CompactFormatter User Manual

Angelo Scotto

February 27, 2005

Contents

1 Introduction

CompactFormatter is a generic formatter for the .NET Compact Framework. It is fully realized in .NET managed code and is able to work on every device supporting the .NET Compact Framework or another standard-compliant CLI implementation (such as the Mono framework).

This version of CompactFormatter (GeNova) is a complete rewrite of previous version (Monga) and tries to solve all the problems met with previous implementation.

In fact previous version of CompactFormatter always tried to serialize Framework classes using its internal serialization algorithm, this algorithm worked for a lot of classes but failed on several others and CompactFormatter was not flexible enough to allow custom serialization of these Framework classes without a complex and cumbersome process. The new version of CompactFormatter has several ways [1] to serialize classes not automatically serializable and can work in two main modes:

UNSAFE : Identical to previous version of CompactFormatter, it will try always to serialize a Framework class even if it's not sure to be able to serialize it (After all Monga worked with a lot of framework classes without problems).

SAFE : The default mode for GeNova; CompactFormatter will serialize Framework classes *only* if it knows how to serialize it (through a previously registered Surrogate or Overrider).

1.1 Features

- Able to serialize primitive types automatically.
- Able to serialize user-defined types automatically.
- Thanks to Surrogate and Overrider mechanism, allows a safer serialization even of framework types.
- Uses an interface very similar to standard Binary/SOAP formatters used in .NET
- Supports `NotSerializable` fields, that are fields which **must not** be serialized.
- Produces a data stream more compact than the one used by .NET BinaryFormatter.
- It is faster than BinaryFormatter and SOAPFormatter.

[1] Surrogate and Overriders, more on these later.

- Allows users to redefine serialization mechanism of their classes (Custom Serialization).
- It correctly rebuilds the object reference graph.
- Allows the use of user-defined **StreamParsers**, these are wrapper around serialization stream (useful to implement transformations just before sending the stream on the wire: soon available StreamParsers will allow zip compression and TEA data encryption out of the box.)
- Fully implemented in .NET managed code.

1.2 License

CompactFormatter is open source and released under GNU Lesser General Public License terms and conditions, which are reported fully at `http://www.gnu.org/copyleft/lesser.html`.

Name	Framework Type	Size (bytes)
bool	System.Boolean	1
byte	System.Byte	1
char	System.Char	1
DateTime	System.DateTime	8
Decimal	System.Decimal	16
Double	System.Double	8
short	System.Int16	2
int	System.Int32	4
long	System.Int64	8
signed byte	System.SByte	1
Single	System.Single	4
String	System.String	variant
unsigned short	System.UInt16	2
unsigned int	System.UInt32	4
unsigned long	System.UInt164	8

Table 1: Primitive types for CompactFormatter

2 Compact Formatter usage

2.1 CompactFormatter modes

The GeNova version of CompactFormatter may work in several modes, based on two flags called PORTABLE and SURROGATE.

PORTABLE When serializing across different frameworks a problem which could arise is due to different implementation of the same object (for example Hashtable implementation on the .NET Framework has different fields respect to the same Hashtable class on the Compact Framework).

Usually CompactFormatter uses an algorithm which guarantees a compact stream of data but is prone to failure when serializing classes between frameworks with different implementation. With PORTABLE mode enabled, CompactFormatter uses a bigger serialization stream which should prevent the majority of cross-framework problems[2].

SURROGATE Differently from previous versions of CompactFormatter, GeNova doesn't serialize a class if it's not sure of being able to serialize it. Considering that primitive types (see table 2.1 for a list of primitive types) are always serializable and that user-defined objects

[2]Unfortunately not all classes are serializable even using PORTABLE mode; when PORTABLE mode is not enough the only solution consists in implementing an overrider(see paragraph 3.1) for the target class

are usually marked with Serializable attribute, and therefore serializable, the only classes which could create problems are Framework classes. With SURROGATE mode enabled CompactFormatter "plays safe", that is it doesn't serialize any object not marked with serializable attribute if it doesn't have a surrogate or an overrider registered. This means that *every* Framework class which will be serialized by CompactFormatter in SURROGATE mode *must* have a surrogate (or an overrider) registered: this is usually an acceptable constraint in the majority of situations anyway, CompactFormatter user may deactivate this mode; in that case CompactFormatter GeNova will work exactly as Monga: it will try to serialize classes even if they're not marked with Serializable attribute.[3]

SAFE Safe mode is simply a combination of SURROGATE and PORTABLE modes and therefore is the safest way in which CompactFormatter may operate.

NONE this mode is obtained simply deactivating both SURROGATE and PORTABLE flags. It's the most *unsafe* mode in which CompactFormatter can operate but it's also the most efficient, no memory is wasted to keep registered overriders or surrogates and the serialization stream doesn't contain any redundant information (necessary in PORTABLE mode).

EXACTASSEMBLY Usually turned off, this flag drives CompactFormatter policy used when deserializing a class. When in EXACTASSEMBLY mode, CompactFormatter deserializes an object only if it has available locally an assembly with the *exact* Fully Qualified Name of the object serialized (This means that both assemblies has the same version, locale culture data and signature); Obviously this is a strict request and is unreasonable when serializing/deserializing between different frameworks (since they have different class implementations their assemblies will obviously have different signatures) and also when testing an application (each rebuild will change the version number of the assembly therefore objects serialized with different builds of the same code won't be compatible).

When not in EXACTASSEMBLY mode CompactFormatter will check only the Assembly name (without considering other informations) and therefore all problems depicted above will be removed.

By default CompactFormatter works in SURROGATE mode, this because advantages given by PORTABLE mode are restricted to very specific

[3]This obviously can bring to the same problem of old CompactFormatter therefore it should be used only when strictly necessary.

situations while the increase in serialization stream size is paid for each object serialized; on the other hand advantages given by SURROGATE mode are huge in terms of stability and robustness of the whole formatter.

Mode is assigned to CompactFormatter using the constructor `CompactFormatter(CFormatterMode)`, the parameterless constructor will instead use the default SURROGATE mode.

2.2 Usage

CompactFormatter usage is nearly identical to standard .NET Formatters[4].

First of all user must create an instance of the CompactFormatter using one of two available constructors (parameterless or mode)

```
...
CompactFormatter CF = new CompactFormatter(CFormatterMode.SAFE);
...
```

At this point we have the first difference with BinaryFormatter or SOAPFormatter because each Overrider or Surrogate must be registered through calls to methods `AddSurrogate` or `AddOverrider`

```
...
CompactFormatter CF = new CompactFormatter(CFormatterMode.SAFE);
CF.AddOverrider(typeof(OverriderClass));
CF.AddSurrogate(typeof(DefaultSurrogate));
...
```

If user wants to use some StreamParser he must register them before serializing/deserializing any data using the method `RegisterStreamParser`.

```
...
CompactFormatter CF = new CompactFormatter(CFormatterMode.SAFE);
CF.RegisterStreamParser(new ZipStreamParser());
...
```

At this point serializing/deserializing an object is done exactly as it would be using BinaryFormatter or SOAPFormatter.

```
...
CompactFormatter CF = new CompactFormatter(CFormatterMode.SAFE);
CF.AddSurrogate(typeof(DefaultSurrogate));

FileStream stream = new FileStream("data.bin",System.IO.FileMode.Create);
```

[4]BinaryFormatter and SOAPFormatter

```
int number = 42;
CF.Serialize(stream,number);
...
int number2 = (int)CF.Deserialize(stream);
...
```

2.3 Serializable Objects

As precedently written an user-defined object is serializable only if is marked with `SerializableAttribute` and if it presents a public parameterless constructor.[5]

Usually SerializableAttribute takes no parameters as in the following class declaration:

```
[Serializable()]
public class UserObject
{
// Parameterless constructor requested
// by the CompactFormatter.
public UserObject()
{}
...
}
```

When a user-defined object is declared as Custom Serializable[6] the SerializableAttribute changes as the following:

```
[Serializable(Custom = true)]
public class CustomObject : ICSerializable
{
// Parameterless constructor requested
// by the CompactFormatter.
public CustomObject()
{}
public void SendObjectData(CompactFormatter parent,
System.IO.Stream stream)
{...}
public void ReceiveObjectData(CompactFormatter parent,
System.IO.Stream stream)
{...}
}
```

[5]An exception to this rule is given by classes serializable through the usage of Surrogates or Overriders.

[6]more on this in paragraph 3.2

2.4 Transient fields

There are situations in which we're interested in serializing a class leaving out (not serializing) a set of fields of it. CompactFormatter allows this kind of *partial serialization* through the use of NotSerializedAttribute.

`NotSerializedAttribute` is an attribute which is valid on Fields of classes marked with Serializable attribute. Each field marked with `NotSerialized` attribute is simply ignored by CompactFormatter during serialization and deserialization phases.

An example can be found in the following class where the field `password` of an object containing user informations is not serialized (to avoid the risk of data being sniffed while moving on the serialization stream).

```
[Serializable()]
public class UserInfo
{
String username;
[NotSerialized()]
String password;
...
}
```

Just a word of caution: obviously after deserialization phase the deserialized object will have `NotSerialized` fields set to their default values. (In the example password won't be deserialized and therefore it will always have its default value)

3 Extension Mechanism

When working in `SAFE` mode, CompactFormatter will not serialize classes which are not *explicitely* serializable:

All primitive types are natively serializable for CompactFormatter and all user-defined classes which should be serialized (being marked with Serializable attribute) are also *explicitely* serializable.

But Framework base classes, and, generally, classes from third-party libraries not specifically written to be serialized with CompactFormatter can't be automatically serialized by CompactFormatter in `SAFE` mode. Therefore CompactFormatter give final users two mechanisms to overcome this problems:

3.1 Surrogates

Surrogates are methods characterized by signature
`Object MethodName(Type t)`
and marked with SurrogateAttribute. One of requirement of CompactFormatter to serialize a class is the existance of a parameterless constructor, this is requested by reflection to summon an instance of the received class during the deserialization phase. The problem is that some third party classes (such as some classes of the .NET Framework BCL) does not have a similar constructor, or doesn't have a constructor at all.

If this is the only problem with class serialization it can be solved easily using a Surrogate, a surrogate is simply a parameterless constructor surrogate: The surrogate method should take an object type and should return a newly created instance of type passed as parameter.

In this way, when CompactFormatter notice that a class has no parameterless constructor, checks if it has a surrogate registered for that class, and if it finds such a surrogate, invokes it and procede with automatic serialization.

The SurrogateAttribute takes a single parameter which contains the type marked class is a surrogate of, therefore a class:

```
[Surrogate(typeof(Alfa)]
[Surrogate(typeof(Beta)]
public class SurrogateOne : ISurrogate
{
Object CreateObject(Type t)
{...}
}
```

is a surrogate for types `Alfa` and `Beta`.

3.2 Overriders

Sometimes a Surrogate is not enough, in fact there are situation in which automatic serialization algorithm fails or is not efficient enough.

In these situation, if the final user has full control on the class being serialized (he is writing it) it's better to use *Custom Serialization* (see section 3.2) but if the class comes from a third party library (and therefore the final user has no control on it) CompactFormatter allows the usage of **Overriders**.

Overriders are particular classes marked with OverriderAttribute and implementing the IOverrider interface (which is very similar to ICFormatter) and are serializers under all aspects. When CompactFormatter notice that a class has an overrider registered, doesn't serialize it using default serialization mechanism but uses the serialization algorithm contained in the Overrider class.

The OverriderAttribute, such as the SurrogateAttribute, takes a single parameter which contains the type marked class is an overrider of, therefore a class:

```
[Overrider(typeof(Alfa)]
[Overrider(typeof(Beta)]
public class OverriderOne : IOverrider
{
void Serialize(CompactFormatter parent,
Stream serializationStream, object graph)
{...}
object Deserialize(CompactFormatter parent,
Stream serializationStream)
{...}
}
```

is an overrider for types Alfa and Beta.

3.3 Custom Serialization

When a user has full control on a class but, for any reason, doesn't want to use CompactFormatter Serialization routines, it's probably better to use CustomSerialization instead of Overriders to override CompactFormatter serialization.

Through CustomSerialization the class itself is responsible of its serialization and therefore it must have public methods to serialize and deserialize instances. This means that a class using Custom Serialization must implement ICSerializable (this interface is very similar to IOverrider, in fact an object able to serialize/deserialize itself is a special case of an overrider, the only differences are due to the fact that ICSerializable mimic .NET standard ISerializable interface) and its Serializable attribute (which must be present since to be CustomSerializable a class must be already Serializable) must explicitly set Custom parameter to true. As in the following class declaration:

```
[Attributes.Serializable(Custom = true)]
public class CustomSerializableObject : ICSerializable
{
public CustomSerializableObject() {}

public void SendObjectData(CompactFormatter parent,
System.IO.Stream stream)
{...}

public void ReceiveObjectData(CompactFormatter parent,
System.IO.Stream stream)
{...}
}
```

When, during serialization, CompactFormatter meets a CustomSerializable object it calls object's SendObjectData method and when , during deserialization phase, CompactFormatter meets a CustomSerializable object, it summon an instance of the object through parameterless constructor and then calls object's ReceiveObjectData.

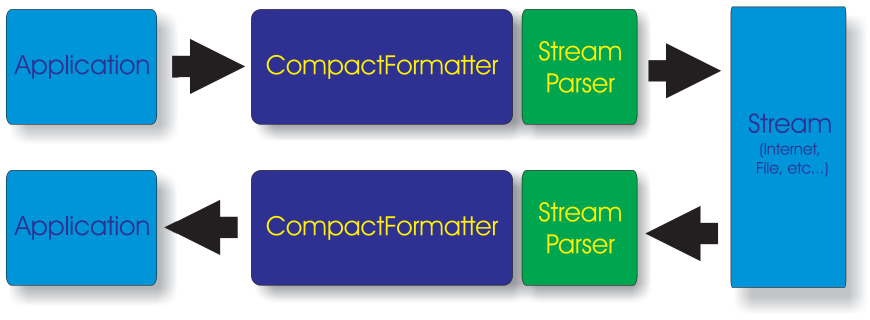

Figure 1: Serialization flow using a Stream Parser.

4 Stream Parsers

Stream Parsers are wrapper around serialization stream and can be used to add external capabilities to CompactFormatter without having to change its source code. For example, compression and encryption could be applied to the CompactFormatter serialization stream simply registering on both instances of CompactFormatter (the one serializing and the one deserializing) custom defined StreamParsers and continuing serializing/deserializing as normal as showed in figure 1.

Different StreamParser can be registered to the same instance of CompactFormatter and this means that serialization stream will be transformed and modified by all registered stream parsers, in the *exact* order in which stream parsers were registered (precisely when deserializing an object, the stream pass through stream parsers in reverse order to guarantee correctness as showed in figure 2).

A StreamParser is simply a class inheriting from the abstract class IStreamParser and therefore implementing two methods:

- `void ParseOutput(ref byte[] buffer, int offset, int len);`

  Used during serialization phase just before sending a byte array on the stream.

- `int ParseInput(ref byte[] buffer, int offset, int len);`

  Used during deserialization phase as soon as data has been received from the stream.

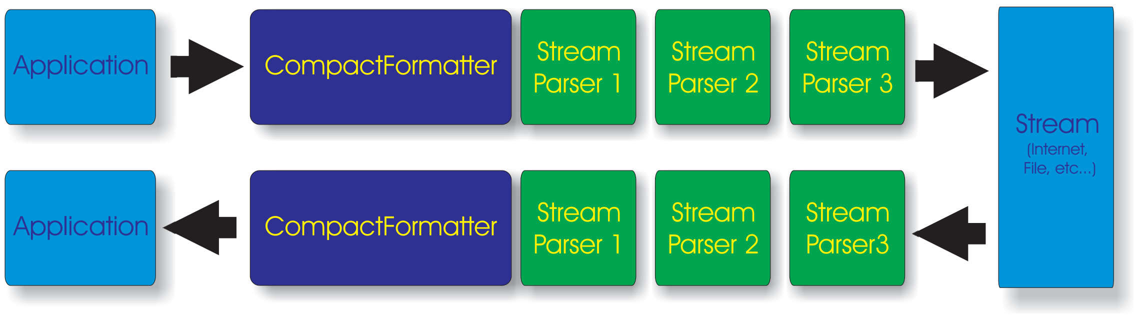

Figure 2: Serialization flow using several Stream Parsers.