

Sustaining the Integration of Long-Lived Systems with .NET

Rajesh Shenoy and Keith Moore

Hewlett-Packard Laboratories, 1501 Page Mill Rd, MS 1203, Palo Alto, CA 94304, USA
{rajesh.shenoy, keith.moore}@hp.com

Abstract. The continued evolution of web services infrastructures like .NET complicates their integration with long-lived devices (such as printers, cameras, scanners, etc.). The issue is that the lifetime of these devices far exceed the effective lifetime of the clients and the protocols in vogue when the devices were deployed. While upgrading device firmware to track the standards might be theoretically possible, in practice it is not. This paper presents a novel approach for sustaining the integration of long-lived systems with .NET applications using a C# application-level bridge. This approach exploits the inherent stability of the programming language APIs on the .NET platform, and uses code auto-generators to minimize the effort to deploy stable protocols on evolving platforms. We demonstrate the technology by connecting .NET applications resident on handheld and desktop computers to applications resident on a remote printer.

Keywords: .NET, C#, ORB, IDL, Compact Framework, Web Services

1 Introduction

A problem for any developer of long-lived devices (such as printer, scanners, cameras, etc.) is how to connect these devices to the rapidly evolving desktop and handheld environments (such as WindowsXP and WindowsCE). The problem is not which protocol to put in the device, but rather whether that choice is appropriate even a few years from now. With the advent of web services standards, it is tempting to think that heterogeneous systems can interoperate using common communication protocols [1]. The reality is that the lifetime of long-lived devices far exceed the effective lifetime of the clients and the particular interoperability protocols that were in vogue when the devices were deployed. Many hope that consortia such as WS-I [2] will address interoperability issues between J2EE [3], [4] and .NET [5]; however, these consortia are still addressing simple SOAP message exchange [2]. Even if an agreement is reached on a set of protocols, there will continue to be a reason¹ to evolve the protocols (e.g., WS-Security, WS-Routing etc.) and the protocols will evolve faster than these deployed devices. Printers, for example, often outlive the desktop platform to

¹ One reason that will always exist is vendor differentiation.

which they are attached, and hence there is a fundamental issue in getting stability while the platform, protocols, and infrastructure evolve.

While it continues to be tempting to trust that stability will come to object communication protocols and that programming languages will have a common inter-language calling convention, our experience is that the only point of stability on a client is the programming language and associated class libraries that all applications depend on.

This disappointing result led us to reconsider how to provide access to long-lived devices and components from .NET. Rather than trying to dictate a common wire protocol, or even to try for .NET interoperability with multiple vendors' products, we decided to aim for application level interoperability. To achieve application-level interoperability, a device access library is brought to the .NET platform. The access library is auto-generated and communicates with the device using a controlled, stable protocol.

The key insights of this work are:

- Stability can be accomplished by moving interoperability to the application level, rather than purely at the protocol level.
- Application-level interoperability can be achieved by using a controlled protocol and tracking the interoperability through code generators.
- The best approach for bridging from a .NET platform to remote native applications through a controlled protocol is by deploying a C# Object Request Broker on the .NET platform.

The rest of the paper is organized as follows. The different approaches for sustaining interoperability with .NET applications written in C# are described in Section 2. We evaluate the different approaches in Section 3. In Section 4, we describe our solution, the C# bridge and its implementation. Our demonstration of the use of the bridge is described in Section 5. Related work is discussed in Section 6 and conclusions in Section 7.

2 Approaches for C#.NET interoperability

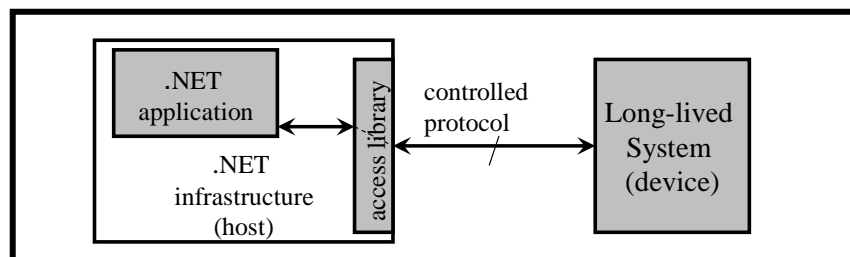


Fig. 1. Access library and proprietary protocol for sustaining integration of long-lived systems with .NET

In this section, we give an overview of existing approaches for C#.NET interoperability at the application layer. Unlike protocol interoperability where the .NET application communicates directly with a wire protocol, for application layer interoperability an access library is used. As Fig. 1 shows, the long-lived system exports an access library that is hosted on the .NET platform. This device access library exposes interfaces presented by the device, and interacts with the device using a controlled protocol². This could be a version of a standard protocol such as SOAP [6] or IIOP [7], which has been frozen until the device vendor decides to upgrade the protocols. The .NET applications are developed using the access libraries, without being aware of the controlled protocol. Since the access library is in the control of the device vendor, the protocols can be kept stable even as the platforms evolve. Since the programming language and function call mechanism on the .NET platform is far more stable than the wire protocols, the access library need not be changed frequently. In contrast, if there is no access library, the long-lived system shown in Fig. 1 has to be directly connected to the .NET application. This requires the long-lived system to implement the protocols that are currently supported by the .NET infrastructure and the device vendor must keep pace with the new protocols as the .NET infrastructure evolves.

Achieving application-level interoperability requires installing an access library onto each client that needs access to the device or consolidating on a server (three-tier architecture). However, from our experience with device drivers, this installation is far less expensive than tracking wire protocols.

As the platform evolves, in the worst case, the access library has to be recompiled for the new platform. However, the platform vendors typically provide migration tools for moving applications to the new platform and these tools could be leveraged for the device access library. The important fact is that the device can remain unchanged as the platform evolves since the protocol is stable. To simplify code-generation and tracking of the device exposure, we use IDL [8] to specify the interfaces that are exposed from the device.

We now give an overview of the approaches for creating the access library. In the following discussion, we use the term “managed” code in the same sense as the .NET Common Language Runtime (CLR) defines it; as code, which is in the control of the .NET runtime (e.g. C#). “Unmanaged” or “native” code is the code that is not under the control of the .NET run time (e.g. Java, C++).

The approaches we consider are

- Bridging managed and unmanaged code
 - Platform Invoke Services (P/Invoke) [9],
 - Managed C++ [10],
 - COM Interop [11],
 - Localhost XML Web Services [12],
- NET Remoting [13] and
- C#ORB.

The first four approaches require assumes that the access library can be generated in the unmanaged form (Java or C++) and address bridging from the managed to this

² We use the term “controlled protocol” in this paper to mean a protocol that is stable and controlled by the long-lived system vendor for the lifetime of the system.

unmanaged code. These access libraries contain a proxy for the remote object and a remote procedure call library (RPC library) to make the remote call using the controlled protocol. However, for pure .NET applications to communicate with the device, the access library has to be exposed using C#. As seen in Fig. 2, this requires an “interop” layer between the native (C++) application and the managed (C#) code. The four approaches differ in the mechanism used to provide the “interop” layer.

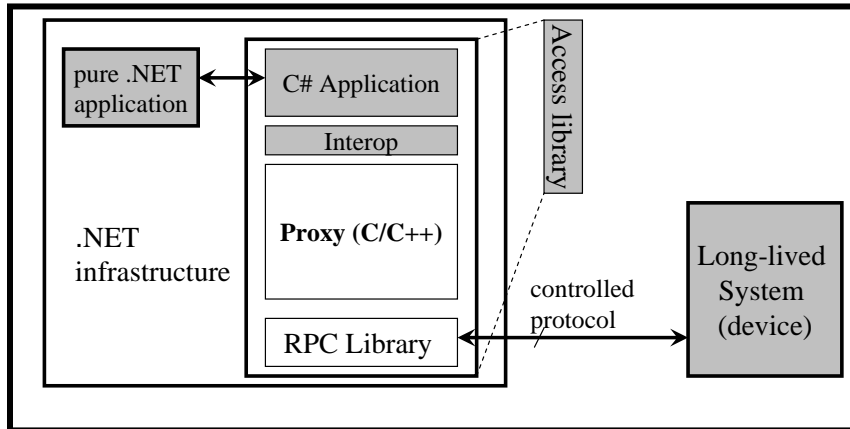


Fig. 2. Managed-Unmanaged bridging for creating access library on .NET

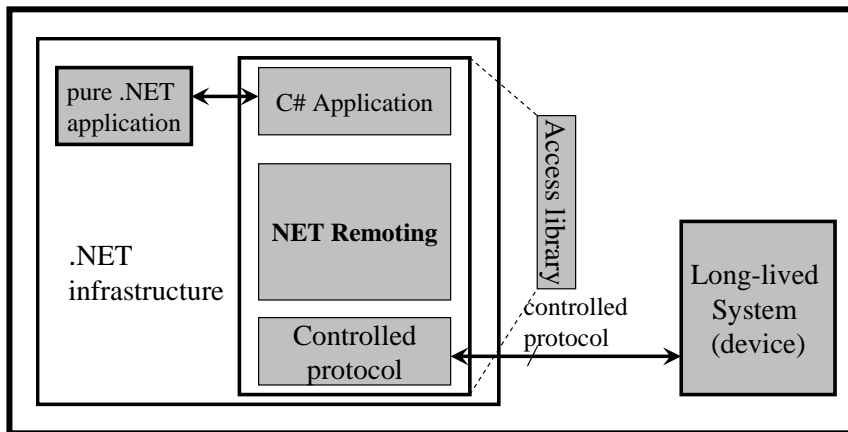


Fig. 3. NET Remoting for creating access library on .NET

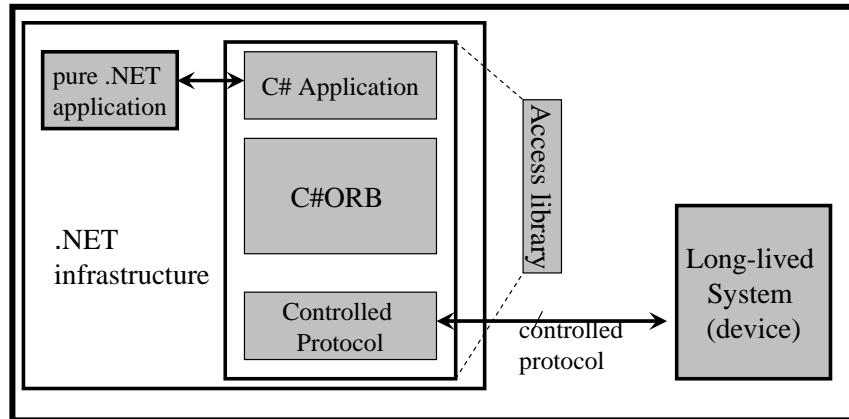


Fig. 4. C#ORB for creating access library on .NET

The fifth approach we consider is to support the controlled protocol directly on .NET using .NET Remoting [13]. As seen in Fig. 3, the controlled protocol has to be implemented underneath the Remoting framework.

The sixth approach consists of using a C#ORB to perform the remote procedure call. As seen in Fig. 4, this approach bypasses the Remoting framework and directly supports the controlled protocol.

These are described and evaluated in the following sections.

3 Evaluation of interoperability approaches

For our evaluation, we consider six criteria. These are

- **Type system support** - whether a rich variety of types are supported
- **Compact Framework support** - whether the feature is available on the compact framework platform
- **Callbacks** - whether callback and events can be easily programmed
- **Stability** – is there legacy which depends on this feature, thus making the feature stable in future releases of the .NET framework
- **Usability** - how easy is it for the developer to use the approach
- **Maintainability** – how easy is it to maintain code written using the approach

Support for the compact framework is needed if WindowsCE handheld devices are part of the solution, as they were in our case. For the case of callbacks, we evaluate whether an object reference can be passed from C# into the long-lived system, which the long-lived system can call back upon. For maintainability, we consider the amount of code that need to be written when a new interface is added.

To evaluate the different bridging techniques, we consider the use case of event handling. The .NET service implements an event handler which the device calls when events occur on the device. The interface for the event handler is:

```
interface EventHandler{
    void report (in Event evt);
};
```

The device has to implement an interface where event handlers could be registered. This becomes:

```
interface Device{
    void register(in EventHandler hndlr);
}
```

This use case demonstrates passing complex types (EventHandler and Event) objects through the bridge and passing object references (EventHandler) which the device uses to call back when events occur.

Based upon these criteria in a realistic use case described above , the various options are summarized in Table 1 with detailed discussion in the following sections.

Table 1 Evaluation of different approaches for C#.NET interoperability

Approach	P/Invoke	Managed C++	COM Interop	Localhost XML Web Services	.NET remoting	C# ORB
Type system	Limited	Good	Good	Fair	Good	Good
CF support	Limited	None	Poor	Fair	None	Good
Callbacks	Poor	Good	Fair	Poor	Fair	Good
Stability	Good	Good	Good	Good	Poor	Good
Useability	Poor	Limited	Fair	Good	Good	Good
Maintainability	Poor	Good	Fair	Good	Poor	Limited

3.1 P/Invoke

The P/Invoke mechanism works only for flat APIs and not for class libraries. P/Invoke works best when the call is made from the managed code into the unmanaged code. The reverse call and callbacks are not very easy to implement. For a par-

ticular flat API exposed from a Windows format dynamic link library (DLL), an equivalent function has to be specified in the managed language by the developer.

To use P/Invoke, the interface class has to be flattened and the implementation wrapped in a DLL declaration. For our example, after interface flattening, the method call becomes:

```
// flattened C++
void EventHandler_report(Event val) {..};
```

This is wrapped in a DLL declaration as

```
extern "C" void __declspec(dllexport) __stdcall EventHandler_report(Event val) {..}; // DLL C++
```

Once this method is implemented in a DLL `EventHandler.dll`, a C# class can call into this unmanaged code by defining an equivalent method using the `DllImport` attribute.

```
public class EventHandler{
    [DllImport("EventHandler.dll", EntryPoint= "EventHandler_report")]
    public static extern void report(Event val);
    ...
};
```

The unmanaged methods declared this way can be called from managed code as if the call is being made on a managed method.

- **Type system support:** Only a limited set of basic types are permitted for the return value of the methods. Most notably, string, float or double cannot be a return value if P/Invoke has to be used on an unmanaged method. All complex types have to be completely flattened into structures before converting into a DLL.
- **Compact framework support:** In the compact framework, P/Invoke can support only simple (32-bit or smaller) types inside a structure. Thus, strings, arrays, nested structures, nested objects, unions or reference types inside a structure are not supported. Function pointers, arrays of complex types or arrays of strings cannot be passed as arguments to methods. Apart from the limited type support, there is limited tool support for P/Invoke on the compact framework. For example, one cannot step into the unmanaged code while debugging P/Invoke methods. If a P/Invoke method fails, the resulting exception raised does not sufficient information to identify the cause of failure. On the compact framework, callbacks from unmanaged code into managed code are not supported. There are workarounds that use the windowing event subsystem to listen to call back events. However, these are very difficult to program.
- **Callbacks:** P/Invoke on the .NET full framework has support for callbacks by using C- style function pointers on the unmanaged side and delegates on the managed side. This approach has the problem that the developer has to manage the memory and layout of the pointers passed through the bridge.

- **Stability:** .NET applications depend on P/Invoke to get access to a large set of legacy Win32 DLLs. For this reason, P/Invoke is expected to remain stable, since the legacy Win32 DLLs would exist in the near future.
- **Usability:** For class libraries, flattening of the APIs are required. In cases where particular types are not supported (such as string inside a structure), memory pointers have to be used and marshalling has to be done in the application code. Thus, advanced uses of P/Invoke are error prone, unsafe, and difficult to automate.
- **Maintainability:** Code written using P/Invoke is not object-oriented. Currently, there are no automated tools to convert a generic class library into P/Invoke calls.

3.2 Managed C++

Managed C++ (MC++) is a Microsoft extension to C++. MC++ code can make bidirectional calls to C# without any extra work on the developer's part since MC++ is also managed by the .NET runtime. MC++ also allows managed and unmanaged code to be mixed in the same source file as well as in the same assembly, but the unmanaged classes have to be first recompiled into the Microsoft Intermediate Language (MSIL) code. To bridge into unmanaged C++ code, the developer has to create a MC++ wrapper around the unmanaged classes. For our example, a class called `MEventHandler` is created, which delegates calls to it to the unmanaged `EventHandler` class.

```
public __gc MEventHandler{
    void report(Event val){...}; // header file
};
// implementation
void MEventHandler::report(Event val){
    EventHandler* unmgcd = new EventHandler();
    iumgcd->report(val);
    delete unmgcd;
    return;
}
```

- **Type system support:** MC++ has rich type support. But, MC++ does not support multiple implementation inheritance, so C++ code that uses multiple inheritance has to be rewritten before compilation into MSIL. This was a problem with the RPC library where multiple inheritance was extensively used.
- **Compact framework support:** MC++ is not supported in the compact framework. This is due to the large footprint and resources necessary to support both managed and unmanaged code in the same assembly.
- **Callbacks:** MC++ allows bidirectional communication with unmanaged code as long the unmanaged libraries are recompiled into the MSIL Code. Without recompilation, MC++ has to use P/Invoke for callbacks and has the same limits as P/Invoke.
- **Stability:** .NET framework depends on MC++ to move legacy C++ code bases into the .NET framework, and thus is very likely to be stable in the near future.

The large install base of Visual C++ developers also require Managed C++ to develop for the CLR.

- **Usability:** Wrapping each of the types and methods into the corresponding managed types required for MC++ is error prone. Furthermore, the wrapped call also takes a performance penalty since one additional layer of function calls are needed for each call. In addition, C# has simpler syntax than MC++ and has compiler support to warn the developer if unverifiable code is being used (such as pointers).
- **Maintainability:** Since there is no clear way to support original C++ code that has multiple inheritance, tools which generate unmanaged C++ (such as the IDL to C++ compiler) have to be rewritten to generate managed C++ code.

3.3 COM Interop

COM Interop was developed to support legacy COM objects. This approach provides a richer object model than flat APIs. The objects from the C++ have to be converted into COM objects to use this technology. For this, the interfaces are first converted into Microsoft IDL (MIDL). Our example interfaces gets converted to

```
[
    uuid(2bcf8246-e42b-43b8-a936-91de09faf554)
]
interface IEventHandler:IUnknown
{
    HRESULT report([in] Event val);
}
[
    uuid(c445a87d-362c-404d-b099-92810f60d3d8)
]
coclass CEventHandler
{
    interface IEventHandler;
}
```

The MIDL file is compiled to obtain stubs and skeletons. The implementation along with the stubs is used to create a type library. The type library is converted into .NET callable libraries by using a tool called `tlbimp`.

- **Type system support:** COM Interop supports a rich variety of types and conversion of types from COM to .NET is automatic on the .NET full framework.
- **Compact framework support:** There is limited support for COM interop on compact framework. Tools for importing the type library and runtime support for COM are available from third party vendors [14].
- **Callbacks:** COM has support for callbacks, but it is difficult to program since intermediate wrappers have to be created.
- **Stability:** .NET depends on COM Interop to access legacy COM/ActiveX components. COM Interop is likely to be available, since COM components are likely to exist in the near future

- **Usability:** Class libraries that are not COM based require conversion to COM before the automatic import tools can be used. Knowledge of COM is required for performing conversions. In the compact framework, P/Invoke has to be used to simulate COM Interop.
- **Maintainability:** If the original class libraries are not COM based, there is an additional step to generating the MIDL file in addition to the steps for bridging between C# and COM.

3.4 Localhost XML Web Services

Localhost XML Web Services allows the managed and unmanaged code to call each other locally using XML over HTTP (using a Microsoft controlled version of SOAP). The instability of the XML Web Services is not of concern in this case as we are using XML Web Services only to call between the managed and unmanaged on the .NET platform. To create a Web Service on the unmanaged end, the class has to be wrapped as a COM object using techniques described in Section 4.1.3. A tool can then be used to extract the WSDL file [15] corresponding to the service from the COM object. For our example, the snippet of the WSDL file created for our example looks as below.

```
<message name='EventHandler.report'>
  <part name='val' type='Service:Event' />
</message>
<message name='EventHandler.reportResponse'>

  <portType name='EventHandlerSoapPort'>
    <operation name='report' parameterOrder='val'>
      <input message='wsdlns: EventHandler.report' />
      <output message='wsdlns: EventHan-
dler.reportResponse' />
    </operation>
  </portType>
```

On the managed side, this WSDL file is used by the tools to generate C# code.

- **Type system support:** XML Web Services has a rich type support and most of the types in IDL can be mapped to types in WSDL. However, WSDL does not support object references, an important construct in IDL.
- **Compact framework support:** XML Web Services client functionality is supported on the compact framework in both native as well as managed domain. Server functionality is supported only in the unmanaged domain.
- **Callbacks:** Since object references are not supported, callbacks are not possible.
- **Stability:** Web Services are an important component of the .NET framework and would have increased support in the future.
- **Usability:** There is a rich toolset to export and import WSDL descriptors. However, using Web Services incur a performance penalty as the message is transported over HTTP XML channels on the platform.

- **Maintainability:** Conversion of IDL to WSDL and then to C# source are automated and thus easy to support.

3.5 .NET Remoting

.NET Remoting is a technique used in the .NET framework to communicate between two distributed .NET applications. We cannot use the .NET remoting protocols directly for our application as this does not satisfy our requirement for a controlled protocol. But .NET Remoting allows adding a new protocol underneath it. This new protocol can be our chosen controlled protocol. .NET Remoting has the advantage that there is no need to bridge to unmanaged code; all code on the service platform is now in the managed domain. To simplify invoking the Remoting APIs, the IDL files can be converted automatically into C# interface classes. The implementation object inherits from the .NET class called MarshalByRefObject and the auto-generated interfaces.

```
public interface IEventHandler
{
    void report (Event val); // from IDL
}
public class EventHandler : MarshalByRefObject, IEventHandler
{
    ...
    public void report(Event val){..}; // implementation
}
```

Remoting allows customization of the formatters (which perform serialization and deserialization of types), transport channels (which transports the data to the remote process), proxies (so additional operations can be performed before the message is formed) and channel sinks (which can be chained to perform custom operations on serialized data such as encryption, compression etc.). Each of these layers could be modified to create the controlled protocol. Once the protocol is created, it can be used in the Remoting channel. The example shown is that of a server activated singleton object model [13] that implements our example interface description.

```
public class Server {
    public static void Main(string[] args){
        // creating the transport
        HP_Channel channel = new HP_Channel( 4467);
        // registering with runtime
        ChannelServices.RegisterChannel(chan);
        // create and publish implementation
        EventHandler svr = new EventHandler();
        RemotingServices.Marshal(svr, "EventHandlerServer");
        ....
    }
}
```

The client communicates with this object using :

```
public class Client{
    public static void Main (string[] args){
        // creating the transport no port specified
        HP_Channel channel = new HP_Channel();
        // registering with runtime
        ChannelServices.RegisterChannel(chan);
        // creating a local object
        IEventHandler rsvr = (IEventHandler) Activator.GetObject(typeof(IEventHandler), "hp_chan://15.25.40.44/EventHandlerServer");
        // call the method
        rsvr.report(val); ..};
```

In prior work, IIOP has been developed as an additional channel for .NET remoting [16].

- **Type system support:** Remoting can use all the types in .NET and most of these have mappings to IDL.
- **Compact framework support:** Remoting is not supported on the compact framework.
- **Callbacks:** Remoting has the notion of object references and thus callbacks and events are possible.
- **Stability:** Microsoft is moving towards a new architecture called Indigo [17] for the next generation .NET architecture. With Indigo, the lower level Remoting features such as channels and sinks (required for custom transport) will undergo major changes. Thus the custom channel would have to be entirely rewritten with Indigo. There is also no legacy reason for maintain customizable channels. Thus, the custom channel is likely to be unstable.
- **Usability:** Remoting is a better approach than MC++ or COM Interop since different object models or languages are not involved. All development is done in C# and with the .NET type system. However, the transport details and the asynchronous/synchronous call selection have to be done by the developer.
- **Maintainability:** Since Indigo would deprecate the protocol layers in the Remoting framework, it would become much more difficult to support it in the future.

3.6 C# Object Request Broker

C# ORB is our final approach to simplify the creation of the device access libraries. This approach is like .NET Remoting that it does not require bridging between managed and unmanaged code. However, unlike .NET Remoting, the C# ORB does not require any support from the .NET infrastructure other than the normal language and API support. Thus, the C#ORB can be also made compliant to the compact framework.

- **Type support:** All IDL types can be supported with an appropriate C# mapping for IDL.

- **Compact framework support:** Since no feature specific to the .NET full framework is used, all ORB functionality can be exploited in the compact framework
- **Callbacks:** Mapping of object references allow callbacks to be implemented.
- **Stability:** Since the C# ORB is an application level entity and uses only the .NET APIs, it is expected to remain fairly stable.
- **Usability:** The C#ORB does not require changing object models or languages. In our case, the details of the transport can also be hidden from the developer and discovered at run time by the infrastructure. All method calls into the access library are inter-library calls and thus does not incur any performance penalty.
- **Maintainability:** We can expect the C#ORB to remain stable. However, since this is not part of the .NET platform as the other methods, it has to be distributed along with the access library.

3.7 Summary of the evaluation

The usability of the approaches increases in the order we had evaluated them. P/Invoke requires flattening of the class library structure and thus loses the benefits of object orientation (such as polymorphism). COM Interop, MC++ and .NET Remoting are not available on the compact framework. XML Web Services incur a performance penalty since calls are made on an HTTP channel. Callbacks cannot be implemented easily with XML Web Services. None of these disadvantages is present with the C# ORB approach. Furthermore, this approach has the benefit that the stubs and skeletons are created automatically and the developer has a simpler programming model. There is no need for the developer to specify neither the attributes of the transport (as in Remoting) nor the mapping of different types on the wire (as in P/Invoke).

However, the C#ORB approach has the following disadvantages.

- The C#ORB require maintaining an additional infrastructure on the .NET platform
- There is a development cost associated with the development of a new infrastructure.

Since the ORB is an application level entity, we expect the maintenance cost to be same as that of maintaining any other application on the .NET platform. We expect this cost to be much less than the cost of maintaining the wire protocol compatibility in the device. With regard to the development cost, we found that to a large extent, existing tools could be used to simplify the tasks. Developing the ORB involves two distinct activities, one is the creation of a mapping for IDL in C# and the other is the development of the run time. Both these can be simplified using existing technologies and standards. We describe these below.

4. C#ORB development

We found that C# and Java are similar languages [18]. We leveraged this observation to use the IDL2Java mapping [19] as the reference for obtaining the C# mapping. Our mapping follows the spirit of the specification and addresses the most commonly used

constructs. However since the ORB is never exposed to the developer (developer only sees the application class library), we do not require strict adherence to the standard. This limitation should be removed if the ORB APIs are exposed to the developer or if a third party ORB has to be used. We also deviated from the standard to reduce the footprint and maintainability of the access library. The following are not supported in our mapping:

1. **Portability interfaces:** The portability interfaces were developed for the case when an application developed in one vendor's ORB can be downloaded into another vendor's ORB. In practice, we have found that when the application is loaded, the corresponding ORB is also typically loaded. Thus, portability interfaces are not necessary.
2. **Attributes and read only attributes:** Attributes and read only attributes can be implemented using the methods corresponding to the accessors and mutators. An attribute is replaced with get and set methods. A read only attribute is replaced with a get method.
3. **Pseudo interfaces, Pseudo Objects, Abstract interfaces:** We do not attempt to map these are used in practice only to specify interfaces used within the ORB libraries. We however specify a mapping for `TypeCode` and `Any`.
4. **Value types and Value box types:** We do not pass value types on the wire. Thus this was not necessary.

We also found that there are differences between C# and Java relevant to the mapping. We do not describe the detailed mapping here (as this can be gleaned from the Java mapping), but only the differences with the IDL2Java mapping. These are:

- **Type system differences:** Since C# has signed types, several of the basic IDL types can be more correctly mapped into C# than in Java. C# interfaces cannot contain fields. This creates a few mapping differences since the fields have to be mapped into separate classes.
- **Access modifier differences:** Java access modifier `protected` does not have a strict counterpart in C#. The closest mapping we could obtain is to map the Java `protected` keyword to `internal` `protected` in C#. The `protected` modifier in Java declares that the member is accessible to all subclasses of the class (regardless of the package where they reside) *and* to all classes in the package containing the class (regardless of any subclass relationship). In C#, the `internal` `protected` modifier declares that the member is accessible to other classes in the same assembly and all the subclasses of the class (regardless of the assembly they exist). However, the .NET assembly is typically a larger unit of compilation and serves different purposes than a Java package. The equivalent of a Java package is a C# namespace. However, C# does not provide a modifier that restricts access with a namespace. In addition, the default access for members of a class in Java is package-level, while for C# is `private`. For mapping members with the default access level in Java to C#, we use the modifier `internal`. Thus, our mapping gives wider access to the member than the Java mapping.
- **Keywords:** C# uses the `readonly` keyword to qualify constant variables determined at runtime and the `const` keyword to qualify constants using compile-time. The equivalent in Java is `final`, which denotes both runtime and compile-time constants. We map `final` to `readonly`. `value` is a keyword in C# used for

the property construct. Where this keyword occurs in the mapping, we replace it with `_value`.

- **Syntax differences:** C# does not allow static initialization blocks as in Java. These are mapped to the static constructor construct. In C#, instance class members with non-constant initialization expressions have to be initialized inside a method body. C# arrays are declared with the array brackets on the type name instead of the variable name, ie. `int [] foo`, compared to Java syntax of `int foo []`.
- **Inheritance:** C#'s inheritance mechanism is very different from Java. In Java, all inherited methods are implicitly polymorphic. In C#, the inherited methods that need to be polymorphic must be specified by the modifier `virtual` in the base class and by the modifier `override` in the derived class. However, if the method is an implementation of an interface or of an abstract method, use of the `override` modifier is illegal. If the method has to hide an inherited function with the same signature, the modifier `new` has to be used (the default modifier is `new`). In this case, the method is no longer polymorphic for instances of that particular class.
- **Call-by-value/result:** In C# the IDL `inout` and `out` semantics can be mapped without holder classes for the arguments of methods. C# has the keyword `ref`, which denotes that the variable being passed points to the same memory as the variable in the calling method. This implies that any change in the variable by the called function is reflected in the variable in the callee when the function returns. This can be used to map the call-by-value/result semantics of the IDL `inout` parameter. The keyword `out` has similar effect as the `ref` keyword, but `out` variables need not be initialized. This can be used to map the call-by-result semantics of the IDL `out` parameter. This mapping gives a more natural mapping for the developer that use of holder classes.
- **Property:** C# has the construct called “property” which is a smart field with a `get` and `set` method. The property is used just as a field; however, the compiler invokes the correct method based on whether the field is used as l-value or r-value. This construct reduces the number of accessor functions.

The C#ORB itself was developed by leveraging an existing Java ORB based on the ORBlite architecture [20] and using off-the-shelf code conversion tools [21]. We found that on an initial release, we could get the C#ORB to a memory footprint of 105 Kbytes. This enabled us to target it for compact framework platform.

5. Demonstration

The C#ORB solution was used for our application of communicating with a printer from a handheld device and a desktop computer. The printer did not natively support our chosen remote controlled protocol. However, the object model used in the device allowed us to install our controlled protocol as a downloadable module³. The

³ Note that this is a one-time install of the controlled protocol. In contrast, to support the evolving protocols, this has to be done each time the protocol changes.

download mechanism also provided us access to several interfaces exposed in IDL. The C#ORB was deployed on two configurations. In the first, it was deployed on a WindowsXP desktop. In the second configuration, we deployed the C# ORB on a WindowsCE platform, which supports the Compact Framework. Using our ORB and compiler, we were able to develop application code in C# in both configurations to control the printer. The same code base (access library including the C#ORB runtime) was used for both the .NET full framework and the compact framework. An example for the EventHandling service is shown below. The code shown below is complete. All remaining code is auto-generated. This shows the usability of the approach compared to the other approaches studied.

```
// Main thread of execution for .NET event handling
// service
public class Server
{
    public static void Main(System.String[] argv)
    {
        org.omg.corba.ORB.init();
        //Get a reference to the Device
        ObjectRef obj =theRegistry.find("MyDevice");
        DeviceRef dev = DeviceStub.narrow(obj);
        // Create the EventHandler
        EventHandlerImpl serv = new EventHandlerImpl();
        obj = EventHandlerSkeleton.createRef(serv);
        //Start the EventHandler
        SOA.run()
        //Register the eventHandler with the device
        dev.register(obj);
    }
}
public class EventHandlerImpl:EventHandlerServant
{
    public void report(Event val)
    {
        //Store it in a database
    }
}
```

As seen from the example, there is no need to specify the transport channel in the application code. The C#ORB is able to add new protocols exploiting the reflection mechanism and static constructors available in .NET.

Using the infrastructure, we were able to get into the Event Subsystem and the Front Panel objects of the printer (both these required use of object references and callbacks similar to the example we considered). This gave us notifications on job complete, paper jams, out of paper and other management related events.

Installing libraries on clients is similar to existing approaches for installing a device driver on desktops for a new hardware device, except we are installing “drivers” at the application level and are auto-generating part of the driver. We have turned an

interoperability problem into an installation problem; but gratefully this problem is well understood.

When device interfaces change, the drivers have to be updated and distributed. Thus, strong versioning mechanisms have to be in place for reducing incompatibilities between the driver and the device. Another issue is that there has to be an access library on each platform the long-lived system vendor wants to support (e.g. IBM WebSphere and BEA WebLogic in addition to .NET). Using a tool chain to automate code generation is expected to mitigate these problems. A limitation of our approach is that as the platforms evolve, the device vendor would have to maintain the old “drivers” in addition to the new ones. However, we have found that programming languages and application level APIs change much slower than wire protocols. Furthermore, this migration is in the control of the device vendor.

6. Related Work

There have been other efforts to develop a C#ORB such as Borland’s Janeva [22] and Middtech’s C#ORB [23] products. Both these products have tools that could be used for application bridging; however, it would require support for their protocols in the long-lived device, including which version of which protocol. Presently, Janeva does not support the compact framework that is necessary for integration with handheld devices.

.NET Remoting with an IIOP channel has been used to support integration of CORBA components with .NET [16]. This has the advantage of using .NET tools and CLR directly; however, it suffers from the longevity of the exposed .NET Remoting extensibility API.

Several CORBA vendors support interoperability with .NET. One solution [24] uses the .NET Remoting channel and managed C++ to provide a bridge. The focus is on hiding the details of CORBA for the .NET developer. However, this solution has the same disadvantages of .NET Remoting. JNBridge [25] is an automated tool chain to link Java applications to .NET application when both run on the same platform. A Java ORB and the JNBridge could be used to bridge into C#. However, the C#ORB does not require moving between multiple languages.

The approach of installing libraries on clients is similar to existing approaches for installing a device driver on desktops for a new hardware device, except we are installing “drivers” at the application level and auto-generating the libraries. In contrast to device drivers, being at the application level has advantage that a single “driver” could provide access to multiple devices.

7. Conclusions

This paper presents an approach on how to sustain the integration of long-lived systems with C#.NET. To maintain stability and interoperate when wire protocols evolve, a device access library is hosted on a .NET platform. This allows C#.NET

applications that access the device to be developed, transparent to the underlying controllable protocol used to communicate with the device, at the expense of calling a set of device APIs. To support a wide variety of languages, we use IDL specifications for exposed device interfaces and have an IDL compiler tool to auto-generate the stub/skeleton glue code.

For the access library exposed on a .NET platform, we evaluated existing approaches, and the approach using a C# ORB was found to be best for our solution. We developed a low footprint C# ORB and a mapping of IDL to C#. Our single code base works on both the compact framework as well as the full framework. We found that a large part of an existing Java code base can be leveraged to produce the C# ORB and the IDL language mappings for Java and C# are quite similar. C# however has additional constructs and types which gives a better mapping for IDL than Java.

Our main finding in this work is that the approach for obtaining and sustaining interoperability using common wire protocols has limitations when integrating with long-lived systems. We find that interoperability can be achieved more naturally and for greater duration by bridging at the application layer than at the protocol level. For the case of .NET , a C#ORB deployed on the device has advantages over other approaches for interoperating with long-lived devices.

References

1. Resources and Guidelines for Web Service Interoperability, <http://www.ws-i.org>
2. Web Service Profiles – An Introduction, http://www.ws-i.org/docs/WS-I_Profiles.pdf
3. IBM WebSphere software platform, <http://www.ibm.com/websphere>
4. BEA Weblogic Server, <http://www.bea.com/products/weblogic/server/index.shtml>
5. Microsoft .NET, <http://www.microsoft.com/net/>
6. CORBA/IIOP specification version 3.0.2, Chapter 15, December 2002, <http://www.omg.org/docs/formal/02-12-02.pdf>
7. SOAP specifications, <http://www.w3.org/TR/soap/>
8. Interface Definition Language, ISO/IEC Standard No. 14750, 1999.
9. Clark.J , “ Calling Win32 DLLs in C# with P/Invoke”, MSDN Magazine, July 2003.
10. Grimes.R, “Programming with Managed Extensions for Microsoft Visual C++.NET”, Microsoft Press, 1st Edition, July, 2003.
11. Troelson.A, “COM and .NET Interoperability”, APress, 1st Edition , April 2002.
12. Freeman.A and Jones.A, “Microsoft .NET XML Web Services Step by Step”, Microsoft Press, 1st Edition, October 2002.
13. Rammer.I, “Advanced .NET Remoting”, APress, 1st Edition, April 2002.
14. Odyssey Software, “CFCOM”, <http://www.odysseysoftware.com/cfcom_main.html>
15. WSDL specification, <http://www.w3.org/TR/wsdl>
16. Oberleitner.J and Gschwind.T, ”Transparent Integration of CORBA and the .NET Framework”, On the Move to Meaningful Internet Systems, Lecture Notes in Computer Science, Springer Verlag, 2003.
17. Box.D, “A Guide to Developing and Running Connected Systems with Indigo”, MSDN Magazine, Jan 2004.
18. Obasanjo.D, “Comparison of Microsoft’s C# programming language to Sun Microsystems’ Java programming language” <http://www.25hoursaday.com/CsharpVsJava.html>
19. Java2IDL mapping, <http://www.omg.org/docs/formal/02-08-05.pdf>
20. Moore.K and Kirshenbaum.E, “Building Evolvable Systems- the ORBlite project”, HP Laboratories Journal, February 1997. <http://www.kirshenbaum.net/evan/publications/orblite-hpj.pdf>

21. Microsoft Java Language Conversion Assistant 2.0, <<http://msdn.microsoft.com>
keyword: "JLCA">
22. Borland Janeva, <http://www.borland.com/janeva/>
23. Middtech Corporation, <http://www.middtec.com/index.html>
24. IONA Corporation <http://www.iona.com/products/orbix.htm>
25. JNIBridge <http://www.jnbridge.com/>