# Using Web Services on Mobile Devices to Transparently Access .NET Remoting Objects

Bert Vanhooff
K.U. Leuven
Celestijnenlaan 200A
B, 3001, Leuven
bert.vanhooff
@cs.kuleuven.ac.be

Davy Preuveneers
K.U. Leuven
Celestijnenlaan 200A
B, 3001, Leuven
davy.preuveneers
@cs.kuleuven.ac.be

Yolande Berbers
K.U. Leuven
Celestijnenlaan 200A
B, 3001, Leuven
yolande.berbers
@cs.kuleuven.ac.be

## 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 to seamlessly integrate with an existing .NET Remoting application. In this paper, we propose an approach that leverages the present support for web services to make such integration possible. Our solution dynamically maps back and forth between .NET Remoting and web service messages. An implementation of this solution resulted in a set of tools and components that can readily be used to start developing mobile clients that interoperate with existing .NET Remoting applications.

## Keywords

.NET Remoting, Web Services, .NET Compact Framework, Interoperability, Mobility

## 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]. It consists of an execution environment for applications and a comprehensive class library. To support the development of distributed applications, .NET Remoting [Mcl03] was included. This is an extensible middleware infrastructure intended to simplify the development of distributed systems. It is comparable to Java RMI [Sun].

The .NET Compact Framework [Wig03] is a slimmed down version of the .NET Framework made to run on embedded devices like PDAs or 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. The entire namespace of the Remoting classes was removed. As a consequence, the only high-level communication facility present in the .NET Compact Framework is provided in the form of a number of classes to support the invocation of web services.

Web services can be interpreted in a broad sense as all means by which a service can be offered by one application and used by another by leveraging Internet technologies. When we refer to web services [W3c02], [Boo03], we specifically refer to SOAP (Simple Object Access Protocol) [Box00] over HTTP and WSDL (Web Service Description Language) [W3c03]. SOAP is the XML based protocol of the messages sent by a web service, while WSDL is the

***.NET Technologies'2005** conference proceedings,*
***ISBN** 7/,75832,/0,0*
Copyright UNION Agency – Science Press, Plzen, Czech Republic

XML language used to describe the interface offered by such a service.

The absence of .NET Remoting in the .NET Compact Framework puts some serious constraints on the development of connected smart clients when these clients need to access remote objects on an existing server. These constraints, which are further discussed in the next sections, cannot be overcome by using the standard web services support available in the .NET Compact Framework.

In this paper, we focus on the problems that are associated with the development of new smart clients that need to be integrated with existing .NET Remoting applications and we offer a solution to these problems. The rest of the paper is organized as follows. Section 2 briefly introduces the .NET Remoting and web services infrastructure for the purposes of formulating the problem in more detail and it ends with a list of requirements for a good solution. Section 3 gives an overview of the basic infrastructure that will be used to solve the problem, while Section 4 explains additional mechanisms employed to support distributed garbage collection and remote events. Section 5 gives an overview of the implemented concepts and presents the results of a small test case. In Section 6, some related work is presented and finally, Section 7 concludes the paper with suggestions for future improvements.

## 2. DISTRIBUTED APPLICATIONS IN .NET

As mentioned in the introduction, .NET offers .NET Remoting and web services for developing distributed systems. This section introduces the parts of these two technologies that will be used further in the paper and it points out the constraints involved when using web services instead of .NET Remoting. To conclude this section, a set of requirements for a solution that overcomes some of these constraints is given.

### .NET Remoting

.NET Remoting simplifies the development of distributed systems by offering an extensible infrastructure that permits objects not residing 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 stack-based) mechanism. Therefore, each message from a local (client) object to a remote (server) object will be intercepted using a proxy pattern. A message, which can for example represent a method or constructor call, will be transformed into an *IMessage* object by the proxy. This object contains all the necessary information needed to reconstruct the original call.

After passing through the proxies (at this point there are two of them), the *IMessage* object is further propagated through the .NET Remoting infrastructure. This part contains several so called *sink chains*, which are series of concatenated 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 custom sink objects. Some sink objects are provided by default. They include a formatter sink to serialize the IMessage data and a transport sink to take care of the actual message transport. Each sink chain, containing instances of these two default sinks, is part of a channel. The channels are the first components in the .NET Remoting infrastructure that get to see incoming messages and the last to see outgoing messages. Each channel is named after its location and the transport mechanism that it supports (e.g. *TcpClientChannel*).

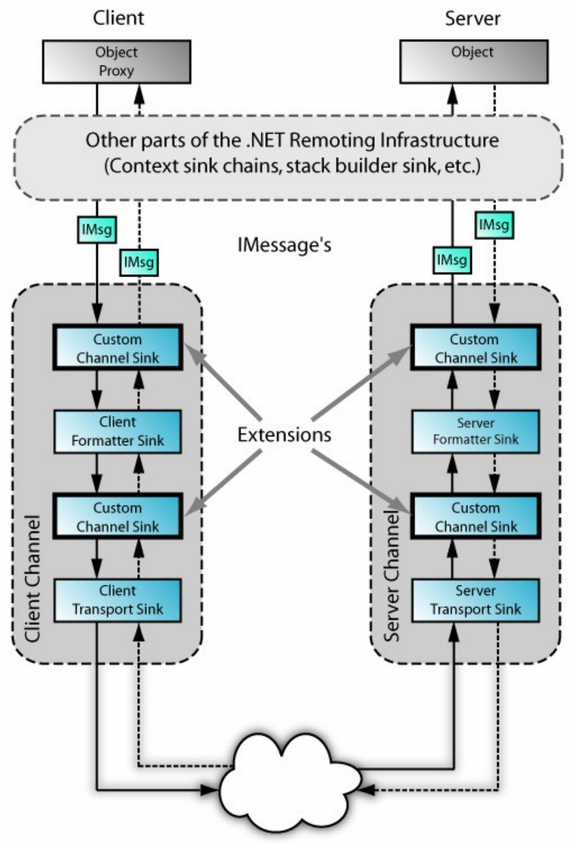

**Figure 1. A limited overview of the .NET Remoting architecture**

Another set of sink chains exists besides the ones belonging to the channel sinks. Depending on the chosen sink chain, different categories of *IMessages* will be intercepted. By choosing the *server object sink chain*, only *IMessages* originating from a specified object will be seen. On the other hand one can choose a *channel sink chain* (discussed in the paragraph above) to intercept every message from every object that uses that channel.

The extension mechanism, using custom sink objects, can be used to 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.

A high-level overview of a limited part of the .NET Remoting architecture can be found in Figure 1. It 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 proxy 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 chain of custom message sinks (only one is shown to save space). The message then moves further to the *Client Formatter Sink*, where it is serialized. After that, another series of *Custom Channel Sinks* and, at last, the *Client Transport Sink* are passed. This last sink physically sends the message to the server using some kind of network technology. When the message is received at the server, 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).

.NET Remoting also offers solutions for considerations such as object lifetime management and object activation, but these will not be discussed here.

## Web services

One of the advantages of using .NET Remoting, besides its extensibility with message sinks, is its direct support for offering web services through its infrastructure. Remote objects can be accessed – in a limited way – using web services, meaning that all .NET Remoting extension mechanisms can be used while handling a web service request. This means that the whole client side in Figure 1 could be replaced by a web service client. However, some functionality, as it is available when using a .NET Remoting client, will be lost due to the inherent limitations of standard web services [Alm01].

The main limitation in this case is that they have a procedure oriented architecture instead of an object-oriented architecture. The full fidelity of an object graph at a server cannot be seen by a web service client because object references cannot be passed. When accessing a remote object through a web service in .NET Remoting, the caller can only call methods that return primitive or structured data-types. As a consequence, he cannot get out of the scope of the initial object because any call to a method, which would normally return a reference to an associated object, will only return the data contained in the associated object and not the object reference itself.

In summary, when web services are used to access remote objects, the objects need to be published on well-known URLs in advance and they may not be removed during the application's lifetime. Other objects that are created during the operation of the system will not be accessible. Consequently, an application offered as a set of web services has to have a static object graph, at least for the objects published as web services. More specifically an object that is published as a web service should not be deleted as this would result in, unanticipated, access faults. In addition newly created objects cannot be directly accessed by web service clients. Mind that data present in newly created objects can 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 al lot of cumbersome coding. There are standard tools available to generate these proxies for a remote object. Whenever the tools encounter a method that returns or accepts an object, this object will be mapped to a complex SOAP data structure. Consequently, for these proxies the very notion of an object disappears.

An additional restriction is the inability to let the server initiate communications, for example in the case of notifying the client of an event occurrence. Client and server are not peers as is the case with .NET Remoting.

These limitations, along with the dynamic nature of most object graphs, make the web service support for .NET Remoting inadequate for developing smart clients with the same capabilities as full .NET Remoting clients. This becomes even more important when extending an existing .NET Remoting application that was not originally designed for extension to web services. The focus of this paper is on extending such applications.

In the next subsection, we state the requirements that need to be fulfilled by a useful solution.

## Requirements

Suppose a server running .NET Remoting is exposing some of its objects for remote access. All .NET Remoting clients can access these objects as if they were local to them. If one wants to port such a client to

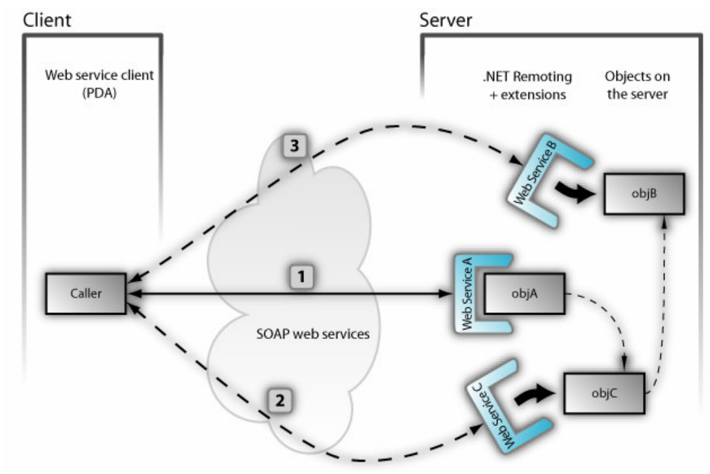

**Figure 2. Dynamically exposing objects as web services.**

run on a smart device, major problems will occur because, apart from web services (with their already discussed shortcomings), the .NET Compact Framework lacks support for accessing these remote objects. Therefore we have to figure out an alternative approach for interacting with remote objects that offers most of the .NET Remoting capabilities. A concrete list of the requirements we expect a good solution to meet is given here:

1. make the object graph on the server navigable from the client;
2. enable the client to refer to a specific object on the server;
3. enable method calls on remote objects (with object references both as parameters and as return type);
4. make interactions as transparent as possible and hide communication details;
5. enable callbacks from the server;
6. enable fast development of new clients;
7. minimize the impact on existing applications.

These requirements need to be fulfilled by reusing large parts of the already available infrastructure on both the client and the server platform. The client implementation must take into account the typical limitations of embedded devices (small memory size, limited processing power, etc.). This last requirement makes the porting of the whole .NET Remoting infrastructure to the .NET Compact Framework an unrealistic option.

## 3. USING WEB SERVICES TO ACCESS REMOTE OBJECTS

In this section we explain the approach we take to making remote objects available to clients who run the .NET Compact Framework. Requirements 1, 2, 3 and 4 will be addressed here. Requirement 5 will be discussed in Section 4 while requirements 6 and 7 will be addressed throughout all the next sections and especially in Section 5.

In the current section we will explain how URLs can be used as object references and web services to enable basic communication.

### Basic approach

As mentioned before, .NET Remoting can publish a degenerated version of the public interface of a remote object through a web service on a well-known URL. We will use this capability and modify the way of using web services to overcome their inherent limitations. The envisioned idea in this paper is to make the publication of a remote object as a web service happen dynamically whenever a client requests an operation which returns a remote object. Furthermore, to enable navigation to another object, the URL that uniquely identifies that remote object will be passed in SOAP messages. This will in fact indicate the web service of that object though it can be mapped one-to-one onto the actual object, effectively replacing the real object reference. The idea is visually represented in Figure 2.

The figure presents a graph of three interconnected objects, *objA*, *objB* and *objC*. The starting object *objA* will be accessible using a web service on a well-known URL (1). By invoking methods on this object, one can navigate to the other objects in the graph as follows. Whenever the client calls a method that should return a reference to another object (which cannot be transported using standard web services), this 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

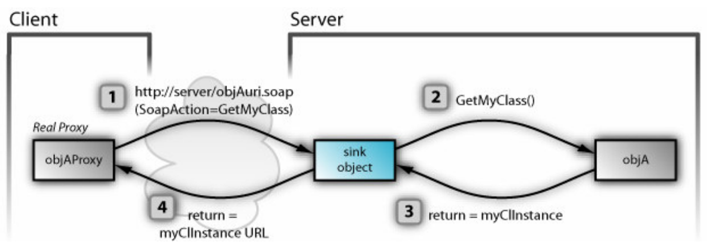

**Figure 3. Invoking a method.**

```
http://145.34.67.10:1200/[type:MyClassLib.MyClass][853b9985]
        server location        object type       unique object
                                                    reference
```

**Figure 4 Our web service URL format**

new object (2). In this way every object in the graph can be reached (3), effectively enabling navigability.

To keep object access as transparent as possible to the client, each remote object will be represented by a proxy object to hide communication details. In this way the client thinks it is working with local objects, which basically is what .NET Remoting is also accomplishing.

This approach will require adjustments on both the client (proxies) and the server (.NET Remoting extensions). In the next section, we present an elaboration of the general idea by using a method call scenario.

## Remote method calls

To make invocations (made by the caller on the client) transparent, two proxies will collaborate to represent a remote object on the client. The first proxy, from here on called the *transparent proxy* will mimic the interface of the remote object. The second proxy referred to as the *real proxy*, will hide communication details. The names chosen for these proxies were inspired by the names of the proxies in .NET Remoting. In this subsection we refer only to the real proxy. These two proxies reside on the client. The server side will also need an extension to be able to handle the client's requests. This extension will be a custom message sink object, inserted on top in the server channel sink.

The real proxy can be partially generated by extracting the interface of its corresponding class. However, some modifications to this interface are necessary when generating the proxy. These have to do with the limitations of web services concerning the transportation of object references. As mentioned in Section 1, web services cannot transport objects (or better: references to objects). Only simple and structured value types can be transported directly. 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. At runtime, this string will contain an object reference represented by a web service URL (see Figure 4). An example of the different possibilities is given in Table 1.

| 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**

We use three different methods to marshal different types. Objects that are normally marshaled by reference by the Remoting infrastructure are marshaled by reference using the URL representation as presented in Figure 4. Primitive types are marshaled by value and can be transported directly using SOAP messages. Complex value types (*struct*s without methods in C# [Alb01]) can also be transported directly. The last case occurs when a complex value type contains extra methods (also *struct*s). We chose to make a local copy of the instance on the server and then marshal it by reference. Another (maybe better) way to achieve a correct transport of these complex types is to transport only the data in the instance using marshal by value. The data can than be loaded into a corresponding type instance on the client that would act as a virtual proxy. It does not communicate with the server but does represent a server type. The latter solution would be more complicated to implement, while the first method can use the existing marshal by reference facility.

If a method does not contain non-transportable types, it can be offered in the interface unmapped and 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

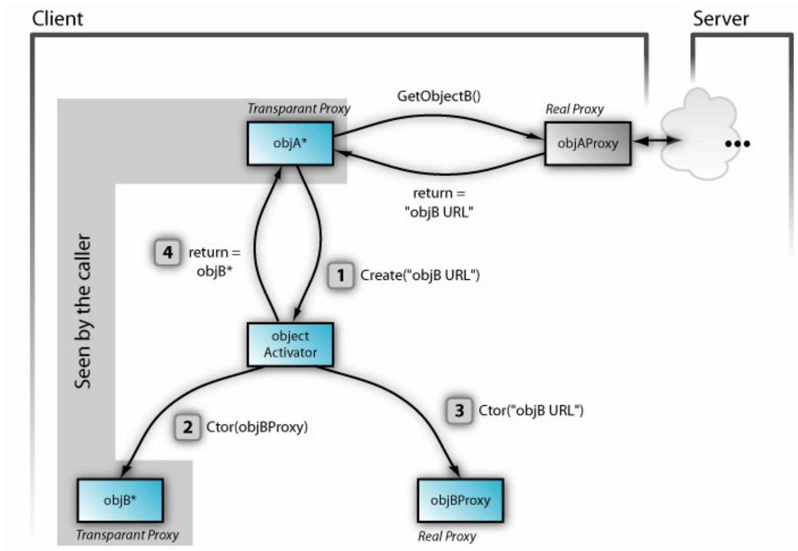

**Figure 5. Using two proxies on the client to provide maximum transparency.**

the client (handled in its proxies) and on the server (using a sink object).

A case where the return type is mapped will be discussed here. Suppose one wants to invoke the method `MyClass GetMyClass()` on a remote object that we can reach via a known URL. Through the mapping mechanism this method will be exposed as `String GetMyClass()`, and will be available as such in the proxy on the client. The sequence of steps that will take place when calling that method is shown in Figure 3.

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 SOAP message then arrives at the server and is accepted by the .NET Remoting infrastructure, where it is automatically deserialized into an *IMessage* object containing the same information. After that, it is inserted into the right sink chains. This also means that our custom sink object will get a chance to process the *IMessage*. In this case, the sink can just pass the *IMessage* further up the chain so that the call can eventually be invoked (2). On the other hand, if the method contains mapped parameters, its arguments will contain URLs that indicate other objects. These URLs should first be replaced by the actual object references (which are known on the server) before the *IMessage* is further propagated. The result 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 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 (see Section 4). 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 the caller — which will in practice be the transparent proxy (see next subsection). 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 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 type used. This does not introduce any limitation in our case. Proxy generation at design time will actually boost performance by taking away the processing cost of generating proxies at run time. While it does enable basic communications, the use of the real proxy directly 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 added 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 real proxy. The interface of the transparent proxy will mimic the object on the

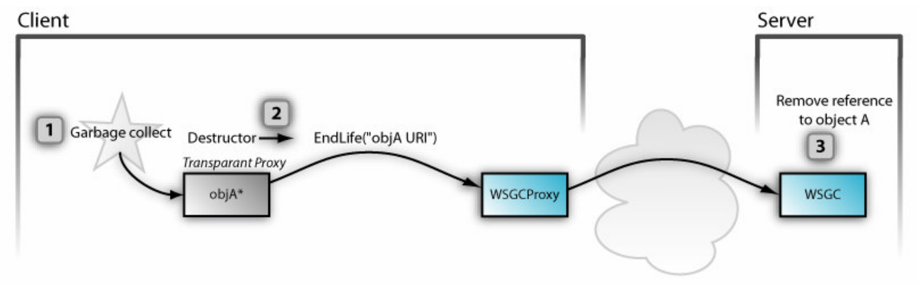

**Figure 6. Simple distributed garbage collection.**

server that it represents, effectively providing transparency. 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 and forward 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 beginning of this section. Figure 5 shows a general model of the structure.

The scenario presented in Figure 5 starts when the transparent proxy *objA** (indicating 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 3 ended by returning an URL to the caller, 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:

**1.** Upon receiving a URL, the transparent proxy needs to create the necessary proxy objects that will enable the client to transparently work with the new object's web service. It therefore sends a `create()` message to the `objectActivator`.

**2.** 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.

**3.** A real proxy to directly interact with the web service will also be created.

**4.** Eventually the newly created transparent proxy `objB*` is given back to `objA*`, whichever object invoked its method caller.

## 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 5 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 when they are still in use. Without further precautions being taken, the first case would apply to the approach explained so far. Whenever a client gets a reference to an object on the server, the object's local life cycle (the life cycle of its proxy on the client) will not be known to the server, which will result in an object that lives eternally. Note that we will 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.

A method for solving this problem of having remote objects that live eternally is to just let the garbage collector on the client do its work on the proxies and, whenever a transparent proxy is destructed, to notify the server of this event. This technique will work well in our specific case. 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. Our basic approach is illustrated in Figure 6.

**1.** A transparent proxy on the client is not referenced anymore and is destroyed by the local garbage collector.

**2.** This results in the invocation of the destructor of that proxy. The transparent proxy will react to this by invoking the `EndLife()` method on a special *garbage collector proxy* (`GCProxy`), giving its URL as argument.

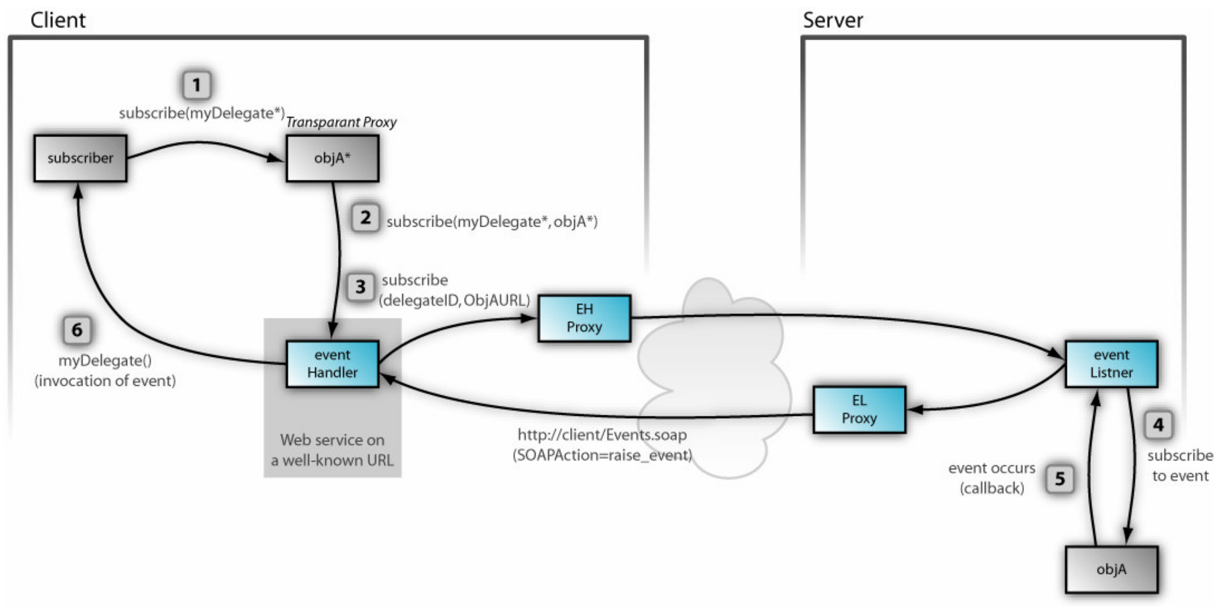

**Figure 7. Distributed events.**

**3.** 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. Hereafter the garbage collector of the server can proceed with its tasks. Because the reference count of the object on the server is now lowered, it 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 now result in the eternal life of its referenced objects because it cannot notify the server of object destruction. Since wireless access is common with portable devices and can suffer connectivity losses regularly, a complementary solution has to be added.

The easiest way to prevent the creation of indestructible objects 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 it can delete all the references associated with that client.

So far, the requirement 5 is still missing. It is not yet possible for the server to initiate contact with a client, for example to send a notification, as would have been done in an event based application. A solution for handling such events will be proposed in the following section.

## 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. An easier solution would be for the client to use some sort of polling mechanism, but this will not be considered here since it is not a real eventing system. Up to this point the solutions have been given in a more or less platform independent manner in the sense that they could be implemented either on a .NET or on a Java platform (using other mechanisms at the server). The way events are supported will be specifically targeted to .NET, using *events* and *delegates*.

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, given 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 7).

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 cannot be used because it cannot be overridden. 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` is a transparent proxy that does the necessary translations of object references to URLs. 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 Figure 7) routes the request to the `eventListener` object, which subscribes itself to the event in place of the transparent proxy (4). When the event occurs (5), the `eventListener` is notified. The `eventListener` then calls its 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.

## 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 sink object 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 corresponding WSDL files as output. It also takes care of the mapping of non-transportable types. Next, this WSDL is automatically transformed into real proxies using standard provided classes 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 was split off into a separate common library module that has to be included with each client.

The 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), the programmer gets a view of all the classes as he would expect them on the server, thus fulfilling requirements 6 and 7.

Splitting the code generation into a few steps facilitates the adaption of the application to generate code 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 re-code the entire logic.

Extending the .NET Remoting behavior did not prove to be as easy as expected. There turned out to be many more subtleties in choosing the right extension mechanism than one would expect. The .NET Remoting introduction in this paper only touches on the many extension possibilities. A suitable extension mechanism was finally found: a custom channel sink inserted above the predefined server formatter sink. This component is responsible for mapping the runtime arguments and return values back and forth to URLs. It therefore shares some functionality with the WSDL generator.

Our channel sink undertakes four steps in intercepting messages:

**1.** Check the input message. Only accept IMethodMessages. We do not treat constructor messages for example.

**2.** Adapt the incoming message:

- Search for references in the parameter list.
- Skip simple messages (containing only primitive types).
- Convert the references into real object references by searching the server's hash table. Create a new writable IMessage, 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:

- If the return type is primitive, the instance is marshaled by value and directly send back.
- 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 extra 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.
- In case of a complex value type with methods, a local copy of the instance is first created and

then, the mechanism of the former bullet 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 applications standard configuration file. This shows the low impact on the server, again supporting requirements 6 and 7.

The 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 integrated 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.

Our test case was a smart client application that acted as a simplified user interface to the warehouse application. Two objects were relevant in this application, namely `Project` and `Agv`. The operations that were used to do some testing are summarized in Table 2. The generated proxies for the two objects were compiled into an assembly of 20 KiB[1]. The client's common library requires 16 KiB. The measured durations for operation executions are presented in Table 3 below. The table contains measurements using our solution and using the Remoting-Remoting case (using the *HttpChannel*).

| **Operation(s)** | **functionality** |
|---|---|
| `string GetName()` | Gets the name of the project |
| `agv[] GetAgvs()` | Gets an array of 4 AGVs from the project |
| `SetSpeed(int s)` `int GetSpeed()` | Sets the speed of one AGV and retrieves it thereafter |

**Table 2. Test operations**

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

**Table 3. Performance measurements**

From these results we can conclude that the performance penalties are acceptable. The large delay of the `GetName()` operation, 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 (see Subsection 2, Requirements) to the .NET Compact Framework.

## 6. Related work

The consuming of web services on mobile devices has only just recently been emerging due to the growing availability offering of Wifi-, or Bluetooth-enabled PDAs and smart phones. These web services have been mainly limited to simple services, such as obtaining weather or news information.

To enable remote events, as discussed in Section 4, 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 interprocess 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 solution 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 handles about isolating 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.

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

## 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 application without specific modifications.

Using our code generation tool, proxies are generated fully automatically simply by selecting the needed 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. The impact on the server is minimized by the implementation of all necessary logic using just one sink object. 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 should be 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 could be further pursued. First, the implemented modules could be elaborated by including an implementation of the proposed garbage collection and eventing concepts. Secondly, we could 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.

## 8. 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