTransportHttp* class.
3. Invoke *YaRpcTransportHttp.BeginConnect* in order to open a network link to `www.hostname.com`.
4. Create a new instance of the *YaRpcConnectionSoap* class.
5. Invoke *YaRpcConnectionSoap.BeginIntialize* to associate the connection with the established network link and to let the connection perform any necessary initialization steps.
6. Returns the instance of the *YaRpcConnectionSoap* class back to the user.

Note that during the connection, the full original URL is provided to both *IYaRpcTransport* and *IYaRpcConnection* implementations, thus enabling them to use other parts of the URL (i.e., hostname, port number, path and query) for an implementation-specific purpose. Typically, different RPC protocols share a common transport layer (e.g., both SOAP and XML-RPC use HTTP), so that a protocol developer might not need to implement the *IYaRpcTransport* interface at all, and he or she can just use an implementation that is already available.

2.3. Serialization

The serialization is a process of encoding of a remote method call request or response, including all the input or output parameters, into a data message that can be transferred over a network link. The deserialization is a process symmetric to the serialization – it decodes a data message received over the network and creates an appropriate request or response object. Even though the serialization is RPC protocol-specific, the YaRpc middleware provides a several supporting classes to help particular protocol implementers, i.e. the implementers of the *IYaRpcConnection* protocol as described in Section 2.1.2.

2.3.1. YaRpcSmartStream

The *YaRpcSmartStream* class serves as a storage of bytes that grows automatically as more data is added to it. The stream supports random access read and write – a user can seek to any position and read or write data to that position. Unlike similar stream implementations available in many programming languages, the interface of the *YaRpcSmartStream* class is designed in a way that enables users to save unnecessary re-copying of byte buffers when reading or writing to the stream, which leads to a higher runtime performance of operations with the stream. Additionally, *YaRpcSmartStream* has methods that allow users to impose a limit on a read position from the stream. This functionality might be used when deserializing object-trees in particular protocol implementations, in order to prevent poorly written custom serialization or deserialization routines to fail the whole deserialization process. Table 9 lists the methods of the *YaRpcSmartStream* class.

Method	**Description**
`Length`	Gets or sets the total length of the stream in bytes.
`Tag`	Gets or sets a user-provided tag associated with the stream.
`Clear`	Resets the stream to the initial state, with 0 bytes stored.
`BuffersCount`	Gets the number of buffers used by the storage.
`Write_Position`	Gets or sets the current write position in the stream.

`Write_Flush`	Updates the total length of the stream to reflect the most recent writes.
`Write_GetBlock`	Gets a continuous chunk of memory which may be used to write the data directly to the stream, without any additional copying.
`Write_FinalizeBlock`	Finalizes a write operation started previously using the *Write_GetBlock* method.
`Write_PutBytes`	Writes a data to the stream.
`Write_PutByte`	Writes a single byte to the stream.
`Write_FromStream`	Writes a part of another *YaRpcSmartStream* to the current stream.
`Read_Position`	Gets or sets the current read position in the stream.
`Read_GetBlock`	Gets a continuos block of memory from which the stream data may be read directly, without any additional copying.
`Read_FinalizeBlock`	Finalizes the read operation started previosuly using the *Read_GetBlock* method.
`Read_GetBytes`	Copies data from the stream to a byte buffer.
`Read_GetByte`	Reads a single byte from the stream.
`Read_RemainingBytes`	Gets the maximum number of bytes that can be read from the stream, starting from the current read position.
`Read_EnsureRemainingBytes`	Checks that a specific number of bytes may be read from the stream and throws an exception if not.
`Read_PushLimit`	Imposes a lower and upper bound for the read position.
`Read_PopLimit`	Removes the top-most read-position limit imposed using the *Read_PushLimit* method.
`Read_Compare`	Reads chunk of data from the stream and compares it to a specified buffer.

Table 9 *YaRpcSmartStream* class methods

Typically, growing memory streams are implemented using a single byte buffer (i.e., an array); if the buffer is full, the stream allocates a new buffer twice as long and copies the data from the previous buffer to the beginning of the new buffer. This

approach is easy to implement, however, it does unnecessary copying of buffers and limits the maximum size of the data stored to the maximum length of an array available on the particular platform (e.g., 2 GB in Java and .NET Framework [Java06, Net07]). Several platforms and operating system, such as .NET Framework running on Microsoft Windows, allow to use a list of buffers when sending data or receiving data from a socket, instead of a single buffer. The use of this API, which is also known as *scatter/gather I/O* [Mwgs07], is typically more efficient then copying buffers into a single buffer before the I/O operation is invoked. Particular YaRpc middleware implementations are highly encouraged to internally implement the *YaRpcSmartStream* storage using a list of buffers, in case the underlying platform supports the scatter/gather I/O. Whenever the existing buffers in a *YaRpcSmartStream* are full, a new larger buffer is allocated and appended to the list of buffers. If the new buffer is twice as large as the previous buffer, the stream can theoretically save up to ca. 2^{60} bytes of data. This capability is important in situations where applications want to use network messages larger than 2 GB; even though at the current day and age there will not be many such applications, we assume that with the evolution of many-Gigabit optic fiber network links such applications will arise.

2.3.2. YaRpcWriter

The *YaRpcSmartStream* interface on its own is not very convenient, because it only allows writing of byte or byte array. The RPC implementations, or custom user serialization routines, often need means to serialize common data-types to a stream. The *YaRpcWriter* class allows writing of all or most of the common data types of a particular language to an instance of *YaRpcSmartStream*. The *YaRpcWriter* is flexible in the sense that it allows users to specify byte ordering for integer data type (little-endian or big-endian [Dce80]), and encoding used for string serialization. The methods of the *YaRpcWriter* class are described in Table 10.

Method	**Description**
`Stream`	Gets the underlying instance of the *YaRpcSmartStream* class.
`IsBigEndian`	Indicates whether integer data types are serialized using little-endian or big-endian byte order.

`Encoding`	Gets the encoding used to serialize text strings. The default encoding is UTF-8.
`Write`	Writes a common data type to the stream.
`WriteString`	Writes a string to the stream, prefixed with a Z-encoded length of the string, or -1 if the string is null.
`WriteArray`	Writes a (multidimensional) array to the stream. Each array is prefixed with a Z-encoded length of the array, or -1 if the array is null.
`WriteRaw`	Writes a string or an array to the stream, without any prefix.
`WriteMagic`	A convenience method that writes a 16-bit or 32-bit magic number to the stream.
`WriteZeroBytes`	Writes a number of zero bytes to the stream.
`WriteZetCode`	Writes a Z-encoded integer to the stream.
`Align`	Advanced the write position in the stream to the nearest multiple of specific number.
`Flush`	Invokes the *Write_Flush* method on the underlaying stream.

Table 10 *YaRpcWriter* class methods

The *YaRpcWriter* uses Z-encoding to write the length prefix for a string or an array. The Z-encoding is an efficient way how to serialize an arbitrarily large integer number. For more details, see Section 2.3.4.

2.3.3. YaRpcReader

The *YaRpcReader* has a role symmetric to the *YaRpcWriter* – it reads common data types from an underlying *YaRpcSmartStream*. The *YaRpcReader* class also enables users to choose the byte ordering and string encoding. Table 11 lists the method of the *YaRpcReader* class.

Method	**Description**
`Stream`	Gets the underlying instance of the *YaRpcSmartStream* class.
`IsBigEndian`	Indicates whether integer data types are assumed to be serialized using little-endian or big-endian byte order.
`Encoding`	Gets the encoding used to deserialize text strings. The default

	encoding is UTF-8.
`ReadXXX`	Reads a common data type XXX from the stream.
`ReadString`	Reads a string from the stream. The string is prefixed with a Z-encoded length of the string.
`ReadArray`	Reads an array from the stream. The array is prefixed with a Z-encoded length of the array.
`ReadRaw`	Reads a non-length-prefixed string or array from the stream.
`ConsumeMagic`	A convenience method that reads a 16-bit or 32-bit magic number from the stream and throws an exception if it does not match.
`ReadZetCodeAsInt32`	Reads a Z-encoded integer as a 32-bit integer.
`ReadZetCodeAsInt64`	Reads a Z-encoded integer as a 64-bit integer.
`Align`	Advanced the read position in the stream to the nearest multiple of specific number.

Table 11 *YaRpcReader* class methods

2.3.4. Z-Encoding

Z-encoding is a new algorithm that we have developed for serialization of an arbitrarily large integer value in a storage-efficient way. In *YaRpcWriter* and *YaRpcReader*, Z-encoding is widely used to serialize lengths of a strings or arrays. The main assumption is that numbers close to zero will occur more frequently than numbers far from zero, thus they should not need too much storage space. Additionally, the encoding needs to be byte-oriented and also support negative numbers. This led us to a definition of Z-encoding as follows:

1. An integer value v is transformed into v' using the following formula:

   $v' = 2v$ (if $v >= 0$)

   or $v' = -2v - 1$ (if $v < 0$)

   For example, 0 gets transformed to 0, -1 to 1, 1 to 2, -2 to 3, 2 to 4, etc. Note that v' is always a non-negative value.
2. The bits of the transformed integer value v' are split into a sequence of 7-bit chunks, starting from the least-significant bits of the value.

3. Each 7-bit chunk is bitwise ORed with a byte value where the most significant bit set if the chunk is not the last one in the sequence, and unset if the byte is last. The whole byte is written to the stream.

The reading of a Z-encoded value is symmetrical to the writing:

1. Read bytes from the stream, and stop when the last byte read has the most-significant bit unset.
2. Concatenate the lower 7 bit chunks from each byte read into a single integer value *v'*, in the least-to-most-significant bit order.
3. Compute the original value *v* using the following formula:

$$v = v' / 2 \qquad \text{(if } v \text{ is even)}$$

or

$$v = -1 - v' / 2 \qquad \text{(if } v \text{ is odd)}$$

The name "Z-encoding" is appropriate for a number of reasons: the transformation of integer values resembles a zigzag pattern or a simply the letter "Z", a mathematical symbol for integer numbers is ℤ, and the letter "Z" also resembles the number 7, which is the length of each bit chunk. The reader is free to choose the most appropriate explanation.

2.4. Utility Services

The YaRpc middleware provides several classes to help both users of the middleware and implementers of RPC protocols. However, these helper classes are not a mandatory part of the YaRpc specification, and might not be available on all platforms, because they might depend on specific features of programming languages.

2.4.1. YaRpcStub

The *YaRpcStub* class is used to simplify creation of remotable objects by developers of server applications. The class implements the *IYaRpcRemotable* interface, and uses code reflection to obtain the signature of methods of a user object that are to be made remotable. The user only needs to mark which methods are that, using for example code meta-data if the programming language supports it, and also the user needs to link the user object with a particular *YaRpcStub* instance, which can conveniently be done using a generic type argument. For example, in .NET Framework the *YaRpcStub*

implementation uses the *YaRpcMethodAttribute* meta-data attribute and a generic type argument for this purpose, respectively, while Java implementation uses the *YaRpcMethod* annotation and an instance reference, respectively. See Section 3 for more details on particular implementations. Table 12 lists methods of the *YaRpcStub* class; the members inherited from the *IYaRpcRemotable* are skipped.

Method	**Description**
`RegisterToYaRpc`	Registers a user object to the YaRpc infrastructure to receive remote method calls, under a specific object key.
`UnregisterFromYaRpc`	Unregister the user object from the YaRpc infrastructure.
`Worker`	Gets or sets an implementation of the IYaRpcWorker interface that should be used to execute the method calls.

Table 12 *YaRpcStub* class methods

2.4.2. YaRpcProxy

The *YaRpcProxy* class simplifies development of client applications by either providing convenience methods that simplify issuing of remote method calls, or even providing methods that can dynamically generate a *proxy* object based on an interface, using a runtime code generation capability of a programming language if it is available. The *YaRpcProxy* class usually provides methods listed in Table 13.

Method	**Description**
`YaRpcTimeOut`	Gets or sets the timeout for remote calls issued from the proxy object.
`YaRpcConnection`	Gets the *IYaRpcConnection* associated with the proxy object.
`YaRpcObjectKey`	Gets the object key of the remote object this proxy is associated with.
`Generate`	Dynamically generates an instance of a proxy object, based on a provided communication interface. This method might not be available on every platform.
`YaRpcRemoteCall`	A convenience method that constructs a *YaRpcRequest* instance and invokes the remote call using the *YaRpcManager* class.

Table 13 *YaRpcProxy* class methods

2.4.3. YaRpcConnectionBase

Even though the YaRpc specification already separates RPC protocols from the physical network transport layer (see Sections 2.1.2 and 2.1.5), many RPC protocol implementations will still share a lot of common functionality, such as remote call dispatching, connection management or response processing. The *YaRpcConnectionBase* is an abstract class that aims to provide these common services to particular *IYaRpcConnection* implementations. In fact, new protocols will usually only need to override the abstract methods of the *YaRpcConnectionBase* class, which are listed in Table 14.

Method	**Description**
`BeginHandshake`	Asynchronously starts a hand-shake with the remote end-point to initialize the connection.
`SerializeMessage`	Serializes a request or response message.
`ControlMessageReceiving`	Manages the receiving of network message from the remote end-point.
`DeserializeMessage`	Deserializes the header of a received network message.
`DeserializeMessageBottomHalf`	Finishes deserialization of a received network message.

Table 14 Abstract methods of the *YaRpcConnectionBase* class

2.5. YaRpc Native Protocol (YNP)

Although the YaRpc middleware is designed to support a number of existing protocols, it also defines a new light-weight *YaRpc Native Protocol (YNP)* that offers a unique set of features compared to existing RPC protocols, such as high performance due to a binary message format, server-initiated callbacks, customizable serialization, asymmetric serialization and nested method calls. The YNP uses TCP or TCP+SSL as the underlying network transport, in order to provide a reliable data transfer [Rfc793] and encryption [Ssl94], respectively. YNP was purposely designed as a minimalistic protocol with a simple message format, in order to promote creation of YNP implementations for other platforms and operating systems.

2.5.1. Message Format

The YNP is a message oriented RPC protocol; both remote call request and response are packed into a single network message. In case a remote method call failed, the remote side sends back a message describing the exception that occurred. Each request, and corresponding response or the exception message, is assigned a random generated 16-byte globally unique identifier (GUID). The messages have a binary format, in order to save the network bandwidth and enable fast serialization and deserialization. Table 15 shows the format of an YNP message header.

Field	**Bytes**	**Description**
Magic	3	A protocol magic number: an ASCII-encoded “YAR” string.
Protocol version	1	A number identifying the protocol version, current version is 0x02.
Message type	1	The type of the message (0x01=REQUEST, 0x02=RESPONSE, 0x04=EXCEPTION)
Message flags	1	OR-combination of message flags (0x01=BIG_ENDIAN)
Specific flags	1	OR-combination of request or response flags, depending on the message type.
Reserved	1	A field reserved for a future use, must be zero when sending and is to be ignored when receiving messages.
Body size	8	The length of the message body in bytes, encoded using a message's byte ordering.
Method call GUID	16	A globally unique identifier of the remote call.

Table 15 YNP protocol message header

The message header is followed by a body which is specific to the *message type*. The REQUEST, RESPONSE and EXCEPTION message body is described in Table 16, Table 17 and Table 18, respectively.

Field	**Bytes**	**Description**
Nested to GUID	16	A unique identifier of remote call to nest to – see Section 2.5.4.
Object key	?	The name of the dispatching remotable object (Z-encoded length + UTF-8 encoded string).

Method key	?	The name of the remotable object method (Z-encoded length + UTF-8 encoded string).
Context data	?	Additional invocation-specific information, specified as a list of string key-value pairs.
Argument count	?	A Z-encoded number of input parameters of the remote call.
Arguments	?	A sequence of input parameters serialized using the YNP serialization scheme – see Section 2.5.2.

Table 16 YNP protocol REQUEST message body

Field	**Bytes**	**Description**
Context data	?	Additional invocation-specific information, specified as a list of string key-value pairs.
Argument count	?	A Z-encoded number of output parameters of the remote call.
Arguments	?	A sequence of output parameters serialized using the YNP serialization scheme – see Section 2.5.2.

Table 17 YNP protocol RESPONSE message body

Field	**Bytes**	**Description**
Context data	?	Additional invocation-specific information, specified as a list of string key-value pairs.
Fault code	?	A Z-encoded error code, which may be used by a user code.
Exception type	?	A name of the type implementing the exception (7-bit encoded length + UTF-8 encoded string)
Message	?	A text with description of the exception (7-bit encoded length + UTF-8 encoded string).
Stack trace	?	A text with stack trace of the exception (7-bit encoded length + UTF-8 encoded string)
Data records	?	A table of key-value pairs with additional exception information.

Table 18 YNP protocol EXCEPTION message body

2.5.2. Serialization

The REQUEST and RESPONSE message contains a sequence of input and output parameters of the remote call, respectively. In the stream, each parameter is prefixed with a 1 byte signature. The signature is a bit-field that describes the type of the serialized value; the meaning of each bit is described in Table 19.

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Value	Is null	Type code				Array dimension		

Table 19 Serialized value signature byte

If the serialized value is null, the bit 7 will be set, otherwise it will be unset. For reference types, the meaning of a null value is clear, but for value types the meaning might be problematic (e.g., consider *int* type in Java). The implementations are free to choose the "best" way, e.g. an implementation in .NET Framework might use nullable types to represent a null value-type [Net07], while a Java implementation might simply use the default value of a value-type, i.e., zero. Bits 3-6 describe the type of the serialized value; all the possible type codes are listed in Table 20.

Type code	**Value**	**Description**
BackReference	0x00	The object has already been serialized/deserialized, and this is just another reference to it.
Boolean	0x01	A boolean value (true of false, 8-bit).
Byte	0x02	8-bit integer value
Int16	0x03	16-bit integer value
Int32	0x04	32-bit integer value
Int64	0x05	64-bit integer value
Single	0x06	A single-precision IEEE 754 floating point number (32-bit)
Double	0x07	A double-precision IEEE 754 floating point number (64-bit)
Decimal	0x08	A 64-bit IEEE 754 decimal.
DateTime	0x09	Date & time value, represented as number of 100-nanosecond intervals that have elapsed since 12:00:00 midnight, January 1, 0001, encoded as a 64-bit integer.
TimeSpan	0x0a	A difference between two time instants, represented as number of

		100-nanosecond intervals, encoded as a 64-bit integer.
Guid	0x0b	A 16-byte globally unique identifier (GUID)
Char	0x0c	A UTF-8 encoded character (1-6 bytes per character)
ComplexType	0x0d	A complex data-type.

Table 20 Signature type code

The type code is followed by a data block in the stream, and the format of the data differs based on the signature. For simple values (i.e. all type codes except of *BackReference* and *ComplexType*, and with array dimension zero), the signature type code is simply followed by the value, serialized using current session's byte-ordering.

The *BackReference* code is a special type code that enables serialization of object-graphs. If an object (e.g. array, string or complex type) has already been serialized/deserialzied in a same serialization/deserialization session, there is no need to write or read it again, respectively. The *BackReference* signature code is followed by a Z-encoded relative offset in the stream where the original object has been serialized. Note that if *BackReference* code is used, bits 7 and 0-2 must always be unset.

The *ComplexType* code is used to specify that the data have been serialized by a complex user-provided object. The *ComplexType* code signature is followed by a 64-bit integer storing a total length of the data, and then by the data stream as encoded by the object. A user on the receiving side is responsible to provide an object (more precisely a type) that is able to deserialized the data.

Bits 0-2 of the signature store the number of dimensions of an array. This is used to encode (multidimensional) arrays of simple or complex data types. The signature byte is followed in the stream by a Z-encoded length of the array, and then a sequence of serialized elements of the array, each encoded the same way as if it were the only element (e.g. for a *ComplexType* code, each element of the array is prefixed with 64-bit byte length). Multidimensional arrays are encoded as array of arrays, recursively. Each sub-array is encoded as a one-dimensional array, including its own full 1-byte signature. This is useful e.g. in case a sub-array has already been serialized, so that only *BackReference* signature and offset might be written instead. Strings are serialized as a single-dimensional array of characters, where each character is encoded using UTF-8 encoding.

The *YaRpcNativeSerializer* class encapsulates all the serialization functionality of the YNP protocol. This makes it possible for user to use the YNP serialization scheme e.g. for serialization of sub-objects of a complex data type. The methods of the *YaRpcNativeSerializer* class are listed in Table 21.

Method	**Description**
`RegisterSerializationSurrogate`	Registers a custom user serializer /deserialzier to the chain of serializers.
`UnregisterSerializationSurrogate`	Unregisters a custom serializer from the chain.
`GetSurrogates`	Gets the chain of serializers.
`Serialize`	Serializes a value using the first serializer in the chain that is able to do it.
`Deserialize`	Deserializes a value from a stream using the first deserializer in the chain that is able to do it.

Table 21 The *YaRpcNativeSerializer* class methods

Note that the *RegisterSerializationSurrogate* method enables users to completely override the default YNP serialization behavior, and thus provide an important feature of the YNP protocol – *custom serialization*.

2.5.3. Asymmetric Serialization

Note that there is no type code to distinguish between signed and unsigned integer values, a string is considered equivalent to an array of chars, and there is no check to ensure a complex object that has been serialized on one side is being deserialized by the same complex type on the other side. This ambiguity, which we call *asymmetric serialization*, however, enables clients and servers to use slightly different communication interfaces, while still enabling them to communicate seamlessly. This can potentially lead to a higher efficiency, as the end-points do not need to convert data to a required form after the deserialization.

2.5.4. Nested Method Calls

Consider a situation where a client synchronously issues a remote call to a server and during the execution of that remote call the server issues a callback to the client (aka

nested call). For a number of reasons, it is desirable that the remote callback is executed on the same thread that is blocked while waiting for the result of the original remote call (aka *causal call*). Such a behavior can not only improve runtime performance by utilizing an otherwise blocked thread, but it can also be crucial for applications that need to guarantee that certain activities are always executed on a single thread, such as update of a graphical user interface (GUI) or an invocation of COM components that use the single-thread apartment (STA) model on Microsoft Windows [Win07].

In order to support execution of nested method calls on a causal thread, the YNP REQUEST message body contains a globally-unique identifier of the remote call to nest to. Also, the YNP implementations need to determine that a remote method call was invoked during execution of another remote call. Fortunately, most programming languages and/or operating systems provide means to save certain information on per-thread basis, for example Microsoft Windows provide the *Thread Local Storage* which is accessible using C#'s *[ThreadStatic]* attribute [Net07, Win07].

A more complicated situation arises when more than two communication end-points are involved. For example, consider the communication between three end-points as depicted in the sequence diagram in Figure 10. When issuing callback GUID4 from the server to client A, the server needs to determine that the causal call is the call GUID1, and not the call GUID3. In order to do that, the YNP implementations need to keep track of causality between the method calls, which basically means to maintain a stack of call-connection pairs that can be used to answer queries such as "what is the top-most call from the client X, that caused the execution of the current request".

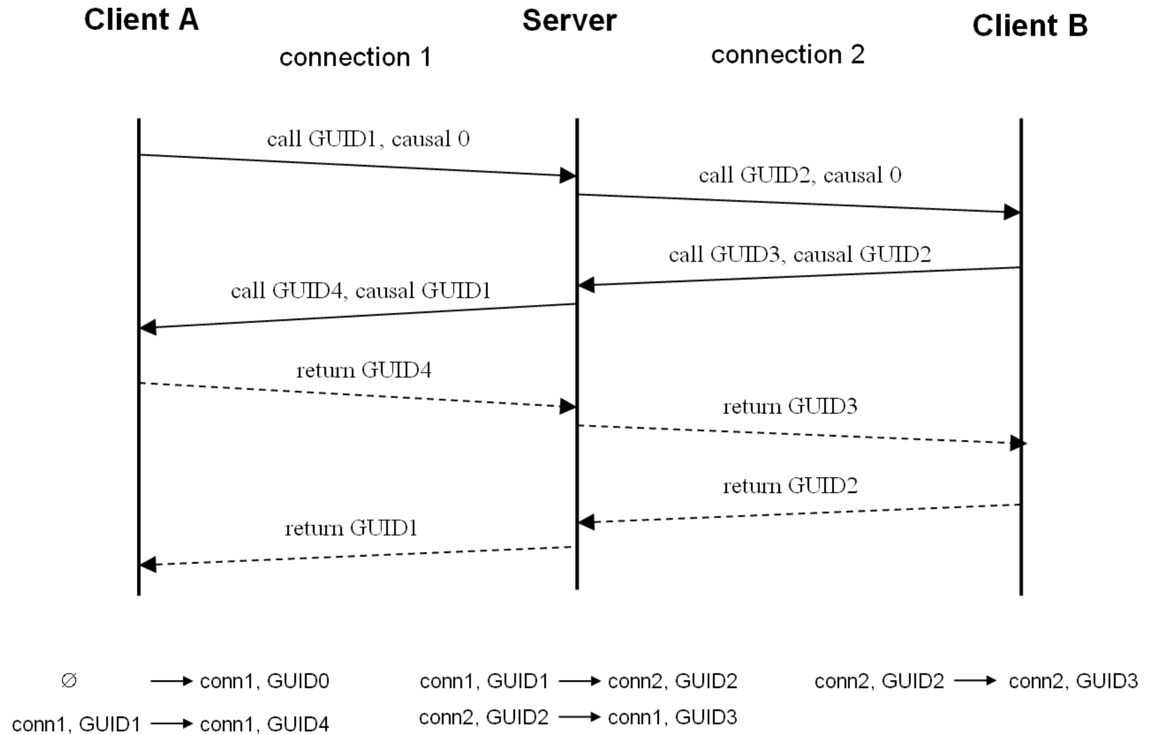

Figure 10 Causality of remote method calls in YNP (Source: [Jc07])

Note that the concept of method call nesting as described here has already been available in the prototype of YaRpc that we used as a starting point of our efforts. As such, it has not been developed as part of this thesis, and we only provide its description here for the sake of completeness.

3. Implementation

Even though the YaRpc middleware specification defines precisely which interfaces, classes and methods should the YaRpc implementations support, and what functionality they should provide, the specification is flexible in order to support a wide range of platforms and programming languages. Different programming languages provide different syntax and run-time features. With respect to the syntax features, languages without support of output or by-reference parameters might need to define a handful of additional objects to be able to return more complex values from a method call, or languages without support of object properties will need to define getter and setter methods, for example. Generally speaking, the variances in syntax features between the languages usually will not be a problem; however, unavailability of certain runtime features, such as the code reflection or dynamic code generation, might seriously affect implementations and potentially force them to redesign certain functionalities.

As a proof of concept, we have implemented the YaRpc infrastructure and YaRpc Native Protocol (YNP) in both Java 6 and .NET Framework 2.0 (with C#) programming languages. In this section, we describe some of the details of the two implementations and how peculiarities of the corresponding platforms can be tackled.

3.1. .NET Framework 2.0

The Microsoft .NET Framework 2.0 with C# language was the initial platform used for prototyping of the whole YaRpc specification, so naturally the first implementation was available in C#. The C# language is a modern high-level programming language with a wide range of features (e.g., events, delegates, object properties, generics) that stream-line the development and enable creation of clean interfaces. Additionally, the standard system libraries provide advanced APIs which take advantage of the host operating system capabilities (e.g. asynchronous I/O, vectored I/O, advanced threading and synchronization).

One of the design goals of YaRpc for .NET Framework was the compatibility – the library only uses standard APIs, does not use P/Invoke [Msdn] and is written in

version 2.0 of the language, so that it might be used on a range of operating systems, including Linux with MONO framework [Mono].

A first step when creating a distributed application is the definition of a communication interface, which is known by both the client and server. Figure 11 shows an example of such an interface. The methods of the interface that are to be made remotable are marked using the *YaRpcMethodAttribute* meta-data attribute, which is then used by both *YaRpcProxy* and *YaRpcStub* classes, as we will see later.

```
public interface IHelloWorld
{
  [YaRpcMethod]
  string GetGreeting();
}
```

Figure 11 A communication interface, shared by both client and server

With C#, the *YaRpcStub* class can be extended with a generic type argument, and a user object dispatching the remote calls can be inherited from such a class. The *YaRpcStub<T>* class implements the *IYaRpcRemotbale* interface, and uses the runtime type reflection to identify which methods of type *T* are to be made remotable, based on the presence of the *YaRpcMethodAttribute* attribute. This leads to a clean and straightforward design pattern for development of the server applications, such as the example in Figure 12.

```
public class HelloWorldStub : YaRpcStub<IHelloWorld>, IHelloWorld
{
   public HelloWorldStub( string objectKey )
    : base( objectKey )
   {
   }

   public string GetGreeting()
   {
     return "Hello world";
   }
}

...

YaRpcManager.StartListening( "ynp://0.0.0.0" );
HelloWorldStub stub = new HelloWorldStub( "key" );
stub.RegisterToYaRpc();
```

Figure 12 An implementation of a server in .NET

Similarly, the *YaRpcProxy* class uses a generic type argument to associate the proxy object with a user-provided communication interface. In a first (simpler) variant, the *YaRpcProxy* uses the type information to dynamically find signatures of the user-provided communication interface methods (again, marked with the *YaRpcMethodAttribute* attribute), and uses them to help the user with issuing remote method calls by aiding with creation of a *YaRpcRequest*, and handling the returned *YaRpcResponse,* via the *YaRpcRemoteCall* method – see example in Figure 13.

```
public class HelloWorldProxy : YaRpcProxy<IHelloWorld>, IHelloWorld
{
  public HelloWorldProxy( IYaRpcConnection conn, string objectKey )
    : base( conn, objectKey )
   {
   }

   public string GetGreeting()
   {
     return (string) YaRpcRemoteCall( "GetGreeting" );
   }
}

...

IYaRpcConnection conn = YaRpcManager.Connect( "ynp://127.0.0.1" );
IHelloWorld proxy = new HelloWorldProxy( conn, "key" );
string greeting = proxy.GetGreeting();
```

Figure 13   Client code, with a manually created proxy in .NET

Microsoft .NET Framework enables dynamic class code generation in the runtime, by emitting the code in form of Microsoft Intermediate Language (MSIL) using the libraries provided by the *System.Reflection.Emit* namespace [Msil00]. A more sophisticated *YaRpcProxy* implementation can use this capability to generate and instantiate a type that implements the user-provided communication interface, with a functionality equivalent to the manually created proxy object. The runtime proxy generation greatly simplifies development of client applications, as shown in the code example in Figure 14. Note that the dynamic proxy generation is not part of the current YaRpc .NET implementation, and will be developed in the future.

```
IYaRpcConnection conn = YaRpcManager.Connect( "ynp://host.com" );
IHelloWorld proxy = YaRpcProxy<IHelloWorld>.Generate( conn, "key" );
string greeting = proxy.GetGreeting();
```

Figure 14 Client code, with a dynamically generated proxy in .NET

The YaRpc .NET implementation is available as a stand-alone assembly called *YaRpc.dll* which does not depend on any other third-party assembly, in order to provide maximum portability and ease of use.

3.2. Java 6

The Java Platform Standard Edition (Java SE) version 6 was the second platform selected for implementation of the YaRpc specification. Java 6 is a very popular and widely adopted software platform, probably due to the fact that it offers a powerful and easy-to-use high-level programming language, a comprehensive set of libraries, and, most importantly, a true portability which enables Java applications to run on operating systems such as Microsoft Windows, Linux, and Mac OS X, without any modification. Compared to the previous versions, Java 6 also offers several language features that enable creation of convenient programming interfaces to the YaRpc middleware, e.g. annotations and generics. An important goal of the Java 6 implementation was to provide a YaRpc programming interface that will be similar to the interface of the .NET implementation as much as possible. In fact, the .NET implementation served as a prototype during development of the Java version, and the only differences between the two implementations usually stemmed from the differences in the language features and libraries available. Another goal of the Java implementation was a maximum portability – for this reason, we chose to only use the standard libraries provided by the Java Platform Standard Edition (*java.** packages only).

Java does not provide any standardized way how to implement asynchronous operations, such as .NET Framework does (i.e., *BeginXXX* and *EndXXX* method pair, a delegate used to supply a callback method, and the *IAsyncResult* interface as a descriptor of the asynchronous operation). We chose to adopt an equivalent model also in the Java implementation. The main issue is that Java does not support method delegates directly, thus user-provided method callbacks need to be supplied using

interfaces, which makes the code a little cumbersome. Figure 15 shows an example of an interface that emulates a method delegate.

```
public interface IAsyncCallback {
    void finished( IAsyncResult asyncResult );
}
```

Figure 15 A method delegate in Java

The *IAsyncResult* interface represents a token to an asynchronous operation, which can also hold a user-provided state object and be used to determine whether the asynchronous operation finished. Figure 16 shows methods of the *IAsyncResult* interface in Java.

```
public interface IAsyncResult {
    Object getAsyncState();
    boolean isCompleted();
    boolean isCompletedSynchronously();
}
```

Figure 16 A token to an asynchronous operation in Java

An example of implementation of an asynchronous operation - acceptance of a connection over a socket - is shown in Figure 17. A user invokes the asynchronous operation using the *beginAccept* method, where he or she provides an implementation of *IAsyncCallback* interface that represents a callback that will be called after the operation finished, and an arbitrary user-defined object. The *beginAccept* method returns an instance implementing *IAsyncResult* interface. After the asynchronous operation is finished, the implementation invokes the user callback, and the user invokes the *endAccept* method, passing it the *IAsyncResult* token, in order to get information about the result of the asynchronous operation, e.g. to obtain the newly accepted socket as in this example.

```
private class AcceptingThread extends Thread {

    YaRpcAsyncResult _asynResult;

    ...

    @Override
    public void run()
    {
        Socket newSocket = null;
        Exception exp = null;

        try {
            newSocket = _socket.accept();
        }
        catch ( IOException ex ) {
            exp = ex;
        }
        finally {
            YaRpcAsyncResult sar =
               YaRpcAsyncResult.getFrom( _asynResult );
            sar.setTag( newSocket );
            if ( exp != null )
                sar.failed( exp );
            else
                sar.completed();
        }
    }
}

...

public IAsyncResult beginAccept( IAsyncCallback cb, Object state )
  throws IOException
{
    YaRpcAsyncResult asynResult = new YaRpcAsyncResult( cb, state );
    AcceptingThread at = new AcceptingThread( this, asynResult );
    at.run();
    return asynResult;
}

...

public Socket endAccept( IAsyncResult asyncResult ) throws Exception
{
    YaRpcAsyncResult sar = YaRpcAsyncResult.getFrom( asyncResult );
    sar.end();

    return (Socket) sar.getTag();
}
```

Figure 17 Implementation of an asynchronous operation in Java

One could ask why YaRpc provides an asynchronous interface to operations that are internally implemented using separate threads. The reason is that the internal implementation might be transparently changed in the future in order to support some asynchronous I/O interfaces provided by the underlying Java platform and operating system, and thus enable users to improve performance and scalability of their application without any change in the code. Asynchronous I/O operations will be available in Java 7, which is announced to be released in few months, at the time of writing of this thesis [Jsr203].

Similarly to the .NET implementation, the *@YaRpcMethod* annotation is used to mark the methods that are to be made remotable. Figure 18 shows an example of a communication interface shared between the client and server.

```
public interface IHelloWorld {

    @YaRpcMethod
    String getGreeting();
}
```

Figure 18 Definition of a communication interface in Java

Although Java provides support of generic type arguments, unlike in C#, the generics in Java are just a language sugar and the generic type information is not available in the runtime. However, the *YaRpcStub* implementation can obtain the current object type anyway, and uses it to inspect the remotable methods. An example of a server implementation is shown in Figure 19.

```
public class HelloWorldStub extends YaRpcStub
         implements IHelloWorld {

    public HelloWorldStub( String objectKey )
    {
        super( objectKey );
    }

    @Override
    public String getGreeting()
    {
        return "Hello world";
    }
}
```

```
...

YaRpcManager.startListening( "ynp://127.0.0.1" );
HelloWorldStub stub = new HelloWorldStub( "key" );
stub.registerToYaRpc();
```

Figure 19 An implementation of a server in Java

Similarly to the .NET implementation, the *YaRpcProxy* class simplifies development of client applications by providing the *yaRpcRemoteCall* method that helps with issuing the remote call by creation of the appropriate *YaRpcRequest* and *YaRpcResponse* objects, based on the signature of remotable methods as discovered by runtime code reflection. The main issue is that Java class files do not preserve names of method parameters, and consequently, this information is not available in the reflection. In order to support RPC protocols that need to know the parameter names, the Java YaRpc implementation provides the *@YaRpcParam* annotation that can gives a name to a method parameter.

Same as the *YaRpcStub* class, the *YaRpcProxy* class is also used without a generic type argument; the *YaRpcProxy* implementation can obtain the runtime type of the inheritor using other means. For an example of a client with a manually created proxy object, see Figure 20.

```
public class HelloWorldProxy extends YaRpcProxy
        implements IHelloWorld {

    public HelloWorldProxy( IYaRpcConnection conn, String objectKey )
    {
        super( conn, objectKey );
    }

    @Override
    public String getGreeting()
    {
        return (String) yaRpcRemoteCall( "GetGreeting" );
    }
}

...

IYaRpcConnection conn = YaRpcManager.connect( "ynp://127.0.0.1" );
IHelloWorld proxy = new HelloWorldProxy( conn, "key" );
String greeting = proxy.getGreeting();
```

Figure 20 Client code, with manually created proxy in Java

Java also offers means to dynamically generate byte code and to dynamically load class files in the run-time. Unfortunately, standard Java libraries do not provide any convenient API to generate the byte code, so a third-party library might be necessary, such as Javassist [Javass] or Apache BCEL [Bcel]. By using dynamic byte code generation, we could generate a particular inheritor of *YaRpcProxy* automatically, and thus greatly simplify development of YaRpc client applications, as shown in Figure 21. Note that this functionality is currently not implemented, and will be added in the future.

```
IYaRpcConnection conn = YaRpcManager.connect( "ynp://127.0.0.1" );
IHelloWorld proxy = (IHelloWorld)
                    YaRpcProxy.generate( IHelloWorld.class,
                                         conn, "key" );
String greeting = proxy.getGreeting();
```

Figure 21 Client code, with a dynamically generated proxy in Java

The YaRpc Java implementation is available as a stand-alone JAR file called *YaRpc.jar*, which does not depend on any other third-party libraries, in order to provide maximum portability and ease of use.

4. Evaluation

The main goal of this thesis was to design a flexible high-performance RPC middleware that would enable software developers to create distributed applications with an unprecedented ease. In order to evaluate the level to which this goal has been accomplished, we provide an example distributed application implemented with Java RMI, .NET Remoting, SOAP web services and YaRpc middleware with the YNP protocol. Each implementation is accompanied with a description of all the steps an application developer needs to perform.

Unfortunately, as described in Section 5, both Java and .NET Framework implementations of the YNP protocol are still in the state of a prototype and do not reach a production quality, which prevented us to provide an accurate comparison of YNP protocol performance compared to other protocols.

4.1.1. Java RMI

In this example, we present all the steps necessary to develop an example distributed application using Java RMI. Even though a number of third-party libraries provide a different means to implement the example application in RMI, we chose to only use standard tools and libraries provided by the Java Development Kit (JDK) 1.6.

1. Define a communication interface to the remote object, which will be shared by both client and server. Every interface to a remote object in RMI needs to extend the *java.rmi.Remote* interface and the remotable methods need to declare that they throw the *RemoteException* exception – see Figure 22.
2. Implement the class representing the remote object. This class is only available on the server and implements the communication interface as defined in step 1 - see Figure 23.
3. Create the server main code, which registers the remote object into *rmiregistry* by calling the *rebind* method on the object *Naming* - see Figure 23.

4. Create the client source code, where first the security policy needs to be set up, and then the remote object needs to be looked up in *rmiregistry* by calling the *lookup* method on the object *Naming* - see Figure 24.
5. Run the *rmiregistry* utility (which is a part of JDK) on both client and server, in order to set up the RMI registry on port 1099.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface IRObject extends Remote {

    void sayHelloWorld( String clientName ) throws RemoteException;

    long getSum( int numA, int numB ) throws RemoteException;

    List<String> reverseArray( List<String> inputArray )
            throws RemoteException;
}
```

Figure 22 Java RMI communication interface

```
public class RObjectImpl extends UnicastRemoteObject implements IRObject {

    private static final long serialVersionUID = 1L;

    public RObjectImpl() throws RemoteException { super(); }

    public void sayHelloWorld( String clientName ) throws RemoteException
    {
        try {
            System.out.println(
                String.format( "Hellow world from %s", clientName) );
        }
        catch (Exception e) {
            throw new RemoteException("Error in sayHelloWorld", e);
        }
    }

    ...

public static void main(String[] args) {

    try
    {
        String codebase = "file:/" +
              new File("").getAbsolutePath() + "/";
        codebase = codebase.replaceAll(" ", "%20");

        System.setProperty("java.rmi.server.codebase", codebase);

        String host = args.length > 0 ? args[0] : "localhost";

        RObjectImpl remoteObject = new RObjectImpl();
```

```
        Naming.rebind("//" + host + "/RemoteTestObject", remoteObject);
        System.out.println("Remote object is registred in RMIregistry.");

        ...
```

Figure 23 Java RMI server code

```
public static void main(String[] args) {
    try
    {
        System.setProperty("java.security.policy", "rmi.policy");
        if (System.getSecurityManager() == null)
            System.setSecurityManager(new RMISecurityManager());

        String host = args.length > 0 ? args[0] : "localhost";

        IRObject remoteObject = (IRObject) Naming.lookup("//"
                                           + host + "/RemoteTestObject");

        remoteObject.sayHelloWorld( "Client" );

        ...
```

Figure 24 Java RMI client code

4.1.2. .NET Remoting

In this section we present an implementation of the equivalent distributed application as in Section 4.1.1, in C# language and using .NET Remoting for underlying communication. In general, .NET Framework provides a fairly easy and straightforward way to create a simple distributed application, however, things get more complicated if a developer wants to get more control over configuration of the remoting infrastructure. In order to create the example application, the following steps need to be performed:

1. Create an implementation of the remotable object in an assembly, which will be referenced by both client and server. The remote object has to extend the *MarshalByRefObject* class (which is "*the base class for objects that communicate across application domain boundaries by exchanging messages using a proxy*", see [Msdn]) - see Figure 25.
2. Create a server code which registers the remote object in the .NET Remoting repository - see Figure 26.

3. Modify a server configuration file that specifies the type of the transport channel (TCP or HTTP), the mode of a calling behavior of object and the URI where the remote object will be made available – see Figure 27.
4. Create the client code, which loads the configuration from a file, and uses that information to create a proxy to the remotable object – see Figure 28.
5. Define the client configuration file, to specify the URI of the remotable object – see Figure 29.

```
namespace RObjects
{
	public class RObject : MarshalByRefObject
	{
		public void SayHelloWorld(string clientName)
		{
			Console.Out.WriteLine("Hello world from " + clientName);
		}

		public long GetSum(int a, int b)
		{
			return ((long)a + (long)b);
		}

		public List<string> ReverseArray(List<string> array)
		{
			array.Reverse();
			return (array);
		}
	}
}
```

Figure 25 .NET Remoting remote object implementation

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;

namespace Server
{
	class Server
	{
		static void Main(string[] args)
		{
			RemotingConfiguration.Configure("server.config", false);
			Console.WriteLine("Server start listening");
			Console.ReadLine();
		}
	}
}
```

Figure 26 .NET Remoting server code

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
                   type="RObjects.RObject, RObject"
                   objectUri="RObject.rem" />
      </service>
      <channels>
        <channel ref="http" port="8989"/>
        <!--if you want tcp then port would be =8080-->
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Figure 27 .NET Remoting server configuration file

```csharp
using System.Runtime.Remoting;
using RObjects;

namespace Client
{
        class Client
        {
                static void Main(string[] args)
                {
                        try
                        {
                                RemotingConfiguration.Configure(
                                    "Client.exe.config", false);
                                RObject remoteObject = new RObject();

                                remoteObject.SayHelloWorld("Client");

                                ...
```

Figure 28 .NET Remoting client code

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="RObjects.RObject, RObject"
                   url="http://localhost:8989/RObject.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Figure 29 .NET Remoting client configuration file

4.1.3. SOAP Web Services with C#

In this section we present the example application developed using a web services created in C# using Microsoft Visual Studio IDE, which allows programmers to define web methods without manually creating a WSDL definition file. The creation of the web services application comprised the following steps:

1. In order to develop the server, create a new web services project in Microsoft Visual Studio IDE. The wizard will automatically generate the basic code.
2. Define methods of the web service – see Figure 30.
3. Deploy the web service to a web server, such as Internet Information Services (IIS) server.
4. In order to create the client, obtain the WSDL definition of the web service from the server, and use the *wsdl.exe* tool to generate a proxy code to access the web service – see Figure 31.
5. Implement the client code, using the generated proxy – see Figure 32.

```
[WebService(Namespace = "http://yarpc.example.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[ToolboxItem(false)]
public class WSexample: System.Web.Services.WebService
{
        [WebMethod]
        public string SayHelloWorld( string clientName )
        {
                return ( "Hello World from " + clientName );
        }

        [WebMethod]
        public long GetSum(int a, int b)
        {
                return ( (long)a + (long)b);
        }

        [WebMethod]
        public List<string> ReverseArray(List<string> array)
        {
                array.Reverse();
                return (array);
        }
}
```

Figure 30 Web service server code in C#

```
// This source code was auto-generated by wsdl utility
[System.Web.Services.Protocols.
SoapDocumentMethodAttribute("http://yarpc.example.org/SayHelloWorld",
RequestNamespace="http://yarpc.example.org/",
ResponseNamespace="http://yarpc.example.org/",
Use=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
public string SayHelloWorld(string clientName) {
    object[] results = this.Invoke("SayHelloWorld", new object[] {
                clientName});
    return ((string)(results[0]));
}
```

Figure 31 Auto-generated web service proxy in C#

```
WSexample webService = new WSexample();
string greetings = webService.SayHelloWorld("C# client");
long sum = webService.GetSum(5, 8);
```

Figure 32 Web service client code in C#

Note that the creation of distributed applications using web services in C# is greatly simplified thanks to the tools available in Microsoft Visual Studio; manual creation of both server and client web service is a much more complicated and error-prone task.

4.1.4. SOAP Web Services with Java

The process of development of web services server and client in Java is very similar to the development process in C#; a lot of the application code can be generated using wizards available in IDEs such as Netbeans [Nbide]. The development process consists of the following steps:

1. In order to develop the server, create a new web services project in Netbeans. The wizard will automatically generate the basic code.
2. Define methods of the web service – see Figure 33. Netbeans offers another wizard to help a developer to get the annotations right.
3. Deploy the web service to a web server, such as Apache Tomcat server.
4. In order to create the client, obtain the WSDL definition of the web service from the server, and use the Java API for XML Web Services (jax-ws) [Jaxws] to generate a proxy code to access the web service – see Figure 34.

5. Implement the client code, using the generated proxy methods – see Figure 35.

```
@WebService()
public class WSExample {

    @WebMethod(operationName = "sayHelloWorld")
    public String sayHelloWorld(@WebParam(name = "clientName")
    String clientName) {
        return "WSJava: hello world from: " + clientName;
    }

    @WebMethod(operationName = "getSum")
    public long getSum(@WebParam(name = "a")
    int a, @WebParam(name = "b")
    int b) {
        return ( (long) a + (long) b );
    }

    @WebMethod(operationName = "reverseArray")
    public List<String> reverseArray(@WebParam(name = "array")
    List<String> array) {
        Collections.reverse(array);
        return (array);
    }
}
```

Figure 33 Web service server code in Java

```
    private static String sayHelloWorld( String clientName )
    {
        javaclientws.WSexample service = new javaclientws.WSexample();
        javaclientws.WSexampleSoap port = service.getWSexampleSoap();
        return port.sayHelloWorld( clientName );
    }

    private static long getSum( int a, int b )
    {
        javaclientws.WSexample service = new javaclientws.WSexample();
        javaclientws.WSexampleSoap port = service.getWSexampleSoap();
        return port.getSum( a, b );
    }
```

Figure 34 Web service auto-generated proxy methods in Java

```
    public static void main(String[] args)
    {
        String sayHelloWorld = sayHelloWorld( "Java client" );
        long sum = getSum( 10, 15 );
        ...
```

Figure 35 Web service client code in Java

4.1.5. YaRpc with YNP in C#

In this section we show how to implement an equivalent distributed application with YaRpc middleware, YNP protocol, and C# programming language. The development process consists of the following steps:

1. Define the communication interface, shared by both client and server – see Figure 36.
2. Implement the methods of the communication interface on the server, and activate the server – see Figure 37.
3. Create the client code, using a runtime-generated proxy – see Figure 38.

```
public interface IRObject
{
  [YaRpcMethod] string SayHelloWorld(string clientName);
  [YaRpcMethod] long GetSum(int a, int b);
  [YaRpcMethod] List<string> ReverseArray(List<string> array);
}
```

Figure 36 YaRpc+YNP communication interface definition in C#

```
public class RObjectStub : YaRpcStub<IRObject>, IRObject
{
   public RObjectStub(string objectKey) : base(objectKey)
   { }


   public string SayHelloWorld(string clientName)
   {
     return "Hello world from " + clientName;
   }

   public long GetSum(int a, int b)
   {
     return (long) a + (long) b;
   }

   public string[] ReverseArray( string[] array )
   {
     Array.Reverse( array );
     return array;
   }
 }

 ...
 YaRpcManager.StartListening("ynp://127.0.0.1:1234");
 RObjectStub stub = new RObjectStub("robject");
 stub.RegisterToYaRpc();
 ...
```

Figure 37 YaRpc+YNP server code in C#

```
...
IYaRpcConnection con = YaRpcManager.Connect("ynp://127.0.0.1:1234");
IRObject proxy = YaRpcProxy<IRObject>.Generate( con, "robject" );

string greeting = proxy.SayHelloWorld("YaRpc C# client");
long sum = proxy.GetSum(5, 8);
...
```

Figure 38 YaRpc+YNP client code in C#

The Java equivalent of the example application looks very much the same as the C# version, so we skipped it. Clearly, the development of the application using YaRpc only requires three trivial steps that can easily be performed manually, without need of any external tool or wizard. In our opinion, YaRpc enables developers to create new distributed applications rapidly, even if they have limited or no knowledge of the underlying infrastructure or distributed computing in general, which is not the case with the other RPC technologies presented in this section.

5. Conclusion & Future Work

In this thesis we have presented a specification of an extensible RPC middleware, which can support virtually any network transport layer and RPC protocol. Additionally, we have designed a new light-weight communication protocol, the YaRpc Native Protocol (YNP), which is an interesting alternative to existing RPC protocols thanks to features such as high performance due to a binary message format, server-initiated callbacks, customizable serialization, asymmetric serialization and nested method calls. As a proof of concept, we provide an implementation of the YaRpc middleware specification in both Microsoft .NET Framework (with C#) and Java programming languages, including the support of the YNP protocol. The implementations try to bring high-performance distributed computing to the fingertips of application developers, by utilizing advanced data structures and capabilities of the underlying operating systems, while still offering a simple to use application programming interface (API).

The main contribution of the thesis is a proof that an RPC middleware can be made simple and generally usable, and still offer a rich set of features. We have shown how the capabilities of modern high-level programming languages can be leveraged to encapsulate a complex functionality into an easy-to-use API. Also, to the best of our knowledge, YaRpc is the only middleware that aims to fully support a wide range of existing RPC protocols under a single API. Some of the features provided by YNP, such as nested method calls or asymmetric serialization, are not readily available in any other RPC middleware.

We need to note that both implementations of the YNP protocol in C# and Java are currently in a state of a prototype, and cannot be used in a production environment as is, without additional testing and debugging. We believe, however, that in forthcoming weeks both implementations will reach a sufficient level of maturity that will allow us to release the source code to the public. We also believe that soon after the release YaRpc will attract an open source development community that will drive further extension of YaRpc with other RPC protocols and features. If we succeed in this

effort, YaRpc has a potential to become a popular tool used by distributed application developers world-wide.

In a longer term, besides adding support for new transport layers and RPC protocols, we plan to introduce a range of unique features such as connection pooling and transparent network failover. After a final release of Java 7 platform we want to provide a YaRpc implementation for Java that would take advantage of the new asynchronous I/O API provided by the JSR 203 [Jsr203], and thus enabled developers to create high-performance distributed applications in Java with a little effort. Also, a new version of the YNP protocol is being planned, that would allow serialization of remotable object references by generating appropriate proxy objects, and intelligent handling of proxy-to-proxy chains.

References & Annexes

[Bcel] The Byte Code Engineering Library Project, http://jakarta.apache.org/bcel/

[Bhkk06] J.W. Berry, B.A. Hendrickson, S. Kahan, P. Konecny: Graph Software Development and Performance on the MTA-2 and Eldorado, 2006

[Bir83] A.D. Birrell, B.J Nelson: Implementing Remote Procedure Calls, XEROX CSL-83-7, 1983

[Dce80] D. Cohen: On Holy Wars and a Plea for Peace, IEN 137, 1980

[Dcom98] The Open Group (1998): The COM/DCOM Reference, Documentation for ActiveX Core Technology

[Etch08] The Etch Project, https://cwiki.apache.org/ETCH/

[Ft07] The Thrift Project, http://thrift.apache.org/

[Gpb08] The Google Protocol Buffers Project, http://code.google.com/p/protobuf/

[Java06] Sun Microsystems, Inc.: JavaTM Platform, Standard Edition 6, API Specification, 2006

[Javass] The Java Programming Assistant Project, http://www.javassist.org/

[Jaxws] The JAX-WS Project, http://jax-ws.java.net/

[Jc07] J. Čurn: Distribution for Open Modeling Interface and Environment, Master thesis, MFF UK, 2007

[Jon02] A. Jones, J. Ohlund: Network Programming for Microsoft Windows, Second Edition, 2002

[Jsr203] Java Specification Requests: More New I/O APIs for the JavaTM Platform ("NIO.2"), http://jcp.org/en/jsr/, 2007

[Mono] The MONO Project, http://www.mono-project.com/

[Msdn] Microsoft Corporation: The Microsoft Developer Network library, http://msdn.microsoft.com/

[Msil00] Microsoft Corporation: MSIL Instruction Set Specification, 2000

[Mwgs07] M. Wilson, Gathering Scattered I/O, 2007

[Net07] Microsoft Corporation: .NET Framework Reference, 2007

[Omg04] Object Management Group, Inc.: Common Object Request Broker Architecture: Core Specification, 2004

[Rest07] L. Richardson, S. Ruby: RESTful Web Services, Web services for the real world, O'Reilly Media, 2007

[Rfc1057] Sun Microsystems, Inc.: Remote Procedure Call, Version 2, 1988

[Rfc1831] R. Srinivasan: RPC: Remote Procedure Call Protocol Specification Version 2, Sun Microsystems, Inc., 1995

[Rfc707] J. E. White: A High-Level Framework for Network-Based Resource Sharing, Stanford Research Institute, 1976

[Rfc793] J.Postel: Transmission Control Protocol, USC/Information Sciences Institute, 1981

[Ssl94] Hickman, Kipp: The SSL Protocol, Netscape Communications Corp., 1994

[Web04] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard: Web Services Architecture, W3C Working Group Note, 2004

[Win07] Microsoft Corporation: Windows API Reference, 2007

[Ws97] Microsoft Corporation: Windows Sockets 2 API specification, 1997

[Xml99] Dave Winer: XML-RPC Specification, 1999

List of Tables

Table 1 RPC technologies comparison 19
Table 2 *YaRpcManager* class methods 25
Table 3 *IYaRpcConnection* interface methods 27
Table 4 *YaRpcRequest* class properties 28
Table 5 *YaRpcResponse* class properties 29
Table 6 *IYaRpcTransport* interface methods 30
Table 7 *IYaRpcRemotable* interface methods 32
Table 8 *IYaRpcWorker* interface methods 33
Table 9 *YaRpcSmartStream* class methods 36
Table 10 *YaRpcWriter* class methods 38
Table 11 *YaRpcReader* class methods 39
Table 12 *YaRpcStub* class methods 41
Table 13 *YaRpcProxy* class methods 41
Table 14 Abstract methods of the *YaRpcConnectionBase* class 42
Table 15 YNP protocol message header 43
Table 16 YNP protocol REQUEST message body 44
Table 17 YNP protocol RESPONSE message body 44
Table 18 YNP protocol EXCEPTION message body 44
Table 19 Serialized value signature byte 45
Table 20 Signature type code 46
Table 21 The *YaRpcNativeSerializer* class methods 47

List of Figures

Figure 1 Local procedure call vs. remote procedure call 5
Figure 2 CORBA architecture 8
Figure 3 Microsoft .NET Remoting architecture 10
Figure 4 Java RMI architecture 11
Figure 5 Web services architecture 13
Figure 6 XML-RPC architecture 15
Figure 7 Etch architecture 16
Figure 8 Thrift architecture 18
Figure 9 YaRpc architecture 23
Figure 10 Causality of remote method calls in YNP (Source: [Jc07]) 49
Figure 11 A communication interface, shared by both client and server 51
Figure 12 An implementation of a server in .NET 51
Figure 13 Client code, with a manually created proxy in .NET 52
Figure 14 Client code, with a dynamically generated proxy in .NET 53
Figure 15 A method delegate in Java 54
Figure 16 A token to an asynchronous operation in Java 54
Figure 17 Implementation of an asynchronous operation in Java 55
Figure 18 Definition of a communication interface in Java 56
Figure 19 An implementation of a server in Java 57
Figure 20 Client code, with manually created proxy in Java 58
Figure 21 Client code, with a dynamically generated proxy in Java 58
Figure 22 Java RMI communication interface 60
Figure 23 Java RMI server code 61
Figure 24 Java RMI client code 61
Figure 25 .NET Remoting remote object implementation 62
Figure 26 .NET Remoting server code 62
Figure 27 .NET Remoting server configuration file 63
Figure 28 .NET Remoting client code 63
Figure 29 .NET Remoting client configuration file 63

Figure 30 Web service server code in C# 64
Figure 31 Auto-generated web service proxy in C# 65
Figure 32 Web service client code in C# 65
Figure 33 Web service server code in Java 66
Figure 34 Web service auto-generated proxy methods in Java 66
Figure 35 Web service client code in Java 66
Figure 36 YaRpc+YNP communication interface definition in C# 67
Figure 37 YaRpc+YNP server code in C# 67
Figure 38 YaRpc+YNP client code in C# 68

Definitions and Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DoS	Denial of Service attack
GIOP	General Inter-ORB Protocol
GUI	Graphical user interface
GUID	Globally Unique Identifier
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I/O	Input/Output
IDE	Integrated development environment
IDL	Interface description language
IEEE	Institute of Electrical and Electronics Engineers
IIOP	Internet Inter-ORB Protocol
J2EE	Java 2 Enterprise Edition
JDK	Java Development Kit
JRMP	Java Remote Method Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MSIL	Microsoft Intermediate Language
MVC	Model View Controller
NFS	Network File System
OMA	Object Management Architecture
OMG	Object Management Group
ONC	Open Network Computing
ORB	Object Request Broker
RMI	Remote Method Invocation

RPC	Remote Procedure Call
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
STA	Single Thread Apartment model
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol / Internet Protocol
TIDL	Thrift Interface Description Language
UCS	Universal Character Set
UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF-8	UCS Transformation Format
WSIL	Web Services Inspection Language
XML	Extensible Markup Language
YaRpc	Yet Another Remote Procedure Call
YNP	YaRpc Native Protocol