A Generic Serializer for Mobile Devices

Mattia Monga
Università degli Studi di Milano
Dip. di Informatica e Comunicazione
Via Comelico 39/41, 20135 Milan, Italy
mattia.monga@unimi.it

Angelo Scotto
Politecnico di Milano
Dip. di Elettronica e Informazione
Via Ponzio 34/5, 20133 Milan, Italy
scotto@elet.polimi.it

ABSTRACT

In this paper we describe a serializer component completely realized in .NET managed code, able to run on a stripped versions of the .NET platform (e.g., COMPACT FRAMEWORK) and still generic enough to be used on .NET or other CLI compatible frameworks. Such a component is not normally provided with stock libraries in their compact version, since its implementation is quite tricky when relying on reduced reflection services. However, this component makes easier the development of distributed applications involving mobile devices and desktop computers or mainframes. Our implementation faced several problems ranging from lack of features in the base classes to the inter-framework portability problem, since the same object could have different implementation. The resulting product shows satisfactory performance figures and has a modular and flexible architecture.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Tools and Techniques—*Software libraries*

Keywords

Serialization, mobile devices, .NET

1. INTRODUCTION

Nowadays, the popularity of wireless-enabled PDAs and smart phones is increasing. As a result, self-organized ad-hoc networks of PDAs, laptops, and smart mobile phones are being proposed as an enabling technology for collaborative work during meetings and conferences; they also promise to be a great help in extreme scenarios as disaster relief, or battlefield environments. However, these networks of computers are characterized by the great heterogeneity of the hardware involved and it would be unfeasible to build from scratch lots of different applications as many as the different devices that are on the market today. Therefore, it is common to program applications on top of virtual machines, like the ones provided by the Java [9] and the .NET [6] platforms.

Virtual machines, together with a set of core libraries, provide a common platform where designers can write their applications abstracting from hardware details. In principle, the applications can be run on every device which provides an implementation of the virtual machine. Many of us remember a slogan forged some years ago to boost Java: "Write once, run everywhere". Unfortunately, notwithstanding the marketing hype, we are quite far from this goal as far as networking application are concerned.

A first problem is that, while the virtual machines are being ported on several devices, the core libraries are not. More precisely, memory and resource limitations often constraint the availability of the entire set of core libraries. Thus, when programmers write applications for PDAs or mobile phones they must rely on a reduced set of core libraries, like the JAVA 2 MICRO EDITION [7] or the COMPACT FRAMEWORK [2].

Moreover, networking applications can be difficult to code in these compact environments, since they lack standard middleware layers, i.e., software that mediates between an application program and the network. In fact, complex middleware platforms such as CORBA [3] are not well suited for networks of devices with limited resources and both Java and .NET do not provide remote invocation support in their compact editions. Nevertheless, currently available light frameworks support the use of mobile devices as clients in distributed applications. The MS COMPACT FRAMEWORK, for example, supports interaction with web-services (via `System.Web.Services` classes) and database engines (via `System.Data` classes). However, no support is provided by the libraries outside these strictly client-server architectures: a simple synchronization application between a mobile device and a desktop computer or an instant-messaging application among PDAs would require implementing them directly at the TCP level.

We focussed on a frequent operation used in object-oriented distributed applications, known as *serialization*. A memory object is converted in a stream of data, suitable to be transferred by the network transport layer. The object is eventually rebuilt (*de-serialized*) in the environment of the destination machine. The desktop version of .NET provides two automatic class serializers, namely the BINARYFORMATTER and the SOAPFORMATTER. They would make easier the development of distributed applications on mobile devices but unfortunately no serializer has been ported to the

.NET Compact Framework[1]. In fact, the specification of the CLI Compact Profile[2] does not include serialization facilities and just a few implementations exist. As a matter of fact, an implementation of such a service should deal with a number of trade-offs among efficiency, expressiveness, genericity, and portability across different environments.

In this paper we will describe CompactFormatter[3], our implementation of a generic serializer targeting the .NET Compact Framework, but able to work on every standard CLI implementation [10], such as the desktop version of .NET, Mono [11] and Portable.NET [5]. In fact, CompactFormatter may be used as a replacement of the BinaryFormatter every time the latter is not available or is unsuitable. The paper is organized as follows: in Section 2 we summarize motivations and challenges in implementing a generic serializer, in Section 3 we describe our implementation strategy, in Section 4 we compare CompactFormatter to other available serializers, and finally in Section 5 we draw some conclusions.

2. MOTIVATIONS

Initially developed as part of the peer-to-peer middleware SmartPeerware[4] targetting mobile devices, CompactFormatter has been finally released as an autonomous component.

The design goals that drove its development were:

1. maximum portability, by using only managed code;
2. good enough performance even considering the burden of managed code;
3. easy transition from BinaryFormatter code, by using a programming interface very similar to BinaryFormatter and SoapFormatter's one;
4. modular architecture to enable flexibility and customization.

In fact, a clone of the BinaryFormatter, would not be enough, since it makes massive use of native code during serialization and de-serialization phase [8]. Unfortunately, this approach does not fit well with our scenario characterized by huge heterogeneity of hardware components. Therefore, portability being our main concern, we decided to implement CompactFormatter completely in managed code. Our performance measures indicate (see Section 4) that the burden introduced by managed code does not affect too much performances: in many cases we actually perform better than BinaryFormatter.

However, implementing an efficient while generic serializer entirely in managed code, poses a number of challenges we will describe in the following subsections.

[1]In this paper we focus on .NET Compact Framework, but similar considerations apply also to J2ME, the implementation of the Java platform for mobile devices, since it lacks the serializer it has in the desktop version

[2]The CLI standard specifies a profile as a set of libraries, grouped together to form a consistent whole that provides a fixed level of functionality. The Compact one is aimed at devices with only modest amounts of physical memory.

[3]The latest version of CompactFormatter (1.0 GeNova) is available from `http://www.freeweb.com/compactFormatter` together with full source code and documentation, released under LGPL license.

2.1 Reduced Reflection Services

The first problem one has to face when implementing a serializer by using just the services provided by light frameworks like the CompactFormatter, is the lack of a complete reflection API. In fact, CompactFormatter's reflection classes are highly stripped down and several methods critical for serialization do not exist. As a result the deserialization phase is seriously affected. In principle, this phase consists in: (1) retrieving type information about the serialized data, (2) creating an instance of the relevant type, (3) populating the newly created object with the serialized status.

A way to obtain an instance of the serialized type is by calling one of the constructors. However, the semantics of constructors is unknown to the serializer or these constructors might even be unaccessible if the type being deserialized implements some form of the Factory Pattern. Though, serialization does not need full initialized instance, since all inner fields of the instance will be overwritten with data received from the stream. What a serializer really needs is a static method which takes as a parameter the type of the object to instantiate and returns an uninitialized instance of the object. This solution is indeed adopted by the full version of the .NET framework and used by BinaryFormatter: it calls a static method `GetUninitializedObject(System.Type)` defined in the namespace `System.Runtime.Serialization.FormatterServices`. Unfortunately, this method has been removed from the Compact Framework.

2.2 Inter-framework Serialization

Another problem we faced was due to implementations of class libraries among different frameworks. In fact, the ECMA standard [10] describes the common interface, while inner details are left to implementors. However, private fields capture the object state and therefore they must be serialized. Moreover, during de-serialization a coherent object state has to be re-created: imagine implementation A has implemented class `C` with a private field `int x`, that is not available in B's implementation of the same class. What is the fate of `x` when an object of class `C` is serialized from A to B? One cannot use or even discard it, without knowing the exact semantics of `x` in A's implementation. We believe that this problem has a general solution only when a known relation exists between two frameworks. For example, if the Compact Framework implementation were a strict subset of the full .NET, one could try to find a workaround solution. However, we aimed at the general case of inter-framework portability, and as one can easily guess, two independent implementation as, for instance, .NET and Mono, have completely different private members, excluding any easy mapping.

3. SERIALIZER IMPLEMENTATION

CompactFormatter serializes objects by using a simple and straightforward algorithm. As far as primitive types (such as `int`, `bool`, `String`, etc) are concerned no further complications are needed: CompactFormatter simply writes a byte (a header) representing the primitive type being serialized and then writes a binary representation of the primitive type. Automatic serialization is obtained by exploiting reflection services: since also private fields are affected, serializable classes should grant all `ReflectionPermissions`. A non-primitive object can be automatically serialized only if both the following conditions hold: (1) it is marked by the programmer with the `Serializable` attribute[4], (2) a parameterless constructor is available.

Object serialization is preceded by serialization of its type. When multiple instances are to be serialized, type serialization is performed only once, by exploiting a type cache. Since CompactFormatter serializes, in a recursive manner, all the inner values of an object, this may cause endless loops when serializing self-referencing or cross-referencing objects. To avoid a duplicated object serialization while guaranteeing correct de-serialization, all serialized references are listed in an *object-table*; when an instance is repeated, only its index in the object-table is serialized. If a field should not be serialized (e.g., for security considerations), it can be marked with the `NotSerialized` attribute. These fields are ignored by CompactFormatter during the serialization and de-serialization phase.

For maximum flexibility, users may replace the serialization algorithm. A class may implement its own serialization algorithm by implementing the `ICFormatter` interface. In this case the class must be marked with the attribute `[Serializable(Custom=true)]`.

An even more flexible way to implement your own serialization algorithm is to define an *overrider* class, to which delegate the serialization work. The overrider class has to be registered with the CompactFormatter, and it is called by `Serialize` and `Deserialize` methods. This approach is suitable also when the class to be serialized cannot be modified to adapt it to CompactFormatter.

If a class has no parameterless constructor and modifying it is unfeasible, CompactFormatter provides the *surrogate* mechanism. A surrogate is a method with the same signature of the *GetUninitializedObject* provided by the full .NET framework (see Section 2.1). Surrogates should contain the code necessary to create an instance of the type to be serialized (typically by calling a constructor with predefined parameters) and they must be registered with the `AddSurrogate` method. A complete set of surrogates and overriders for CompactFormatter classes is being developed in order to allow users to safely serialize all the standard classes without bothering with their details.

In order to guarantee inter-framework portability (see Section 2.2), CompactFormatter provides a `PORTABLE` mode. In this mode CompactFormatter put in the serialized stream the name and the value of each serialized field (instead of sending just the value). This increase the size of serialization stream and therefore `PORTABLE` mode is off by default. If this is not enough, one can always implement an ad-hoc serialization with custom serialization or the overrider mechanism. For example, an overrider for a `Hashtable` (a class whose implementation is different in Compact Framework and full .NET) could simply extract the array of keys and the values array and serialize them instead of the complete set of the inner `Hashtable` fields, thus avoiding any incompatibility.

4. PERFORMANCE EVALUATION

Since CompactFormatter is entirely written in managed code, we were particularly concerned about its performances. Space (the size of the serialized stream) and time efficiency are very important on mobile devices where bandwidth is scarce and CPUs are normally not very powerful.

4.1 Experimental Setup

We compared CompactFormatter and BinaryFormatter. Since BinaryFormatter is not present in the Compact Framework, all the tests were done with the full version of .NET Framework (v1.1) and on the Mono Framework (v1.0).

Stream size was measured by serializing objects to the file-system. Measuring time instead was more difficult, even if profilers can be used to filter non-deterministic factors such as the load of the machine at the time of measurement or the garbage collector from measurements. In order to minimize this problem we repeated each of the following test 5 times and averaged the resulting data. The mean averages are tabulated in Tables 1 and 2 .

Performances were calculated on primitive types (`string`, `int` and `System.DateTime`[5]), on a composite object type (`SimpleObject`) containing just primitive types, and on `System.Data.DataSet`[6], a data type frequently used in web services implementations.

It is worth notice that figures related to `System.Data.DataSet` objects are actually referred to the overriders we implemented to serialize them, since the class is not `CompactFormatter.Serializable` and we could not modify it. In other words, when using an overrider, we measure performances of overrider and not of the CompactFormatter.

Our performance analysis used an Intel Pentium IV 2.8 GHz computer with 1 GB RAM and Windows XP Professional with SP1 as the hosting operating system. Timing was measured with the AQTime Profiler .NET Edition (v1.0.1.0) by AutomatedQA, and the built-in profiler (`--profile`) available in the `mono` runtime.

4.2 Analysis

The figures we found were very encouraging. When we designed CompactFormatter we hoped to be able to build a library whose performance were comparable to the BinaryFormatter's. Indeed CompactFormatter serializes objects very efficiently, so efficiently that it could be used (and in fact is currently used[1]) on the full framework as a replacement of standard BinaryFormatter when efficiency is critical. Implementation devoted particular attention to primitive type serialization, since every class eventually boils down to a composition of primitive types. In fact, the performance gain is particularly evident when looking at primitive types and less significant when moving to non-primitive objects.

As far as concern space use the CompactFormatter is consistently more efficient than the BinaryFormatter and for both formatters size grows linearly with the number of serialized objects.

The tests done with the Mono Framework confirm the results found with Microsoft .NET. When serialization time is separated from de-serialization time, our analysis shows that while both Microsoft BinaryFormatter and CompactFormatter spend more time during serialization phase,

[4]Attributes are annotations that can be queried at run-time. In the .NET framework they can be associated to classes, methods, properties, assemblies.

[5]so popular that CompactFormatter treats it as a primitive type, transforming it in a `Int64`

[6]It represents an in-memory cache of data, usually taken from a database

System.Int32				
instances	BF time (ms)	CF time (ms)	BF size (bytes)	CF size (bytes)
1000	153,15645	23,24959	54000	7000
500	115,58045	19,29751	27000	3500
100	104,06823	15,29723	5400	700
System.String				
1000	84,94456	29,50241	26890	12780
500	70,55079	21,5633	13390	6280
100	54,86126	18,88777	2590	1080
System.DateTime				
1000	158,4888	25,08933	59000	11000
500	122,30187	18,46967	29500	5500
100	94,66547	15,22662	5900	1100
SimpleObject				
1000	219,46724	166,62544	233000	69245
500	144,75773	100,51498	116500	34745
100	105,98202	47,56664	23300	7145
System.Data.DataSet (instances reduced, due to their size)				
10	329,38096	167,34773	106140	49741
5	271,08221	124,06514	53070	25061
1	214,98476	85,35767	10614	5317

Table 1: Performance comparison between CompactFormatter and BinaryFormatter (.NET Framework v1.1)

System.Int32				
instances	BF time (ms)	CF time (ms)	BF size (bytes)	CF size (bytes)
1000	2443,099	236,906	54000	7000
500	1303,857	130,595	27000	3500
100	350,95	45,825	5400	700
System.String				
1000	1309,438	331,812	26890	12780
500	660,718	194,813	13390	6280
100	146,775	53,148	2590	1080
System.DateTime				
1000	2867,52	290,283	59000	11000
500	1397,224	123,842	29500	5500
100	332,416	45,35	5900	1100
SimpleObject				
1000	7571,285	4121,948	233000	69245
500	3827,449	2217,121	116500	34745
100	872,3393	480,993	23300	7145
System.Data.DataSet (instances reduced, due to their size)				
10	5477,06	4404,602	106140	49741
5	3081,701	2425,781	53070	25061
1	1077,993	786,458	10614	5317

Table 2: Performance comparison between CompactFormatter and BinaryFormatter (Mono Framework v1.0)

Mono BinaryFormatter spends a longer time during its de-serialization phase.

Moreover, the performance figures we found support the conclusion drawn by [14] that Mono has still to tune its optimizations in order to compete with Microsoft .NET.

4.3 Comparison with other Light Framework Serializers

As already noticed the lack of serializers on the Compact Framework makes problematic the development on mobile devices. Therefore it does not come as a surprise that there are several third party components which tries to bring serialization on the Compact Framework.

4.3.1 Compact Framework *v2.0*

In the upcoming 2.0 version of the Compact Framework– still not *Released to Manufacturing* (RTM) at the time of writing– Microsoft decided to add a `XmlSerializer` class; this class, already present in the full desktop framework is useful to transform objects to XML and viceversa. Differently from the SoapFormatter, `XmlSerializer` poses several constraints to classes that can be serialized: (1) a class must have a parameterless constructor, (2) only public properties and fields are serialized, (3) there is no guarantee that an object is de-serialized using the same type (type identity and assembly informations are not transferred), it is up to the final user to instruct a `XmlSerializer` about the type it should serialize/de-serialize. CompactFormatter uses a custom binary format to serialize objects instead of XML and therefore it is more efficient (both in time and space used, since less transformations are involved to obtain a binary representation instead of a XML one). Moreover, it uses reflection to serialize and de-serialize objects correctly identifying types being serialized (together with assembly information) and serializing also inner private fields.

4.3.2 Pickle *Framework*

Written by Dominic Cooney, the Pickle Framework[13] is able to serialize primitive types and objects implementing the interface `IPickleable`. `IPickleable` classes must have a parameterless constructor in order to be serialized and there is no support to serialize instances of Compact Framework classes (the user has to be aware of potential problems when writing `IPickleable` classes).

While the Pickle framework is quite similar to CompactFormatter in its design goals, overriders and surrogates make CompactFormatter much more flexible. Moreover, Pickle uses its own custom interface to do serialization resembling Java habits, instead CompactFormatter tries to emulate the BinaryFormatter– standard– interface and should be found more natural by .NET programmers.

4.3.3 OpenNETCF `XmlSerializer`

Part of the Smart Device Framework[12] OpenNETCF has been developed to substitute the `XmlSerializer` class on the Compact Framework v1.0 (where it was not present). Unfortunately OpenNETCF's `XmlSerializer` suffers the same problems of Framework version (see Section 4.3.1) but it has even more tightening requirements because it is able to serialize only public properties (instead of all public fields as in .NET `XmlSerializer`) and every serialized property must be marked with an attribute mapping the property to the XML tag that should be used in the output (.NET `XmlSerializer` is able to rebuild this information using reflection.)

5. CONCLUSIONS

We designed CompactFormatter in order to have a generic serializer, with acceptable performances, usable during the development of mobile device applications and portable across different framework implementations. The results exceeded our best expectations, since thanks to its efficiency it is spreading also on desktop computers as a replacement for the standard BinaryFormatter.

Writing a generic serializer without any support from the hosting framework poses several problems, that we tackle with a modular and extensible architecture based on the overrider pattern and surrogates. We plan to increase the number of available surrogates and overriders to ease the work of application programmers. Our approach can be easily applied to Java J2ME framework. We are currently working also on Java and CLI interoperability.

An alternative solution we would like to investigate in the future is based on the `Emit` API (not available on the Compact Framework). This library allows byte-code manipulation and it makes possible to build tools that modify binary assemblies to make them "CompactFormatter-aware" by adding parameterless constructors and implementing suitable interfaces.

6. REFERENCES

[1] P. Bromberg. True binary serialization and compression of datasets. *EggHeadCafe.com*, 2004.

[2] .NET Compact Framework. `http://msdn.microsoft.com/mobility/netcf`.

[3] Common Object Request Broker Architecture. `http://www.corba.org`.

[4] G. Cugola and G. Picco. Peerware: Core middleware support for peer-to-peer and mobile systems, 2001.

[5] Portable.NET. `http://www.dotgnu.org/`.

[6] Microsoft^{tm} .NET. `http://www.microsoft.com/net/`.

[7] Java 2 Micro Edition. `http://java.sun.com/j2me/`.

[8] P. D. Jong. The History, Architecture, and Implementation of the CLR Serialization and Formatter classes. Talk at 2*nd* Rotor Workshop at Pisa, Italy, Apr. 2003.

[9] T. Lindholm and F. Yellin. *The Java^{tm} Virtual Machine Specification*. The Java^{tm} Series. Addison Wesley Longman, Inc., second edition, Apr. 1999.

[10] Microsoft Corporation. Standard ECMA-335. Technical report, ECMA International, 2002.

[11] Mono Framework. `http://www.mono-project.com`.

[12] Smart Device Framework. `http://www.opennetcf.org`.

[13] Pickle Framework. `http://www.dcooney.com/wiki/default.asp?PickleFramework`.

[14] W. Vogels. HPC.NET — are CLI-based virtual machines suitable for high performance computing? In *SC2003: Igniting Innovation. November 15–21, 2003*, Phoenix, AZ, Nov. 2003. ACM, ACM.