eProsima Fast Buffers

User Manual
Version 0.3.0

The Middleware Experts

eProsima
Proyectos y Sistemas de Mantenimiento SL
Ronda del poniente 2 – 1ºG
28760 Tres Cantos Madrid
Tel: + 34 91 804 34 48
info@eProsima.com – www.eProsima.com

Trademarks

eProsima is a trademark of *Proyectos y Sistemas de Mantenimiento SL*. All other trademarks used in this document are the property of their respective owners.

License

eProsima Fast Buffers is licensed under the terms described in the FAST_BUFFERS_LICENSE file included in this distribution.

Technical Support

- Phone: +34 91 804 34 48
- Email: support@eprosima.com

Contents

1 Introduction

eProsima Fast Buffers is a high-performance serialization library capable of translating structured data into a format that can be stored and recovered later in the same or another computer environment. You have to define your structured data once, and then you can use specific generated source code to easily serialize and deserialize it in different operating systems.

To declare your structured data, you have to use IDL (Interface Definition Language) format. IDL is a specification language, made by OMG (Object Management Group), which describes an interface in a language-independent way, enabling communication between software components that do not share the same language.

The main component of *eProsima Fast Buffers* is a tool that reads IDL files and parses a subset of the OMG IDL specification to generate serialization source code. This subset includes only the data type descriptions included in section 3 (Defining your data types). The rest of the file content is ignored.

eProsima Fast Buffers has another relevant component: a C++11 library which provides different serialization mechanisms. Currently two serialization mechanisms are supported:

One of them is the standard CDR (Common Data Representation)[1]. CDR is a transfer syntax low-level representation for transfer between agents, mapping from data types defined in OMG IDL to byte streams.

The other one is a modified implementation of CDR, which has an overwhelming speed-up compared to the classical approach. The reason why this implementation is faster is because it does not use some of the features of the standard CDR serialization, in order to improve performance.

One of the main features of *eProsima Fast Buffers* is it avoids users from knowing anything about serialization or deserialization procedures. For that task, it provides a homogeneous API which supports serialization of several different data types.

With this document you are about to learn how to use *eProsima Fast Buffers* components, improving the communications in your applications by using a fast serialization. It is structured this way:

- Section 1 (Introduction) presents *eProsima Fast Buffers*.
- You begin learning how to build source code for serializing data types in section 2 (Building your serialization code).
- All supported data types by *eProsima Fast Buffers* are listed in section 3 (Defining your data types). This section also shows how to define these data types using the IDL definition language.
- Section 4 (Common API) explains the common API used by the generated source code.
- In section 5 (Integrating new source code) you learn how to integrate the new source code in your application.
- In section 6 (HelloWorld example) a complete HelloWorld example is shown.

1 Visit http://www.omg.org/cgi-bin/doc?formal/02-06-51 for more information about CDR serialization standard

This release of *eProsima Fast Buffers* only supports the generation of source code in C++11. Forthcoming releases will integrate more programming languages.

2 Building your serialization code

eProsima Fast Buffers provides a Java application responsible for generating source code using the data types defined in an IDL file. This generated source code can be used in your applications in order to serialize your data types. This section guides you through the usage of this Java application and briefly describes the generated files.

The Java application can be executed using the following script:

In Windows:

```
> fastbuffers.bat
```

Note: The script will be found by Windows if its location has been set in the PATH environment variable. This can be done during the installation process. More information can be found in the Installation Manual.

In Linux:

```
$ fastbuffers.sh
```

eProsima Fast Buffers Java application is capable of processing several IDL files. The expected argument line is:

```
fastbuffers [<options>] <IDL file> [<IDL file> ...]
```

where the available options are:

Option	**Description**
-help	Shows the help information.
-version	Shows the current version of *eProsima Fast Buffers*.
-d <directory>	Output directory where the generated files are created.
-example <platform>	Generates an example and a solution to compile the generated source code for a specific platform. The help command shows the supported platforms.
-replace	Replaces the generated source code files whether they exist.
-ser <cdr | fastcdr>	Serialization mechanism used. *eProsima Fast Buffers* currently supports two serialization mechanisms. CDR serialization is the default mechanism, but it also supports a Fast CDR serialization.

For each processed IDL file *eProsima Fast Buffers* tool generates four source code files. Next diagram shows the generated source code files for a *MyTypes.idl* example IDL file.

Next table summarizes the content of each generated file.

File	Description
MyTypes.h	C++ Header file containing the user data types mapped to C++11 programming language.
MyTypes.cpp	C++ Source file containing the user data types mapped to C++11 programming language.

3 Defining your data types

You have to define your data types using IDL format. Not all IDL features are supported in this version of *eProsima Fast Buffers*. This section shows you what IDL types are supported and how *eProsima Fast Buffers* maps them in C++11 programming language.

For each IDL file parsed by *eProsima Fast Buffers* two source code files are generated. They contain the C++11 mapping of your IDL types. For example, from an IDL file named *Foo.idl*, *eProsima Fast Buffers* tool generates the files *Foo.h* and *Foo.cpp* that contain the C++11 mapping of the defined IDL types.

Next subsections introduce the supported IDL types.

3.1 Basic types

The following table shows the basic IDL types supported by *eProsima Fast Buffers* and how they are mapped to C++11.

IDL	C++11
`char vartype;`	`char vartype;`
`octet vartype;`	`uint8_t vartype;`
`short vartype;`	`int16_t vartype;`
`unsigned short vartype;`	`uint16_t vartype;`
`long vartype;`	`int32_t vartype;`
`unsigned long vartype;`	`uint32_t vartype;`
`long long vartype;`	`int64_t vartype;`
`unsigned long long vartype;`	`uint64_t vartype;`
`float vartype;`	`float vartype;`
`double vartype;`	`double vartype;`
`boolean vartype;`	`bool vartype;`
`string vartype;`	`std::string vartype;`

3.2 Arrays

eProsima Fast Buffers supports arrays and multidimensional arrays. An array is mapped to the STD array container. The following table shows the supported array types and how they are mapped.

IDL type	C++11 type
`char arraytype[5];`	`std::array<char, 5> arraytype;`
`octet arraytype[5]`	`std::array<uint8_t, 5> arraytype;`
`short arraytype[5]`	`std::array<int16_t, 5> arraytype;`
`unsigned short arraytype[5]`	`std::array<uint16_t, 5> arraytype;`
`long arraytype[5]`	`std::array<int32_t, 5> arraytype;`
`unsigned long arraytype[5]`	`std::array<uint32_t, 5> arraytype;`

`long long arraytype[5]`	`std::array<int64_t, 5> arraytype;`
`unsigned long long arraytype[5]`	`std::array<uint64_t, 5> arraytype;`
`float arraytype[5]`	`std::array<float, 5> arraytype;`
`double arraytype[5]`	`std::array<double, 5> arraytype;`

3.3 Sequences

eProsima Fast Buffers supports sequences. A sequence is mapped to the STD vector container. The following table shows the supported sequence types and how they are mapped.

IDL	C++11
`sequence<char> sequencetype;`	`std::vector<char> sequencetype;`
`sequence<octet> sequencetype;`	`std::vector<uint8_t> sequencetype;`
`sequence<short> sequencetype;`	`std::vector<int16_t> sequencetype;`
`sequence<unsigned short> sequencetype;`	`std::vector<uint16_t> sequencetype;`
`sequence<long> sequencetype;`	`std::vector<int32_t> sequencetype;`
`sequence<unsigned long> sequencetype;`	`std::vector<uint32_t> sequencetype;`
`sequence<long long> sequencetype;`	`std::vector<int64_t> sequencetype;`
`sequence<unsigned long long> sequencetype;`	`std::vector<uint64_t> sequencetype;`
`sequence<float> sequencetype;`	`std::vector<float> sequencetype;`
`sequence<double> sequencetype;`	`std::vector<double> sequencetype;`

3.4 Structures

In IDL, a structure is defined as a set of members, each one with its own type. An IDL structure type is mapped as a C++ class, while each member is mapped as a class attribute. A set of methods to get and set the attribute is also created. The following table shows and example of an IDL structure type and its mapping to the C++11 programming language.

IDL	C++11

struct Structure
{
    octet octet_value;
    long long_value;
    string string_value;
};
	class Structure
	{
	public:
	   Structure();
	   ~Structure();
	   Structure(const Structure &x);
	   Structure(Structure &&x);
	   Structure& operator=(const Structure &x);
	   Structure& operator=(Structure &&x);
	
	   void octet_value(uint8_t _octet_value);
	   uint8_t octet_value() const;
	   uint8_t& octet_value();
	   void long_value(int64_t _long_value);
	   int64_t long_value() const;
	   int64_t& long_value();
	   void string_value(const std::string
	      &_string_value);
	   void string_value(std::string &&_string_value);
	   const std::string& string_value() const;
	   std::string& string_value();
	
	private:
	   uint8_t m_octet_value;
	   int64_t m_long_value;
	   std::string m_string_value;
	};

3.5 Unions

In IDL, a union is defined as a sequence of members with their own types and a discriminant that specifies which member is in use. An IDL union type is mapped as a C++ class with access functions to the union members and the discriminant. The following table shows an example of IDL union type and its mapping to the C++11 programming language.

IDL	C++11

union Union switch(long)
{
  case 1:
    octet octet_value;
  case 2:
    long long_value;
  case 3:
    string string_value;
};
	class Union
	{
	public:
	   Union();
	   ~Union();
	   Union(const Union &x);
	   Union(Union &&x);
	   Union& operator=(const Union &x);
	   Union& operator=(Union &&x);
	
	   void d(int32t __d);
	   int32_t _d() const;
	   int32_t& _d();
	
	   void octet_value(uint8_t _octet_value);
	   uint8_t octet_value() const;

	uint8_t& octet_value();
	void long_value(int64_t _long_value);
	int64_t long_value() const;
	int64_t& long_value();
	void string_value(const std::string
	&_string_value);
	void string_value(std::string &&_string_value);
	const std::string& string_value() const;
	std::string& string_value();
	
	private:
	int32_t m__d;
	uint8_t m_octet_value;
	int64_t m_long_value;
	std::string m_string_value;
	};

3.6 Enumerations

An enumeration in IDL format is a collection of identifiers that have a numeric value associated. An IDL enumeration type is mapped directly to the corresponding C++11 enumeration definition. The following table shows an example of an IDL enumeration type and its mapping to the C++11 programming language.

IDL	C++11

enum Enumeration
{
RED,
GREEN,
BLUE
};
	enum Enumeration : uint32_t
	{
	RED,
	GREEN,
	BLUE
	};

4 Common API

All generated data types contain the necessary functionality to be serialized or deserialized using the selected mechanism. This section shows you the API provided to serialize your data types. For more detailed information read the C++11 API Manual.

eProsima FastBuffers provides two classes which implement the supported serialization mechanisms. The *eprosima::fastcdr::Cdr* class implements the CDR serialization and the *eprosima::fastcdr::FastCdr* class implements the FastCDR serialization. This section splits this API in three categories and shows examples using both serialization mechanisms.

Initialization API

eprosima::fastcdr::Cdr and *eprosima::fastcdr::FastCdr* classes need a raw buffer to read or write serialized data. This raw buffer must be previously managed by an object of the class *eprosima::fastcdr::FastBuffer*.

Here is an example of the creation of both *eprosima::fastcdr::FastBuffer* and *eprosima::fastcdr::Cdr* objects:

```
// Raw buffer where data is serialized.
char buffer[500];
// Object that manages the raw buffer.
eprosima::fastcdr::FastBuffer fastbuffer(buffer, 500);
// Object that serializes the data.
eprosima::fastcdr::Cdr ser(fastbuffer);
```

An example of the creation of *eprosima::fastcdr::FastBuffer* and *eprosima::fastcdr::FastCdr* objects:

```
// Raw buffer where data is serialized.
char buffer[500];
// Object that manages the raw buffer.
eprosima::fastcdr::FastBuffer fastbuffer(buffer, 500);
// Object that serializes the data.
eprosima::fastcdr::FastCdr ser(fastbuffer);
```

Serialization API

The *eprosima::fastcdr::Cdr* class offers two ways to serialize each data type:

- `eprosima::fastcdr::Cdr& operator<< (const type&)`

  This operator serializes a data value of the type *type*.

- `eprosima::fastcdr::Cdr& serialize (const type&)`

  This method serializes a data value of the type *type*.

The *eprosima::fastcdr::FastCdr* class offers two ways to serialize each data type:

- `eprosima::fastcdr::FastCdr& operator<< (const type&)`

  This operator serializes a data value of the type *type*.

- `eprosima::fastcdr::FastCdr& serialize (const type&)`

  This method serializes a data value of the type *type*.

Deserialization API

The *eprosima::fastcdr::Cdr* class offers two ways to deserialize.

- `eprosima::fastcdr::Cdr& operator>> (type&)`

  This operator deserializes a data value of the type *type*.

- `eprosima::fastcdr::Cdr& deserialize (type&)`

  This method deserializes a data value of the type *type*.

The *eprosima::fastcdr::FastCdr* class offers two ways to deserialize.

- `eprosima::fastcdr::FastCdr& operator>> (type&)`

  This operator deserializes a data value of the type *type*.

- `eprosima::fastcdr::FastCdr& deserialize (type&)`

  This method deserializes a data value of the type *type*.

Here is an example of the serialization and deserialization of a Foo object:

```
// Variable of the type Foo defined in Foo.idl
Foo foo;
// Serializing the variable with the previously created serialization object.
ser << foo;
// Deserializing the variable with the previously created serialization object.
ser >> foo;
```

5 Integrating new source code

In order to integrate the generated source code in your application you have to take some points into consideration. This section shows these points and how to deal with them.

eProsima Fast Buffers tool provides an option to generate and compile working examples. You can obtain detailed information about integrating the generated code in your applications by using that option.

Including C++11 headers

To compile the generated source code in your application you must specify the location of the C++11 headers used by this source code. These headers are located in the *eProsima Fast Buffers* installation directory.

In Windows:
`%FAST_BUFFERS%\include`

In Linux: they are found automatically because they were installed in `/usr/include`

Linking against *eProsima Fast Buffers* library

eProsima Fast Buffers contains a serialization library. This library gathers all supported serialization mechanisms. Your application must be linked against this library in order to achieve a successful compilation. This library is located in the *eProsima Fast Buffers* installation directory.

In Windows:
`%FAST_BUFFERS%\lib`

In Linux: it is found automatically because it was installed in `/usr/lib`

6 HelloWorld example

This section explains how to write a HelloWorld example step by step. This example has a simple structure with a string to serialize/deserialize.

6.1 Writing the IDL file

The first step is to write an IDL file containing the structures you want to serialize/deserialize. In this case, it is a simple structure:

```
struct HelloWorld
{
    string message;
};
```

Save this file as *HelloWorld.idl*.

6.2 Generating specific code

Open a command prompt and go to the directory containing *HelloWorld.idl*.

If you are running this example in Windows, execute the following line:

```
fastbuffers -example x64Win64VS2010 HelloWorld.idl
```

If you are running it in Linux, execute this one:

```
fastbuffers -example x64Linux2.6gcc HelloWorld.idl
```

Note that if you are running *eProsima Fast Buffers* in a 32-bit operating system you have to use *-example i86Win32VS2010* or *-example i86Linux2.6gcc* instead.

This command generates your serialization code and some project files designed to build your HelloWorld example. In Windows, it generates a Visual Studio 2010 solution, named solution-x64Win64VS2010.*sln*. This solution has two projects: *HelloWorld*, with the generated serialization code, and *HelloWorldExample*, with a usage example. In Linux, on the other hand, it generates a makefile.

6.3 Using the common API

Edit the file *HelloWorldExample.cpp*. It takes a HelloWorld object (defined as a structure in your IDL file), serializes and deserializes it into another HelloWorld object. You just have to add the initialization code and a statement to print the result, like this example does:

```
int main()
{
    FastBuffer fastbuffer;
    Cdr HelloWorld_ser(fastbuffer);

    // Structure to serialize.
    HelloWorld ser_var;
```

```
    // The structure has to be initialized.
    ser_var.message("Hello World!");

    // Serialization.
    HelloWorld_ser << ser_var;

    // Reset the reading position in the serializer object to start
deserialization.
    HelloWorld_ser.reset();

    // The buffer will be deserialized in the next structure.
    HelloWorld des_var;

    // Deserialization.
    HelloWorld_ser >> des_var;

    // Print the deserialized message
    cout << des_var.message() << endl;

    return 0;
}
```

Once your structure has been serialized, if you want to obtain the corresponding char* buffer you just have to invoke *fastbuffer.getBuffer()* method.

6.4 Building and executing

To build your code using Visual Studio 2010, make sure you are in the Debug (or Release) profile, and then build it (F7). Now go to `<example_dir>\bin\x64Win64VS2010` directory and execute HelloWorldExample.exe. You will get the message:

```
Hello World!
```

This example was created statically. To create a DLL containing the serializators and the structures, select the Debug DLL (or Release DLL) profile and build it (F7). Now, to get your .dll and .lib go to `<example_dir>\objs\x64Win64VS2010` directory. You can now run the same application dynamically using the .exe file generated in `<example_dir>\bin\x64Win64VS2010`, but first you have to make sure your .dll location directory is appended to the PATH environment variable.

To build your code in Linux use this command:

```
make -f makefile_x64Linux2.6gcc
```

Now go to `<example_dir>\bin\x64Linux2.6gcc` directory and execute *HelloWorld*.

In both scenarios you will get this message when you run the binary file:

```
Hello World!
```