JOURNAL OF OBJECT TECHNOLOGY

.NET Remoting and Web Services: A Lightweight Bridge between the .NET Compact and Full Framework

Bert Vanhooff, K.U. Leuven, Department of Computer Science, Belgium
Davy Preuveneers, K.U. Leuven, Department of Computer Science, Belgium
Yolande Berbers, K.U. Leuven, Department of Computer Science, Belgium

Abstract

With the growing popularity of powerful connected mobile devices (PDAs, smart phones, etc.), an opportunity to extend existing distributed applications with mobile clients emerges. The Microsoft .NET Compact Framework offers a development platform for mobile applications but is lacking support for .NET Remoting, which is the .NET middleware infrastructure for inter-application communication. The current version of the .NET Compact Framework (1.0, SP2) does support communication using web services. Unfortunately this support cannot be used in its current form to seamlessly integrate with an existing .NET Remoting application. In this paper, we propose an approach that leverages the present support for web services and augments it to make such integration possible. Our solution dynamically maps back and forth between .NET Remoting and web service messages without needing to alter the existing Remoting applications.

1   INTRODUCTION

.NET is a Microsoft brand name that encompasses a whole array of technologies. A few key terms associated with this brand name are *connected systems*, *smart devices* and *web centric computing*. These terms could be categorized under the more general denominator of distributed systems. In short, .NET offers a complete package of tools and technologies for developing applications, especially targeted towards distributed systems.

The most important part of .NET is the .NET Framework [Mic], which consists of an execution environment for applications and a comprehensive class library. The framework includes .NET Remoting [Mcl03] in order to support the development of distributed applications. This is an extensible Distributed Object Computing (DOC) middleware infrastructure comparable to the Java Remote Method Invocation (RMI) [Sun] although the latter adopts an entirely different internal architecture.

The .NET Compact Framework [Wig03] is a slimmed down version of the .NET Full Framework made to run on embedded devices such as *PDAs* and *smart phones*. To take into account the resource limitations of these devices, a dedicated execution environment was crafted and some classes and methods of the standard .NET class library were removed. .NET Remoting was among those removed namespaces. As Remoting cannot be used on the .NET Compact Framework, a communication barrier exists between the .NET Full Framework and the .NET Compact Framework (Figure 1).
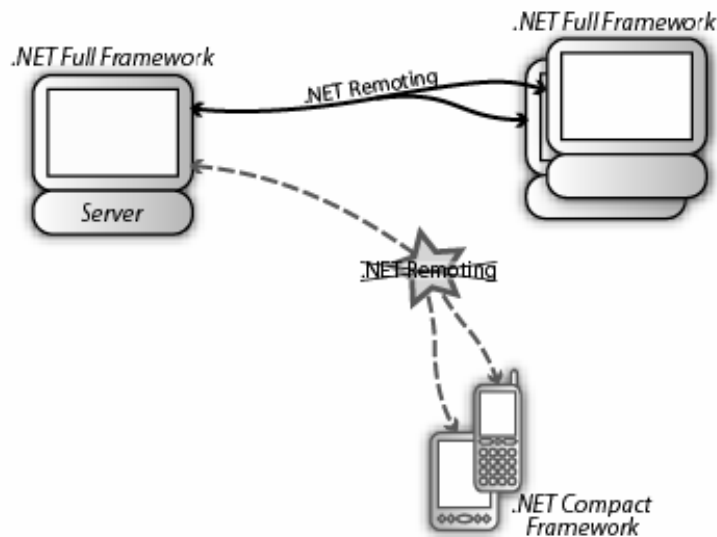

Figure 1 The .NET Remoting barrier between the .NET Full and Compact Framework

The absence of .NET Remoting in the .NET Compact Framework puts some serious constraints on the development of connected smart clients. Recently, advanced and Wifi-enabled PDAs and smart phones have created the need to extend existing distributed applications to incorporate these smart clients. However, the lack of support for .NET Remoting on these devices makes it hard to integrate them into legacy systems, often build with that technology (Figure 1). In this paper we propose a mechanism that enables smart clients running the .NET Compact Framework to access these Remoting objects to a certain extent, with little overhead for the programmer and little changes to be made to the existing part of the distributed application. Our approach focuses on client/server architectures: the server is not aware of any objects on the client, while the client can see and access objects on the server. Our solution uses .NET Remoting's built-in extension support (on the server) and a custom extension to the existing web services support on the client.

In the next section, .NET Remoting and web services are discussed in detail and a list of requirements for our solution is presented. In section 3 we explain the concepts that solve the basic requirements. These concepts are then used as the foundation to solve other requirements (section 4). We discuss our implementation of the basic concepts and some practical results in section 5. Finally we present some related work (section 6) and

we round up the paper by drawing conclusions, summarizing the strengths and weaknesses of our solution and giving suggestions for future improvement (section 7).

2   DISTRIBUTED APPLICATIONS IN .NET

The .NET Framework offers .NET Remoting and web services as high-level technologies for developing distributed systems. This section introduces the parts of these technologies that are relevant for the rest of the paper and it points out the constraints involved when using web services instead of .NET Remoting. We conclude this section by giving a minimum set of requirements for a useful solution.
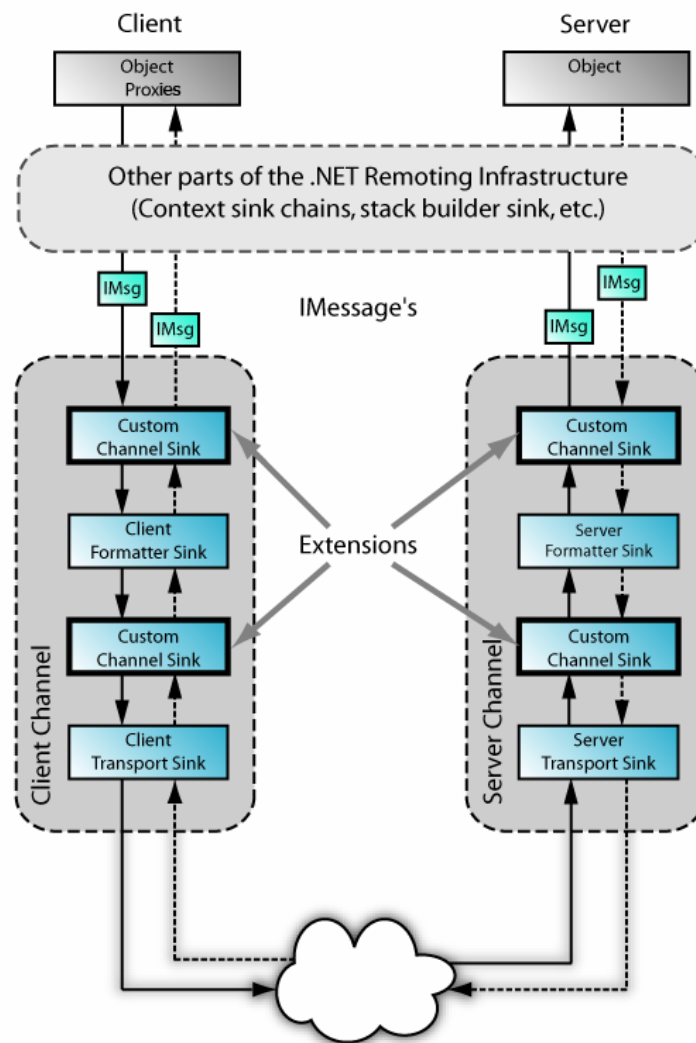

Figure 2 Channel sinks and IMessages in the Remoting infrastructure.

.NET Remoting

.NET Remoting simplifies the development of distributed systems by offering an extensible infrastructure that permits objects that do not reside in the same memory space (or even on the same host) to communicate with one another in a transparent fashion. This implies that every message sent to a remote object will have to be delivered through an alternative (non-local) mechanism. Therefore, each message from a local (client) object to a remote (server) object will be intercepted using a (double) proxy pattern. A message, which can for example represent a method or constructor call, will be transformed into a Remoting message-object. Such a message object implements the IMessage interface (we will refer to these types of objects as IMessage objects) and contains all the necessary information needed to reconstruct the original call. IMessage objects are created and inserted into the Remoting infrastructure by the proxies. A graphical representation of the relevant parts of the Remoting infrastructure is shown in Figure 2. We elaborate on these parts in the next few paragraphs.

After passing through the proxies, an *IMessage* object is further propagated through the .NET Remoting infrastructure. This part contains several so called *sink chains*, which are series of concatenated sink objects, each given the opportunity to modify the *IMessage* object as in a *pipe-and-filter* architecture. The sink chains provide the main extension mechanism by enabling the insertion of your own custom sink objects. Some sink objects are provided by default, including a *formatter sink* that serializes *IMessage* data and a *transport sink* that takes care of physical message transportation. Sink chains that contain these two types of sinks are located at the end of the Remoting infrastructure and are called *channels*. Channels are the first components in the .NET Remoting infrastructure that get to see *all* incoming messages and the last to see *all* outgoing messages. Sink chains that do not belong to a channel only get to see specific categories of *IMessages*. For example a sink in the *server object sink chain*, only gets to see *IMessages* originating from one specific object.

Figure 2 shows the possible flow of an *IMessage* through the sinks in a channel when both client and server are using .NET Remoting. An *IMessage* is created in the proxies (top-left of the figure) on the client and travels through the infrastructure (full lines) until it arrives at the first *Custom Channel Sink*, which is a specialized version of a message sink. Each custom sink shown in the figure actually represents either one custom message sink or a subchain of custom message sinks (only one is shown due to space considerations). The *IMessage* then moves further to the *Client Formatter Sink*, where it is serialized. After that, the message passes another series of *Custom Channel Sinks* to finally end in the *Client Transport Sink*. This last sink physically sends the message to the server using some kind of network technology. When the message is received at the server (right part in the figure), an equivalent chain of sinks is passed on the server until the call to the actual object can be executed. A response will, in turn, be represented by an *IMessage* that travels in the opposite direction (dotted lines).

To summarize, *Proxies*, *IMessages* and *Custom Sinks* are three important elements that provide for the extensibility of .NET Remoting. Using one or more of these
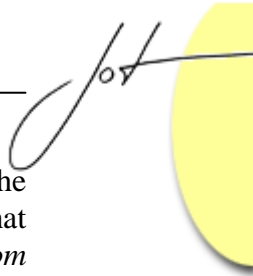
extension elements we can add, for example, encryption or logging facilities to the standard .NET Remoting functionality. A more exotic extension could be one that provides a new serialization mechanism. Later on in this paper we will use a *Custom Channel Sink* that manipulates *IMessages*.

Web Services

In general, "web services" refer to all techniques that enable applications to offer their services to one another over a network by means of Internet technologies. In this paper, we use the more restricted interpretation of [W3c02, Boo03]: SOAP over HTTP and WSDL. SOAP (Simple Object Access Protocol) [Box00] is an XML based protocol for the messages sent by a web service, while WSDL (Web Service Description Language) [W3c03] is an XML language used to describe the interface offered by such a service. The use of these protocols is fully supported in the .NET Compact Framework.

An important characteristic of web services is that they have a document-oriented (or, in RPC mode, a simple procedure-oriented) architecture instead of an object oriented architecture. They do not support an object reference-model; all data in a document is passed by value. This prevents us from directly using web services to interoperate with .NET Remoting because exchanging documents is totally different from typical object oriented operations such as instantiating new objects, invoking methods on objects, passing object references, navigating object graphs, etc. Such operations are typically supported by DOC (Distributed Object Computing) middleware such as .NET Remoting, CORBA and Java RMI.
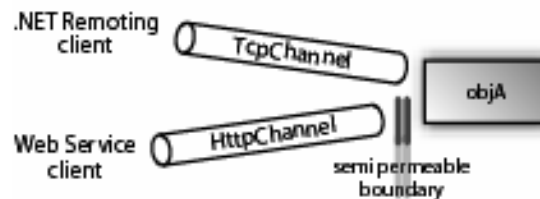

Figure 3 A Remote object published as a web service through the Remoting infrastructure.

One of the features of Remoting (that can be quite confusing), is its direct support for offering remote objects as if they were web services, through its own infrastructure. Individual remote objects can easily be published and accessed – in a very limited way – using web services (see Figure 3). When accessing a remote object through a web service in Remoting, the client can only call methods that return primitive or structured data types due to the lack of a reference-model. As a consequence, he cannot get outside the scope of the initial object by navigating the object graph because any method call, which would normally return a reference to an associated object, will now only return the data contained in the associated object (non-recursive) and not the object reference itself. One of the advantages of offering web services through .NET Remoting is that we can use its comprehensive extension mechanisms for handling web service requests. However, some functionality will be lost due to the inherent limitations of standard web services [Alm01].

In its current implementation, accessing remote objects through web services requires that the objects are (1) published on well-known URLs in advance and (2), that they are not removed during the application's lifetime in order to prevent calls to dead objects. Objects that are created during the operation of the system will not be accessible. Consequently, an application offered as a set of web services must have a static object graph (at least for the published objects) and may not delete any of the published objects since this would result in unanticipated access faults. In addition, newly created objects cannot be directly accessed by web service clients. Note, however, that data present in newly created objects could be accessed indirectly through methods from another object that is published as a web service.

A web service is generally accessed using a proxy in order to provide for some transparency and to keep the programmer from having to do a lot of cumbersome coding. There are standard tools available to generate these proxies for a remote object directly or using a WSDL description. Direct generation is only applicable if an object is published as a web service by the Remoting infrastructure. Whenever the tools encounter a method that returns or accepts an object, this object will be mapped to a complex SOAP data structure, removing some of the object's richness in the process. Consequently, from the point of view of these proxies the very notion of an object disappears. Additionally the notion of event notification is not present in the basic web service architecture, so this functionality also disappears in the WSDL.
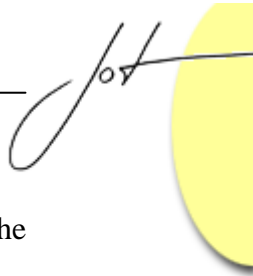
The web service limitations, along with the dynamic nature of most object graphs, make the web service support provided by .NET Remoting inadequate for developing smart clients with the same capabilities as full Remoting clients. This becomes an even greater issue when extending an existing Remoting application if the application was not originally designed with this puprose in mind. The focus of this paper is on extending such applications.

Requirements for Crossing the .NET Compact / Full Framework Boundary

We described that the lack of .NET Remoting on the .NET Compact Framework creates a communication boundary between the .NET Compact and Full Framework. Using the standard ability of both frameworks to consume (and offer, in case of the Full Framework) web services does not offer an adequate solution to cross that boundary. Therefore we have to figure out an alternative approach for interacting with remote objects that is able to offer many of the .NET Remoting advantages, without having to port the full Remoting infrastructure to the .NET Compact framework. A concrete list of the requirements we expect a good solution to meet is given here:

Functionality

1. Make the object graph on the server navigable from the client and enable the client to refer to a specific object on the server;
2. enable method calls on remote objects (with object references both as parameters and as return type);
3. enable callbacks (event occurrences) from the server to the client;

4. enable fast client development by hiding communication details from the programmer, generally minimize the programmer's overhead;
5. minimize the impact on existing applications.

We will show in the next sections that these requirements can be fulfilled by reusing large parts of the readily available infrastructure (and its extension mechanisms) and extending them on both the client and the server platform. Leveraging the existing infrastructure significantly shortened the development time of our solution.

3 USING TWO WEB SERVICE ABSTRACTION LAYERS TO ACCESS REMOTE OBJECTS

In this section we explain our approach to make remote objects available to clients running the .NET Compact Framework, or more generally to clients supporting web services. We start by describing the basic approach and continue by discussing some extensions that are build on this foundation.

Basic Approach

As already discussed, .NET Remoting can publish a degenerate version of the public interface of a remote object (no object references, no events, etc.) through a web service on a known URL. We will extend this basic capability with an extra layer of abstraction in order to overcome the inherent limitations. Figure 4 shows how we accomplish our goal at a highly abstract conceptual level. We will introduce two web service abstraction layers that hide communication details from the programmer and enable object-oriented-like communications over a web service link. The abstraction layer at the server side will make use of a Remoting extension and the Remoting support for offering web services, while the one at the smart device client will make use of custom generated proxies and the existing web service support of the .NET Compact Framework.
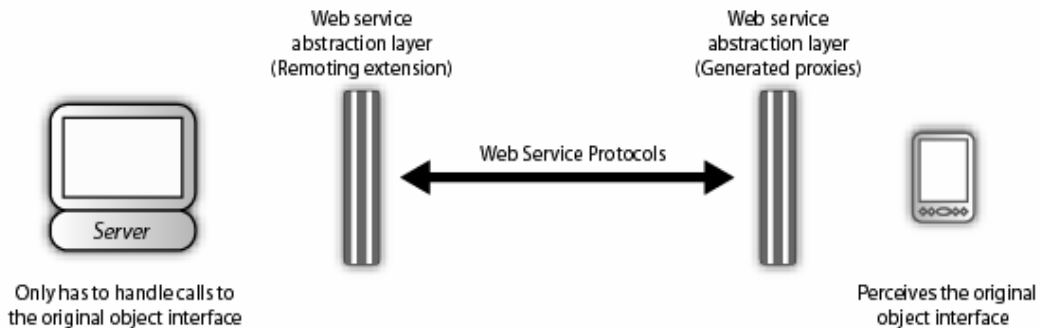

Figure 4 Web service abstraction layers.

The main reason we still like to use web services to build our solution, in spite of the mismatch between .NET Remoting and web services is twofold. First of all the support

for web services in the .NET Compact Framework provides us with high-level and easier to use communication primitives compared to plain sockets for example. Web services provide an RPC-style of communication and a common network representation of data. Secondly, since Remoting supports processing web service requests through its infrastructure, the modifications on the server can be better contained and are less intrusive on existing applications. These choices also ease the development effort of the abstraction layers themselves because we can use large parts of the existing infrastructure. A possible disadvantage is that we trap ourselves in the limitations of web services, hampering development with the search for workarounds. We will show that this was not the case.
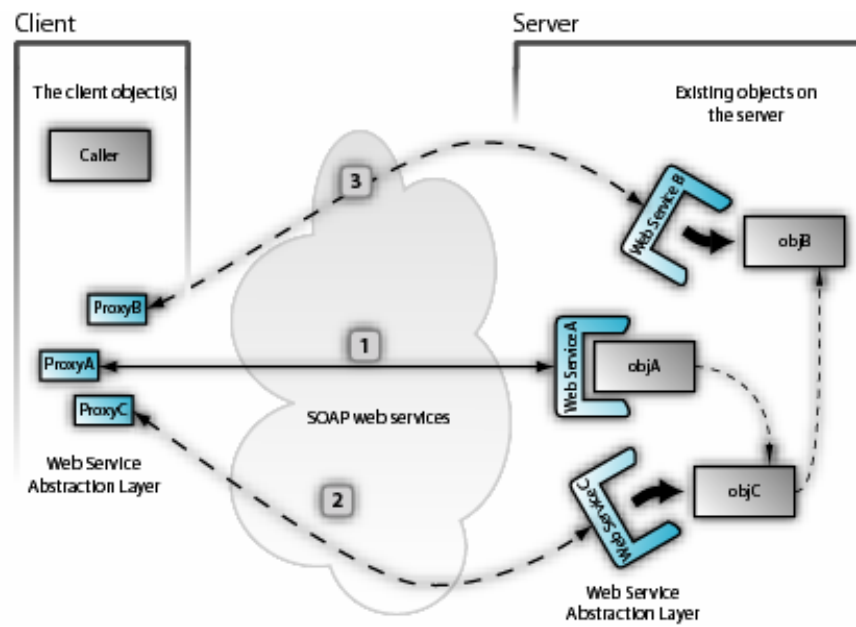

Figure 5 Basic operation of the web service abstraction layers.

The idea behind our solution is to dynamically publish a remote object as a web service whenever a web service client invokes an operation which returns a reference to that object. An object reference can then be uniquely mapped to a web service URL, which also functions as a globally unique identifier and locator for our server objects. Whenever an object reference is required we substitute it by the URL of the object's corresponding web service. Figure 5 demonstrates the idea by an example scenario. We start from an existing application consisting of a graph of three interconnected objects, objA, objB and objC. The starting object objA will be accessible – through a proxy – using a web service on a well-known URL (1). By invoking methods on this object, we can navigate to the other objects in the graph as follows. Whenever the client calls a method that returns a reference to another object (not transportable using standard web services), that object will be exposed through a web service. The URL to reach this service will instead be returned to the client as a substitute for the real object reference. Using this URL, the client can access the new object (2) using a corresponding proxy. In this way every object
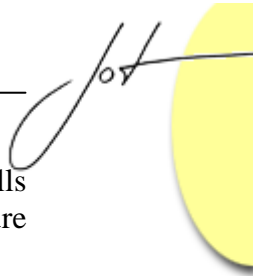
in the graph can be reached (3), effectively enabling navigability. Simple method calls not involving object references, can be handled by the standard web service infrastructure without intervention.

This approach requires two distinct web service abstraction layers, both on the client (in the form of custom proxies) and on the server (in the form of a .NET Remoting extension). In the next section, we present an elaboration of the general idea by using a more detailed method call scenario.

Remote Method Calls

To make invocations transparent, we introduce two new (non-.NET Remoting) proxies that will reside on the client and collaborate in order to represent a remote (server-)object on the client. The transparent proxy[1] mimics the interface of the remote object whereas the real proxy hides communication details. Both of them have only their name in common with the .NET Remoting proxies! In this subsection we refer only to the real proxy. On the server side, a custom message sink is inserted on top of the server channel sink to handle a client's request.

<div align="center">

`http://145.34.67.10:1200/[type:MyClassLib.MyClass][853b9985]`

server location   *object type*   *unique object id*

</div>

Figure 6 An object reference URL.

Real method signature	Mapped method signature
int Sqrt(int a)	int Sqrt(int a)
Car GetCar(int id)	**String** GetCar(int id)
Car Clone(**Car** c)	**String** Clone(**String** c)

Table 1 Mapping an object's interface.

The real proxy can be partially generated by extracting the interface of the class it will represent. However, some modifications to this interface are necessary when generating the proxy. These have to do with the limitations of web services concerning the notion of object references. As mentioned before, web services cannot transport objects (or better: references to objects). Only types that have an equivalent SOAP type [Box00] representation can be transported, which comes down to (C#) simple and structured value types and strings. This means that each time a non-transportable type is encountered in a method signature (the return type or a parameter type), it will be mapped to the transportable string type. An example of the different possibilities is given in

---

[1] The names for these proxies were inspired by the names of the proxies in .NET Remoting. To avoid any confusion, we do not refer to the .NET Remoting variants further in this text unless explicitly stated.

Table 1. We can see here that every time a class type is encountered, it is replaced by a string type. At runtime, this string will act as a container for the object reference, represented by a web service URL. An example of such an URL is given in

Figure 6. We need to include the type of the object reference in the URL for an optimization concerning the sending of collections/arrays. Without going into too much detail, we send all the references in a collection in one message to limit the number of requests. Since many collections can contain objects of arbitrary types, we need to dynamically determine the right proxy for each reference and we can do this by including the type in the URL of every collection element.

We use three different methods to marshal different (object) types. Objects that are normally marshaled by reference by the Remoting infrastructure are also marshaled by reference by our extension using the URL representation as presented in

Figure 6. Primitive types are marshaled by value and can be transported directly using SOAP messages without extra intervention. Complex value types (*structs* in C# [Alb01]) are also transported directly, except if they contain methods. In the latter case we use a third technique that combines marshal-by-value and marshal-by-reference. This hybrid technique first makes a server-local copy of the object and then creates a URL reference to that copy. The copy is required to enable several consecutive method-calls on the same *struct*-instance. Not doing so would result in a new volatile copy to be made for each call, hence losing the information from possible previous calls (meant for the same instance) in the process.

An alternative approach to achieve correct transport of complex types with methods is to always transport only the data in the instance using marshal-by-value. This data can then be loaded into a corresponding instance on the client that would act as a virtual proxy (does not communicate with the server but does represent a server type) that contains an implementation of the required methods. This solution would be more complicated to implement, while the first proposed method can reuse the existing marshal-by-reference facility. Another disadvantage of the alternative solution is the necessity to have the method implementations available on the client, causing possible hazards with versioning and incompatibilities with the .NET Compact Framework.

If a method does not contain non-transportable types, it can be offered in the interface unmapped and can be invoked without special intervention. On the other hand, if a method contains mapped parameters or return types, then the default mechanisms cannot be used and the invocation needs special care both on the client (handled in its proxies) and on the server (using a sink object) to take care of the marshaling.

A case where the return type is mapped will be discussed here. Suppose we want to invoke the method *MyClass GetMyClass()* on a remote object that we can reach via a web service on a known URL. Through the mapping mechanism this method will be exposed as *String GetMyClass()* and will be available as such in the *real proxy* on the client. The sequence of steps that will take place when calling that method is shown in

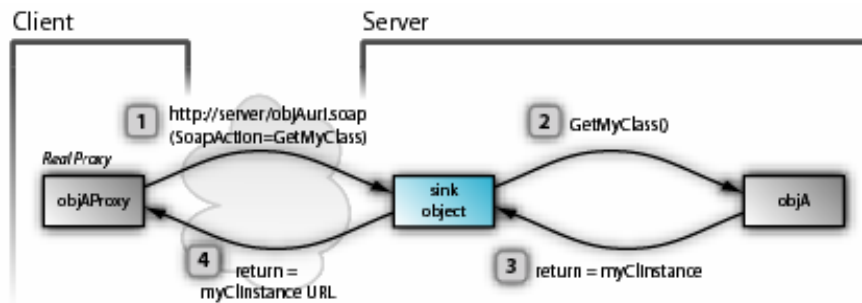Figure 7 and is explained in the following paragraph.

Figure 7 Remotely invoking a method.

When calling the method, all the details of that call are serialized into a SOAP message and this message is sent to the known URL (**1**). The method is actually called on a web service proxy that uses the standard class library of the .NET Compact Framework to hide the communication details from the caller. The proxy classes for the real proxy can be generated with the standard .NET tools (from our custom WSDL file, see further) The SOAP message then arrives at the server and is accepted into the .NET Remoting infrastructure. Here it is automatically deserialized into an *IMessage* object containing the same information as the SOAP message. After that, it is inserted into the sink chains, which means that our own custom sink object will get a chance to process the *IMessage*. In this case, the sink just passes the *IMessage* further up the chain (it does not contain any arguments) so the call is eventually invoked (**2**). If the method had contained mapped parameters, its arguments would have contained URLs that refer to other objects. These URLs should than first have been replaced by the actual object references before the *IMessage* is further propagated. The return value(s) of the method call will also be intercepted by our message sink (**3**). In response it will expose the returned object as a web service and replace the object reference in the return value with the URL of the created web service. Also, an extra reference to this object must be stored on the server to prevent it from being garbage collected. Whenever the returned object is a (non-primitive) value type (struct in C#), a local copy is stored to preserve the right semantics (see earlier in this section). The modified *IMessage* is now handed over to the next sink object to eventually be serialized to a SOAP message and sent back to the client (**4**). When the SOAP message is received, it is deserialized. The returned URL is then given to the proxy, which will give it back to its caller — which will in fact be a transparent proxy (see next section). The caller can in turn start invoking methods on the returned 'object' represented by the new web service. This will happen by instantiating a new real proxy for the corresponding type, and initializing it with the given URL.

The mechanism described above implies that proxies are available a priori for each used type. We do not think this introduces a serious limitation in this case because the programmer always knows which types he will use at compile time (not taking into account reflection-like mechanisms). Proxy generation at design time will in fact boost performance by taking away the processing cost of generating proxies at run time. While the mechanisms described up till now do enable basic communications, directly using real proxies does not provide for much transparency. The caller does not see the real

method signatures and has to manipulate URLs instead of real object references. In the next subsection, the transparent proxy is introduced in order to solve this problem.

Providing a Transparent Client Interface

To make the approach described above more transparent to the caller on the client, an extra level of indirection is introduced by adding a *transparent proxy* that interacts with the already discussed *real proxy*. The interface of the transparent proxy will mimic the interface of the object on the server that it represents, effectively providing transparency.

Table 2 shows the interface of the transparent proxy and its relation to the real proxy and the real object. The similarity between the interfaces of the transparent proxy and the real object are obvious. The only difference is that the underlying *Car* object is physically different, indicated in the interface as the *T* namespace (including the namespace will not be necessary in practice). In the transparent proxy, *T.Car* refers to the other transparent proxy type for the real *Car* object. The mapping between the different interfaces is done behind the scenes by the web service abstraction layers. The application programmer only gets to see the transparent proxy, while the server only needs to take into account the real object's interface.

Transparent Proxy	Real Proxy	Real Object
`int Sqrt(int a)`	`int Sqrt(int a)`	`int Sqrt(int a)`
`T.Car GetCar(int id)`	`String GetCar(int id)`	`Car GetCar(int id)`
`T.Car Clone(T.Car c)`	`String Clone(String c)`	`Car Clone(Car c)`

Table 2 The transparent proxy interface and its correspondence to the real proxy and real object.

Whenever a method, invoked on a transparent proxy, contains instances of other transparent proxies in its arguments, the transparent proxy will translate these arguments into their corresponding URLs before forwarding the call to the real proxy. The reverse translation is done with returned values. The real proxy in turn hides the rest of the communication details as discussed in the previous section. We illustrate the mechanism of transparent proxies in the scenario depicted in Figure 8.
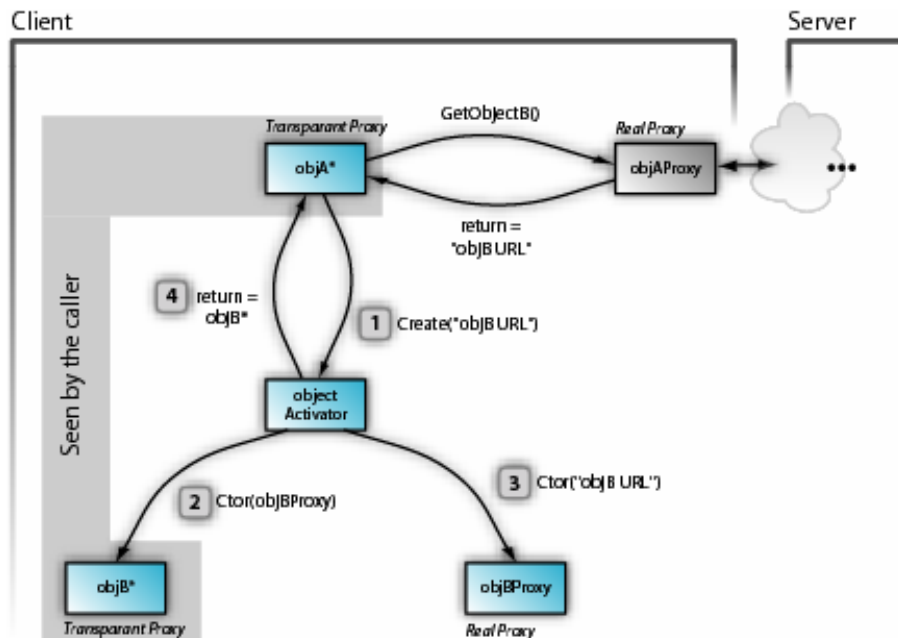
Figure 8 Transparent proxies hiding the URL object references.

This scenario starts when the transparent proxy *objA** (the * indicates that it mimics the interface of the remote object A) receives a response from the real proxy after calling its *GetObjectB()* method. This is where the scenario presented in

Figure 7 ended by returning a URL to the caller of *objAProxy*, which is represented by *objA** in the current scenario. The returned value is the URL to the web service of object B. The rest of the scenario goes as follows. Upon receiving the URL, the transparent proxy creates the necessary proxy objects that will enable the client to transparently work with the new object's web service. It delegates this task to the dedicated *objectActivator* – a proxy factory – by sending it the *create()* message (**1**). This *objectActivator* will check its cache to see if it already contains a transparent proxy that refers to the given URL. If none is found, it will create a new one and add it to the cache (**2**). By doing this we assure that only one reference to the remote object exists from within each client. A corresponding *real proxy* is also created (**3**). Eventually a reference to the newly created transparent proxy *objB** is returned to *objA** (**4**), which can in turn return it to its own caller (probably a local client object). The programmer only gets to see the transparent proxies *objA** and *objB**, the rest happens automatically behind the scenes.

Up till now we discussed our basic scheme. We did not give an answer to requirement 3 (callbacks from the server or events) yet. We also need to think about a mechanism for distributed garbage collection, preventing clients from influencing the server objects in malicious ways. These two extensions of the basic scheme are discussed in the next section.

4   EXTENSIONS FOR LIFETIME MANAGEMENT AND EVENTS

The previous section explained how references to remote objects can be obtained and how method calls can be carried out in a transparent fashion. However, there should also be a mechanism to manage the lifetime of remote objects that are accessed in this way. The server needs to know which objects are still referenced in order to carry out meaningful garbage collection. Requirement 3 also states that events on the server should be capable of being propagated to the clients. The mechanisms for addressing these two issues are presented in this section.

Distributed Lifetime Management

Distributed garbage collection is all about keeping track of remote references to an object and letting them play a role in the life cycle of the object. The goal is to prevent remote objects either from living forever or from being deleted too soon. Without further precautions being taken, the first case would apply to the approach explained so far because we only increase and do not decrease the reference count on the server. Whenever a client gets a reference to an object on the server, the object's local life cycle (the one of its proxy) will not be known to the server, which will result in an object that lives eternally. Note that we do not address the inverse problem of managing the life cycle of objects on the client that are referenced by the server because until now this has not been capable of happening. This client/server approach rules out the problem of dealing with circular references, which can only occur if an object acts as both client and server. We are only providing a view of the server's object graph on the client, nothing more, nothing less.

We propose the simple method of just letting the garbage collector on the client do its work on the proxies and, whenever a transparent proxy is destructed, notifying the server of this event. This technique comes down to a synchronization of life cycles rather than a fully fledged distributed garbage collection mechanism. Although easy to implement and working well in our specific case, it requires the *objectActivator* (proxy factory) to implement all references to transparent proxies as *weak references*[2] because otherwise the proxies would never qualify for garbage collection. A survey of more elaborate techniques for distributed garbage collection is given in [Pla95]. [Vei03] presents a distributed garbage collector that improves the current mechanisms used in .NET. The garbage collector is implemented in Rotor [Mic2] using the sink based extension mechanism.

---

[2] A weak reference is an object reference that does not influence garbage collection. Objects that only have weak references may be destroyed by the garbage collector. A user has to acquire a strong reference to the target object before being able to use it.
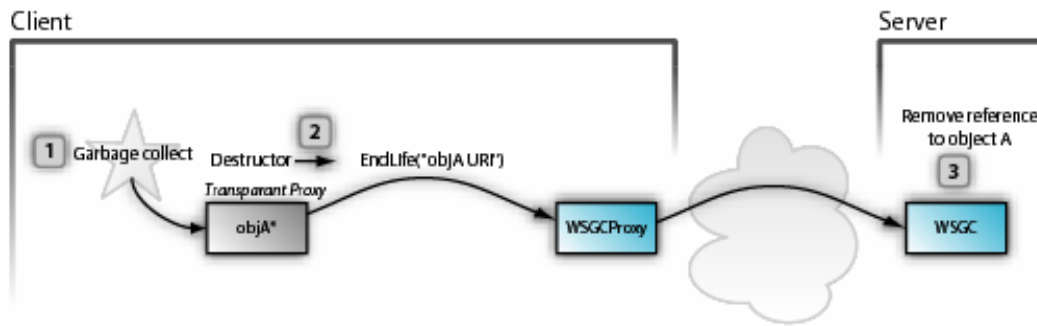
Figure 9 Lifetime synchronization.

Our basic approach is illustrated in Figure9. As explained in the previous section, the server adds an extra reference to an object each time it is requested by a client. We only increment the reference count by a maximum of one for each client. Once a transparent proxy on the client is not referenced anymore, it is destroyed by the local garbage collector (1). This results in the invocation of the destructor of that proxy. The transparent proxy reacts to this event by invoking the *EndLife()* method on a special garbage collector proxy (*WSGCProxy*), giving its URL as argument (2). The message is received at the server (using the mechanisms described earlier), where a special garbage collecting object (*WSGC*) will remove a reference to the corresponding remote object (3). Hereafter the garbage collector of the server can proceed with its tasks. Because the reference count of the object on the server is lowered, the object could possibly be removed in the next run of the garbage collector.

Of course this method does not take into account the unexpected connectivity loss of a client. The unexpected loss of a client will result in eternal life for its referenced objects because it cannot notify the server of object destruction. Since wireless networks are common, portable devices suffer connectivity losses regularly and a complementary solution has to be added. A simple and satisfactory solution is to implement a simple leasing system where the client announces its presence to the server at regular intervals. When the server does not get any life signs for a specified amount of time or after an active poll, it can delete all the references associated with that client. Because life signs can get lost, this solution can fail but this technique detects whether an object can be destroyed with a high probability.

Remote Events

Using the given descriptions, invocation from client to server becomes possible. What is lacking here is a mechanism for notifying clients of events generated by a remote object. This will require the client to act as a (web)server; we assume that firewalls and other filtering mechanisms (e.g. Network Address Translation) are configured to allow all inbound traffic. An easier solution would be for the client to use a polling mechanism but this will not be further considered here since it is not a real eventing system.

In C# (probably the most popular .NET language) the keywords *event* and *delegate* are provided. A (multicast) delegate is a special object that can contain pointers to

methods in other objects, provided that these methods have the same signature as the delegate declaration. These methods can consequently be called all together by triggering the delegate. The event keyword is actually an access modifier on a delegate to prevent external triggering of the delegate. Other objects can subscribe to an event by instantiating the delegate with one of their methods and adding it using the += operator. How these events and delegates are integrated into the previous parts is discussed below (see
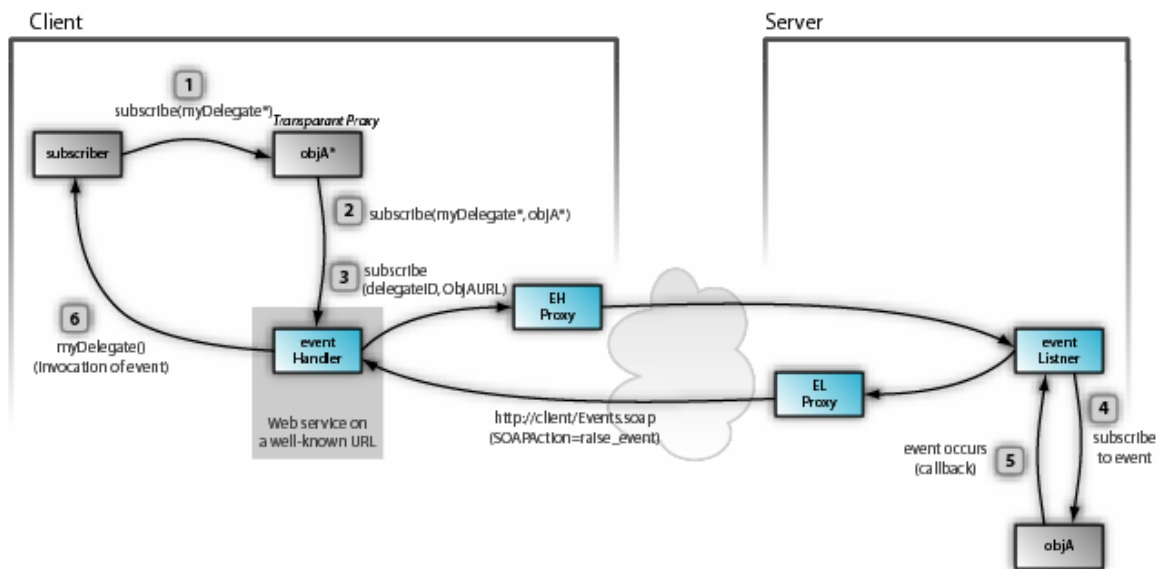
Figure 10).

Figure 10 Remote events.

In the same way that the transparent proxy mimics the interface of a remote object, it also mimics the events published by that object. To subscribe to an event published by the transparent proxy *objA**, one calls the *subscribe()* method with an instance of the appropriate delegate as its argument (**1**). The standard mechanism to subscribe to an event cannot be used because the += operator cannot be overridden (we need this in order to invoke *subscribe()* (**2**)). As a consequence, this part cannot be made completely transparent. Next, the transparent proxy *objA** passes the request to the client's *eventHandler* object (**2**). The *eventHandler* acts as a transparent proxy in the sense that it does the necessary translations of object references to URLs, but it does more than that (see further). The request is then passed to the real proxy (**3**) belonging to the *eventHandler* object, which sends the message to the server. A delegate is identified by an ID number in this stage, so the server can find the right delegate. When the message arrives at the server, the custom sink object (not shown in the figure) routes the request to the *eventListener* object, which subscribes itself to the event in place of the transparent proxy *objA** (**4**). When the event occurs (**5**), the *eventListener* is notified. The *eventListener* then calls its proxy (a custom server proxy) to translate the event arguments
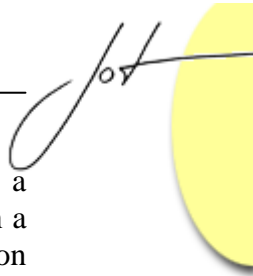
and send them to the *eventHandler* on the client. This is accomplished by running a simple web server [Pra03] on the client and publishing the *eventHandler's* interface on a well-known URL. The *eventHandler* can, if necessary, call the corresponding delegate on the client to raise the event locally (**6**). Thus it will seem that the event has occurred locally. Again, every action originally takes place at the server. The client just gets a view of the activities as they are happening on the server.

If the eventing system as explained above is used and the required server-to-client communication mechanism is in place, we could reuse this facility to make the garbage collection mechanism more efficient. Instead of notifying the server each time a client proxy is destroyed (as explained in the previous section), the server could ask the client which objects may be destroyed – just-in-time – whenever its own garbage collector runs. This would reduce the amount of garbage-related messages because they can then be grouped into larger aggregated messages. Mind that the trivial approach explained here could potentially slow down the server garbage collector due to the delay caused by the requested network message. Furthermore, new infrastructure parts will be needed on the client and server to make this work. We do not discuss this approach further in this article.

5   IMPLEMENTATION OF THE MODULES TO SUPPORT THE PROPOSED CONCEPTS

An implementation of the basic ideas was carried out to prove the feasibility of the proposed concepts. The results of the implementation can roughly be divided into two parts: a C# code generator for the client side proxies and an extension for the .NET Remoting infrastructure in the form of a channel sink and supporting objects.

The code generator was implemented in two steps. First a WSDL generator was developed. It takes one or more existing classes (residing in compiled assemblies) as input and generates WSDL files for the choosen types as output. The outputted WSDL represents the real proxy interface, having reference types mapped to strings. Next, this WSDL is automatically transformed into real proxies using standard provided classes (or tools) in the .NET Framework class library. In a second phase a code generator for the transparent proxies was implemented. This was accomplished using the excellent support for dynamic code generation and compilation of the .NET class library.

All the functionality mentioned was then integrated into one tool which enables one-click generation of all the needed proxies. The functionality needed by all proxies (such as the *objectActivator*) was split off into a separate common library module that has to be included with each client. Our generator tool can be set to output a compiled assembly of proxies, ready to be used. By importing this assembly into a project (in Visual Studio.NET, see Figure 11), the programmer gets a view of all the classes as he would expect them on the server, thus fulfilling requirement 4.
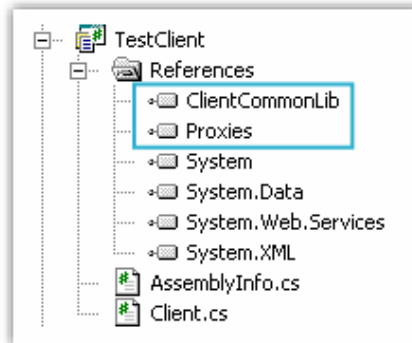
Figure 11 Using the generated proxies.

For now our generator generates transparent proxies only for .NET. Real Proxies can already be generated for other platforms such as Java because they are solely based upon the WSDL file. We have split the code generation into a few steps to facilitate the adaptation of code generation for other (non-.NET) programming languages. Especially the generation of the intermediate WSDL files opens up the possibility of using existing tools to generate real proxies in other languages without having to recode the entire logic. To be even more reusable, we should integrate all neccesary information – extra information to generate transparent proxies – into the WSDL file in the future. This can be accomplished by using a distinct XML namespace.

Development of the channel sink took longer then expected due to the many little pitfalls of the Remoting framework. The channel sink component is responsible for mapping the run-time arguments and return values back and forth to URLs. The reverse mapping is necessary for the WSDL generator, so the mapping part is split off in a separate component. In summary, the channel sink undertakes four steps in intercepting messages:

1. Check the input message and only accept objects implementing *IMethodMessage*. We do not handle constructor messages for example.
2. Adapt the incoming message:
    a. Search for references in the parameter list.
    b. Skip simple messages (containing only primitive types).
    c. Convert the references into real object references by searching the server's hash table. Create a new writable Imessage object, copy the data from the original message and replace the references.
3. Forward the newly created message to the next sink in the chain.
4. Adapt the return message:
    a. If the return type is primitive, the instance is marshaled by value and directly sent back.
    b. If the return type has to be marshaled by reference, a unique ID is generated to be able to construct a valid URL. Next, the instance is published as a web service on this URL and the mapping between URL and real object reference is saved in a hash map, which also places an ex-tra reference to the
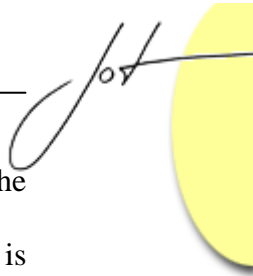
object on the server for use in the distributed garbage collection. Finally the return message is changed with the marshaled return value.

c.   In case of a complex value type with methods, a local copy of the instance is first created and then, the mechanism of b is followed.

Inserting a channel sink in the server formatter sink chain can be accomplished by adding a few lines of code to the server application or even simply by adding some configuration information to the application's standard configuration file. The latter one requires no source code changes to the server. The code below shows the few lines of code that need to be added to the server, assuming that a *SoapFormatterSink* is already present. This shows the low impact on the server, supporting requirement 5.

```
WebServiceMapperChannelSinkProvider mapperProv = new
WebServiceMapperChannelSinkProvider();
serverProv.Next = mapperProv;
```

At the client side the proxies need to be imported (see Figure 11) and some bootstrap code (shown below) needs to be manually inserted by the programmer. This can be done using just a few lines of code. From here on everything is handled behind the scenes through transparent proxies only.

```
ProjectProxy projectP = new ProjectProxy();  // The real proxy
projectP.Url = "http://134.56.76.2:808/MyProject.soap";
Project project = new Project();  // The transparent proxy
project.RealProxy = projectAp;
```

Our implementation was tested against an existing application of a company active in the warehouse automation sector. This automation is accomplished using automated guided vehicles (AGVs). To enable rapid application deployment they developed an inte-grated designer suite offering the basic building blocks of a warehouse application. The suite is fully written using the .NET Framework. It includes generic building blocks for logging, scheduling transports and user interfacing. The user interfacing building blocks communicate with the other parts using .NET Remoting in a client/server fashion.

Our test case was a smart client application that acted as a simplified graphical user interface (GUI) to the warehouse application. The advantage of our approach in this specific case was the possibility to reuse the existing GUI code with only minor adjustments because the generated transparent proxies use the same class names as the full applications. Two objects were relevant in this application, namely *Project* and *Agv*. The operations that were used to do some testing are summarized in Table 3. The generated proxies for the two objects were compiled into an assembly of 20 KiB[3]. The client's common library requires 16 KiB. The measured durations for operation executions are presented in

---

[3] KiB is short for kibibyte, where kibi=2^{10} (an IEC prefix). KB is short for kilobyte, where kilo=10^3 (an SI prefix).

Table 4 below. The table contains measurements using our solution and using the Remoting-Remoting case (using the *HttpChannel*).

| Operation(s)	Description
`string GetName();`	Gets the name of the project
`agv[] GetAgvs();`	Gets an array of 4 AGV's from the project
`void SetSpeed(int s);` `int GetSpeed();`	Sets the speed of one AGV and retrieves it immediately thereafter

Table 3 Test operations.

Operation(s)	Time (ws-rem)	Time (rem-rem)
`string GetName();`	25ms	455ms
`agv[] GetAgvs();`	25ms	8ms
`void SetSpeed(int s);` `int GetSpeed();`	250ms	24ms

Table 4 Performance measurements.

From these limited measurements we can see that performance of our approach is slower than the Remoting-Remoting case. This is probably also due to our less-than-optimal implementation of the concepts. With some optimization we believe that our implementation can deliver better results that are acceptable for small applications. The large delay of the *GetName()* operation (first row in

Table 4), in the Remoting-Remoting case is caused by the dynamic generation of proxies. This type of delay always occurs when invoking the first method on a remote object and has nothing to do with the type of its return value/parameters. This supports our early decision not to port the complete .NET Remoting infrastructure to the .NET Compact Framework. Further tests showed no performance decline when offering many objects (tested with arrays of 3, 30, 300 and 3000 objects).

6   RELATED WORK

The consuming of web services on mobile devices has been emerging due to the growing availability of Wifi-, or Bluetooth-enabled PDAs and smart phones. These web services have been mainly limited to simple request/response services, such as obtaining weather or news information. To enable remote events, as discussed earlier, a mobile web server will be needed. A proposal to implement such a server, keeping in mind the resource constraints, is given in [Pra03]. To lower the device's requirements, some constraints were introduced. One of them is to allow only simple SOAP types. This would not be a problem in integrating it with our solution, because we do not use complex SOAP types.
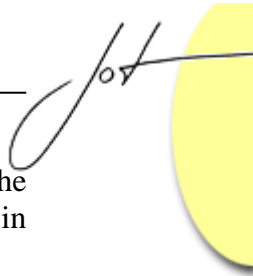
In [Cam00], techniques for optimizing the performance of Java RMI are proposed. The optimizations are made with wireless communication and resource-constrained devices in mind, making Java RMI more suitable for mobile devices.

An approach to optimizing the use of web services on resource-constrained devices by applying specialized code generation techniques is presented in [Eng]. Also, some runtime optimizations are implemented using the *gSOAP* environment, which is portable to most platforms including Pocket PC (which can run the .NET Compact Framework).

Middsol [Mid] provides standard CORBA inter-process communication for the .NET Compact Framework. This support is provided in the form of an assembly (520 KiB) that needs to be included on the mobile client. While being very useful, this solu-tion does not allow one to directly connect to .NET Remoting objects.

An approach that enables communication between the .NET (Compact) Framework and long-lived embedded devices is proposed in [She04]. It isolates applications from the underlying wire protocol by using application-level bridges. This is similar to what we are accomplishing by using independent proxies on the client.

The approach in [Vei04] enables the .NET Compact Framework to communicate with a .NET Remoting infrastructure using bridges based on web services. The main focus of the paper is on object replication on mobile devices to enable connectionless operation and boost performance. As in our approach, automatic proxy generators are provided.

7   CONCLUSION

To enable the introduction of smart clients (PDAs, smart phones) into existing distributed applications, we proposed an approach that dynamically maps web services to .NET Remoting. This approach enables the quick development of applications that interact with remote objects, solely using the .NET Compact Framework. By presenting a transparent interface using proxies, the programmer does not have to worry about any communication details. The solution is fully generic so it can be used for any existing ap-plication without specific modifications.

Using our code generation tool, proxies are generated fully automatically simply by selecting the required classes in an assembly. Thus a complete representation of the needed server-objects becomes available at the client in the form of proxies that mimic these objects. This also enables the reuse of possibly existing (.NET Full Framework) client code. The impact on the server is minimized by the implementation of all necessary logic using just one sink object (a Remoting extension). This sink can be inserted into the .NET Remoting infrastructure by adding as little as three lines of code or even simply by modifying the application configuration file, without influencing the rest of the application. In addition, the portability to other client platforms is easy. It would only require an extension of the C# code generator for the transparent proxies. The server side requires no modifications.

To refine the solution, two paths can be further pursued. First, the implemented modules can be completed by including an implementation of the proposed garbage collection and eventing concepts. Secondly, we can search for good solutions to handle the more efficient communication of frequently used classes such as collections and, more in general, all classes common to the class libraries of both client and server.

REFERENCES

[Alb01] B. Albahari, P. Drayton, and B. Merrill, C# Essentials. O'Reilly, 2001.

[Alm01] J. P. Almeida, L. Ferreira, and M. J. van Sinderen, "Web services and seamless interoperability", 2001 [Online], Available: http://wwwhome.cs.utwente.nl/~pires/ publications/eoows2003.pdf

[Boo03] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, "Web services architecture," 2003. [Online]. Available: http://www.w3c.org/TR/2003/WD-ws-arch-20030808/

[Box00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (soap) 1.1," 2000. [Online]. Available: http://www.w3.org/TR/2000/NOTESOAP 20000508/

[Cam00] S. Campadello, O. Koskimies, K. Raatikainen, and H. Helin, "Wireless java rmi."

[Eng] R. van Engelen, "Code generation techniques for developing light-weight xml web services for embedded devices.", [Online], Available: http://websrv.cs.fsu.edu/~engelen/SACpaper.pdf

[Mcl03] S. McLean, J. Naftel, and K. Williams, Microsoft .NET REMOTING. Microsoft Press, 2003.

[Mic] "Microsoft .NET framework development center." [Online]. Available: http://msdn.microsoft.com/netframework/

[Mic2] "Microsoft Rotor - Shared Source Common Language Infrastructure", [Online], Available: http://msdn.microsoft.com/net/sscli

[Mid] MiddSol, "Middleware solution for integrating .net, j2ee and corba.", [Online], Available: http://www.middsol.com/MinCor/index.html

[Pla95] D. Plainfossé and M. Shapiro, "A survey of distributed garbage collection techniques.", 1995, [Online], Available: http://mega.ist.utl.pt/~ic-arge/arge-96-97/artigos/

[Pra03] D. Pratistha, N. Nicoloudis and Simon Cuce, "A micro-services framework on mobile devices," 2003. [Online]. Available: http://plato.csse.monash.edu.au/MobileWebServer/pervasive3.pdf
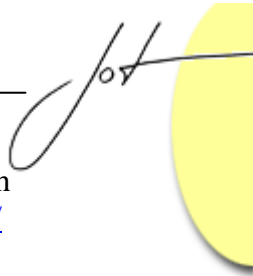
[She04] R. Shenoy and K. Moore, "Sustaining the integration of long-lived systems with .NET", 2004, [Online]. Available: http://www.hpl.hp.com/techreports/2004/HPL-2004-133.pdf

[Sun] "Java RMI" [Online]. Available: http://java.sun.com/products/jdk/rmi/

[Vei03] L. Veiga and P.Ferreira, "Complete distributed garbage collection: an experience with Rotor", [Online], Available: http://csce.unl.edu/~witty/sp2004/csce496/repository/upload/10.pdf

[Vei04] L. Veiga, N. Santos, R. Lebre and P.Ferreira, "Loosly-Coupled, Mobile Replication of Objects with Transactions", 2004, [Online], Available: http://www.gsd.inesc-id.pt/~pjpf/icapds-2004.pdf

[W3c02] "W3c web services activity." [Online]. Available: http://www.w3c.org/2002/ws

[W3c03] R. Chinnici, M. Gudgin, J.-J. Moreau, and S. Weerawarana, "Web services description language (wsdl) version 1.2," 2003. [Online]. Available: http://www.w3.org/TR/2003/WD-wsdl12-20030303/

[Wig03] A. Wigley and S. Wheelwright, Microsoft .NET Compact Framework (Core Reference). Microsoft Press, 2003

About the authors

Bert Vanhooff received his master's degree in Computer Science in 2004 from the K.U. Leuven and is now a PhD student at the K.U. Leuven. His main research interests are situated in the area of Model Driven Development and the application of that paradigm to Component Based Software Engineering and model transformations. He can be reached at bert.vanhooff@cs.kuleuven.be.

Davy Preuveneers received his MS degree in Computer Science in 2002, and his MS in Artificial Intelligence in 2003 from the K.U. Leuven. Since 2003, he is working towards his PhD at the Department of Computer Science of the K.U. Leuven. His research is focussed on context-aware service interaction and adaptation, in particular in mobile and pervasive computing environments. Contact him at davy@cs.kuleuven.be.

Prof. Dr. Ir. **Yolande Berbers** received her PhD degree from the K.U. Leuven in 1987. She is currently a full professor at the Department of Computer Science at the K.U. Leuven and a member of the DistriNet research group. Her research interests include software engineering for embedded and real-time systems, model-driven architecture, context-aware computing, distributed and parallel systems and distributed computing, and multi-agent systems with emergent behaviour. She runs several projects in cooperation with other universities and/or industry. She can be reached at yolande@cs.kuleuven.be.