Subject: Flatbuffers package
Posted by Xemuth on Tue, 15 Dec 2020 11:48:36 GMT
View Forum Message <> Reply to Message

Hello,

I had to work with Google Flatbuffers and for my test I have made a TCP/IP client with Upp (in order to connect to server (which work with flatbuffer)).
That's why I'm sharing here my Flatbuffers package and a quick client/server exemple.

What is Flatbuffers ?

FlatBuffers is an efficient cross platform serialization library for C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust and Swift. It was originally created at Google for game development and other performance-critical applications.

How Flatbuffers work ?

FlatBuffers provide an executable (named flatc.exe)(present in bin folder of my package, Windows only, I plan to add debian/linux executable too) capable of converting fbs file to header file representing data. Let me show a simple example :

Command.fbs :

```
// My command fbs

namespace TcpIpServer;

table Command {
  id:int;
  name:string;

}

root_type Command;
```

In this simple fbs file we can quickly remarque a namespace, a table (wich can basicly be interpreted as structure or object) and a root_type. This file describe layout of our data.
-The schema starts with a namespace declaration. This determines the corresponding package/namespace for the generated code. In our example, we have the TcpIpServer namespace
-The Command table is the main object in our FlatBuffer. This will be used as the template to store all our defined command.
-The last part of the schema is the root_type. The root type declares what will be the root table for the serialized data. In our case, the root type is our Command table

You can find a far more precise description of FBS file here : Flatbuffers tutorial

By passing this fbs file to flatc.exe as follow:

```
~/flatc.exe --cpp command.fbs
```

it gonna generate an header file (in our case named command_generated.h) contening a huge implementation of our data structure (you can find this file here if you are curious !)
this header file will be inserted in our project as follow :

```
#include <Core/Core.h>
#include "command_generated.h"

using namespace Upp;
...
```

Right now flatbuffers is inserted in our project and can be used to Serialize/Deserialize command object. Lets quickly see how it's done :

```
//a simple Command struct
struct Command{
 public:
  int id;
  String name;

  Command(int _id, String _name){id = _id; name = _name;}

  String ToString()const{
   String toStr;
   toStr << "Id: " << id << ", Name: " << name << "\n";
   return toStr;
  }
};

struct CommandSerializer{
 public:
  CommandSerializer(const Command& cmd){
   flatbuffers::Offset<flatbuffers::String> str1 = builder.CreateString(cmd.name.ToStd()); //First, we
create a special flatbuffers object that handle string
   TcpIpServer::CommandBuilder commandBuilder(builder); //Then we create a commandBuilder
(a special object that will construct our Command (the fbs file))
   commandBuilder.add_id(cmd.id); //We define value of each parameters of our command
   commandBuilder.add_name(str1); //We define value of each parameters of our command
   flatbuffers::Offset<TcpIpServer::Command> freshCommand = commandBuilder.Finish(); //we
retrieve our command by telling our commandBuilder we are done with it
   builder.Finish(freshCommand); //we end the builder of our newly command (named
freshCommand)
  }

  int GetSize(){return builder.GetSize();} //Return a ptr to our serialized object
  uint8_t* GetDatasPtr(){return builder.GetBufferPointer();} //Return size of our serialized object

 private:
  flatbuffers::FlatBufferBuilder builder; //Builder is used to create everything we want with
flatbuffers
};

Command CommandDeserializer(uint8_t* datas, int size){
 flatbuffers::Verifier verfier(datas,size); //Verifier is used to verify our buffer of data is a valid
command
 if(TcpIpServer::VerifyCommandBuffer(verfier)){
  const TcpIpServer::Command* serializedCmd = TcpIpServer::GetCommand(datas); //If our
buffer is valid then we retrieve a ptr to the command within this buffer
  return Command(serializedCmd->id(), String(serializedCmd->name()->c_str())); //then we use
our buffer to build our command
 }
 throw Exc("Invalide command");
}
```

the ptr to the binary serialized data can be used to sent data over the network for example. Lets see in the next post a simple Client/Server flatbuffer exchange

File Attachments

1) Flatbuffers.7z, downloaded 303 times

Subject: Re: Flatbuffers package

Posted by Xemuth on Tue, 15 Dec 2020 11:50:10 GMT

View Forum Message <> Reply to Message

Server flatbuffers example:

```
#include <Core/Core.h>
#include "flatbuffer-command/command_generated.h"

using namespace Upp;

//a simple Command struct
struct Command{
 public:
  int id;
  String name;

  Command(int _id, String _name){id = _id; name = _name;}
  String ToString()const{
   String toStr;
   toStr << "Id: " << id << ", Name: " << name;
   return toStr;
  }
};

struct CommandSerializer{
 public:
  CommandSerializer(const Command& cmd){
   flatbuffers::Offset<flatbuffers::String> str1 = builder.CreateString(cmd.name.ToStd()); //First, we create a special flatbuffers object that handle string
   TcpIpServer::CommandBuilder commandBuilder(builder); //Then we create a commandBuilder (a special object that will construct our Command (the fbs file))
   commandBuilder.add_id(cmd.id); //We define value of each parameters of our command
   commandBuilder.add_name(str1); //We define value of each parameters of our command
   flatbuffers::Offset<TcpIpServer::Command> freshCommand = commandBuilder.Finish(); //we retrieve our command by telling our commandBuilder we are done with it
   builder.Finish(freshCommand); //we end the builder of our newly command (named freshCommand)
  }

  int GetSize(){return builder.GetSize();} //Return a ptr to our serialized object
  uint8_t* GetDatasPtr(){return builder.GetBufferPointer();} //Return size of our serialized object

 private:
  flatbuffers::FlatBufferBuilder builder; //Builder is used to create everything we want with flatbuffers
};

Command CommandDeserializer(const uint8_t* datas, int size){
 flatbuffers::Verifier verfier(datas,size); //Verifier is used to verify our buffer of data is a valid command
 if(TcpIpServer::VerifyCommandBuffer(verfier)){
  const TcpIpServer::Command* serializedCmd = TcpIpServer::GetCommand(datas); //If our buffer is valid then we retrieve a ptr to the command within this buffer
  return Command(serializedCmd->id(), String(serializedCmd->name()->c_str())); //then we use our buffer to build our command
 }
 throw Exc("Invalide command");
}

CONSOLE_APP_MAIN
{
 StdLogSetup(LOG_COUT|LOG_FILE);
 TcpSocket server;
 if(!server.Listen(3214, 5)) {
  LOG("Unable to initialize server socket!\n");
  SetExitCode(1);
  return;
 }
 LOG("Waiting for requests..");
 for(;;) {
  TcpSocket s;
  if(s.Accept(server)){

   String datas;
   dword sizeReceived;
   if(s.Get(&sizeReceived, sizeof(sizeReceived)))
    datas = s.Timeout(5000).Get(sizeReceived);
   try{
    LOG(CommandDeserializer((uint8_t*)datas.ToStd().c_str(),datas.GetLength()));

    CommandSerializer cmdValide(Command(0,"Valide"));
    dword sizeToSend = cmdValide.GetSize();
    if(s.Put(&sizeToSend,sizeof(sizeToSend)))
     s.Timeout(5000).Put(cmdValide.GetDatasPtr(),cmdValide.GetSize());
   }catch(Exc& exception){
    CommandSerializer cmdInvalide(Command(-1,"Invalide"));
    dword sizeToSend = cmdInvalide.GetSize();
    if(s.Put(&sizeToSend,sizeof(sizeToSend)))
     s.Timeout(5000).Put(cmdInvalide.GetDatasPtr(),cmdInvalide.GetSize());
   }
  }
 }
}
```

File Attachments

1) ServerFlatbuffers.7z, downloaded 296 times

Subject: Re: Flatbuffers package
Posted by Xemuth on Tue, 15 Dec 2020 11:51:27 GMT
View Forum Message <> Reply to Message

client flatbuffers example:

```
#include <Core/Core.h>
#include "flatbuffer-command/command_generated.h"

using namespace Upp;

//a simple Command struct
struct Command{
 public:
  int id;
  String name;

  Command(int _id, String _name){id = _id; name = _name;}

  String ToString()const{
   String toStr;
   toStr << "Id: " << id << ", Name: " << name ;
   return toStr;
  }
};

struct CommandSerializer{
 public:
  CommandSerializer(const Command& cmd){
   flatbuffers::Offset<flatbuffers::String> str1 = builder.CreateString(cmd.name.ToStd()); //First, we create a special flatbuffers object that handle string
   TcpIpServer::CommandBuilder commandBuilder(builder); //Then we create a commandBuilder (a special object that will construct our Command (the fbs file))
   commandBuilder.add_id(cmd.id); //We define value of each parameters of our command
   commandBuilder.add_name(str1); //We define value of each parameters of our command
   flatbuffers::Offset<TcpIpServer::Command> freshCommand = commandBuilder.Finish(); //we retrieve our command by telling our commandBuilder we are done with it
   builder.Finish(freshCommand); //we end the builder of our newly command (named freshCommand)
  }

  int GetSize(){return builder.GetSize();} //Return a ptr to our serialized object
  uint8_t* GetDatasPtr(){return builder.GetBufferPointer();} //Return size of our serialized object

 private:
  flatbuffers::FlatBufferBuilder builder; //Builder is used to create everything we want with flatbuffers
};

Command CommandDeserializer(const uint8_t* datas, int size){
 flatbuffers::Verifier verfier(datas,size); //Verifier is used to verify our buffer of data is a valid command
 if(TcpIpServer::VerifyCommandBuffer(verfier)){
  const TcpIpServer::Command* serializedCmd = TcpIpServer::GetCommand(datas); //If our buffer is valid then we retrieve a ptr to the command within this buffer
  return Command(serializedCmd->id(), String(serializedCmd->name()->c_str())); //then we use our buffer to build our command
 }
 throw Exc("Invalide command");
}

Command SendCommand(const Command& cmd){
 TcpSocket s;
 s.Timeout(600);
 if(!s.Connect("127.0.0.1", 3214)){
  throw Exc("Unable to connect to server!");
 }
 CommandSerializer serializer(cmd);

 dword sizeToSend = serializer.GetSize();
 if(s.Put(&sizeToSend,sizeof(sizeToSend)))
  s.Timeout(5000).Put(serializer.GetDatasPtr(),serializer.GetSize());

 String datas;
 dword sizeReceived;
 if(s.Get(&sizeReceived, sizeof(sizeReceived)))
  datas = s.Timeout(5000).Get(sizeReceived);

 return CommandDeserializer((const uint8_t*)datas.ToStd().c_str(),datas.GetLength());
}


CONSOLE_APP_MAIN{
 StdLogSetup(LOG_COUT | LOG_FILE);
 try{
  LOG(SendCommand(Command(3,"Exemple")));
  LOG(SendCommand(Command(3,"Exemple2")));
  LOG(SendCommand(Command(3,"Exemple3")));
 }catch(Exc& exception){
  LOG(exception);
 }
}
```

File Attachments

1) ClientFlatbuffers.7z, downloaded 311 times

Subject: Re: Flatbuffers package
Posted by Xemuth on Tue, 15 Dec 2020 11:53:27 GMT
View Forum Message <> Reply to Message

the client/server example should work. however, in my computer for whatever reason (my code seems correct comparing to socketClient/socketServer package) the data is received on server only when I close the client. If someone know why, I will fix it !

Subject: Re: Flatbuffers package
Posted by Oblivion on Tue, 15 Dec 2020 12:15:13 GMT
View Forum Message <> Reply to Message

Hello Xemuth,

Quote:. however, in my computer for whatever reason (my code seems correct comparing to socketClient/socketServer package) the data is received on server only when I close the client.

I'm on Linux now, but from the examples you've posted, my guess is that your calls are "blocking" (I don't see a Timeout value is set to 0 (non-blocking) or > 0 (time-constrained), which means it is blocking (Timeout == Null, by default).

```
int numberByteRead = s.Get(datas, 2048);
```

From docs:

int TcpScoket::Get(void *buffer, int len)

Reads at most len bytes into buffer, trying to do so at most for specified timeout. Returns the number of bytes actually read.

See, TcpSocket::Get() variant you use seems to expect 2048 bytes. And if the call is blocking, unless it receives >= 2048 bytes, it will block.

Upp ServerSocket/ClientSocket example, on the other hand, uses TcpSocket::GetLine(), which expects '\n', and as you can see in the examples, that is always provided.

Best regards,
Oblivion

Subject: Re: Flatbuffers package
Posted by Xemuth on Tue, 15 Dec 2020 12:18:54 GMT
View Forum Message <> Reply to Message

Hello Oblivion,

what an error... you got it thanks and sorry for losing your time

PS: my example is not properly working, I'm updating it right now

Subject: Re: Flatbuffers package
Posted by Xemuth on Tue, 15 Dec 2020 13:41:28 GMT
View Forum Message <> Reply to Message

the example is now working, source have been updated, I guess adding '\n' to every packet of data is ugly. if you have a better way of handling this without having to set a tiny timeout ...

Subject: Re: Flatbuffers package
Posted by Oblivion on Tue, 15 Dec 2020 14:22:35 GMT
View Forum Message <> Reply to Message

Quote:, I guess adding '\n' to every packet of data is ugly

Ah, I didn't mean that. :) What I was trying to say is that the socket examples are different then yours, and it is not suprising that you've encountered difficulties.

Since this is going to be a simple example why don't you just send the size of the data to be transferred first?
Let the other side fetch that first, and then call Get() again with the size and timeout in 30 secs.

Example (server side):

```
String data;
dword len = data.GetLength();
if(socket.Put(&len, sizeof(dword))
 socket.Timeout(30000).Put(~data, len);
```

Same thing goes for the client side. First fetch the size, then use it to fetch the data.

P.s: And if you are sending binary data over the network, it is in general a good practice to convert it to base64 encoding).

Best regards,
Oblivion

Subject: Re: Flatbuffers package
Posted by Xemuth on Tue, 15 Dec 2020 17:15:01 GMT
View Forum Message <> Reply to Message

Quote:Since this is going to be a simple example why don't you just send the size of the data to

be transferred first?
Let the other side fetch that first, and then call Get() again with the size and timeout in 30 secs.
 Indeed it is smart ! Exemple have been udpated.

Quote:P.s: And if you are sending binary data over the network, it is in general a good practice to convert it to base64 encoding) I think protobuff do it by himself. I will check this later

thanks for all advise !