Solomon

version 2.4

Introduction

This guide is just a starter reference to help you to be proficient with library and build a complex infrastructure in just few days without worry about how it works, how it communicate to and complexity behind the real-time server it provide.
Using this library you can create every kind of logical protocols you need, every kind of server plug-ins to manage your requests (connect to a database, send sms or email, download over http or whatever you want) without go through the complex details of integration.
The library provides you a ready-to-use configuration plug-in in which you can easily configure behaviours of your server, such as database connection info (actually it only works with mysql server), number of thread pools, number of worker threads, some default polices to permit upload of files, a maximum size of those files and so on...
The configuration plug-in is the first operation server executes to load the custom behaviours you want to set and then it starts procedure to load all handle plug-ins (or command plug-ins) to manage the single requests of your own protocol. When the server start, it falls in listen on port you have specified (configured in configuration plug-in) using both UDP/IP and TCP/IP protocols and is able to receive/send on both protocols, but we recommend you to respond over TCP/IP to avoid lost of packets due to firewall blocks and to have a more reliable servers.
Solomon implements a persistent server capability; it means a connection will always be active and working until it will be marked as inactive and then closed by server (see configuration plugin details). When this happen, client will be notified about this and automatically closes its connection. To avoid expiration time due to inactivity, it's up to you to create handle and request to update connection time. We will explain it further next in documentation.
Whereas server side can only be actually created on *nix machines (due to some *nix features), the client side of library has actually a cross platform nature (usually all system supporting C runtime) and with simple steps you can easily send your own requests over TCP/IP or UDP/IP using asynchronous or synchronous ways, send files and receive notifications sent by servers.

Details of ready-to-use configuration plug-in

The heart of server can be of course tailored on you own wishes and needs, thanks to configuration plug-in composed by two parts (library with .so extension and file with .config extension);
so let's start to see how it looks like:

```
<transport>tcp://1.2.3.4:3306/mydb</transport>
<conn_timeout>4</conn_timeout>
<conn_inactive_secs>1200</conn_inactive_secs>
<conn_clean_secs>30</conn_clean_secs>
<username>myuser</username>
<password>mypasswd</password>
<server_ip>1.2.3.4.5</server_ip>
<server_port>1234</server_port>
<plugins_path>/home/server/plugins/</plugins_path>
<permission_file_upload>1</permission_file_upload>
<max_bytes_file_upload>1024</max_bytes_file_upload>
<system_upload_root>/home/repository/files/</system_upload_root>
<link_download_root>http://www.mysite.net/transfer/</link_download_root>
<worker_thread>1</worker_thread>
<number_pool_threads>2</number_pool_threads>
<trusted_passwd>trusted_password</trusted_passwd>
```

Let's see in detail meaning of each property:

- <transport> the value of this property represents the transport to connect to mysql server (if there is any); more informations on mysql reference guide.
- <conn_timeout> the value of this property indicates how many seconds a thread needs to wait to connect to database.
- <conn_inactive_secs> the value of this property indicates how many seconds of inactivity are necessary to mark a connection as to be closed by server.
- <conn_clean_secs> the value of this property indicates the interval in seconds to start check of inactive connections to clean and close.
- <username> the value of this property indicates user name to be used to get authenticated on database server.
- <password> the value of this property indicates password to be used to get authenticated on database server.
- <server_ip> the value of this property indicates IP of machine where the server instance is running on.
- <server_port> the value of this property indicates port number where the server instance is listen to on.
- <plugins_path> path in the file system (it can also be a remote mounted directory) where server instance can load handle plug-ins to manage requests; path need to be a directory and must terminate with “/” like in above example.
- <permission_file_upload> the value of this property indicates default permission to upload a file when a new connection is initially established (it can be programmatic changed tailored on each user connection) and must have value of 1 to enable permission; 0 otherwise.
- <max_bytes_file_upload> the value of this property indicates default size of file server can accept as upload (it can be programmatic changed tailored on each user connection). Value is expressed in bytes.

- <system_upload_root> path in the file system (it can also be a remote mounted directory) where server instance stores uploaded files; path need to be a directory and must terminate with "/" like in above example.
- <link_download_root> a way to download uploaded file. It can be every protocol that maps <system_upload_root> physical path. When a file has been uploaded, server creates a new request sending a complete link where to download uploaded file. This technique is useful in cluster environment so you don't have to worry about creating different storage and different public download points.
- <worker_thread> number of worker threads server needs to use. A worker thread has its own thread pool to manage to consume the main queue. Generally the number of worker thread to use is based on number of CPUs.
- <number_pool_threads> indicates how many threads a pool will have per worker thread. This value cannot be less then of 2 per worker thread otherwise server won't work.
- <trusted_passwd> as the name suggest, this is an internal password you can use just to trust some operations that need a little of protection. Handle plug-ins loaded by server are always public, so it is up to you limit access, for example, using authentication or authorization comparing data in a request with your trusted source (e.g. Database or whatever) or using a trusted password. Use this password only when a request is generate internally and need to be handled by exclusive handle plug-in.

You can use the name you want for the configuration plug-in, but remember configuration file need to have the same name plus ".config" extension.
For example if you have, say "**lib_odbc.so**", you will have to rename configuration file like: "**lib_odbc.so.config**". Generally we pass path and configuration plug-in file as parameter of server executable at command line.
After this little introduction, we start to cover how typically you manage your own source codes in a tree of projects.

Organization of source codes

Generally Solomon base library is always linked to all projects we are going to create:

1. Create your **custom library project** linking Solomon base library. In this library you put all of your logical request/response objects, server custom bridge and all other stuff needed.
2. Create your **handle command projects** linking Solomon base library and your custom library and all other libraries you are going to use (e.g. database or something like that).
3. Create your **server project** linking references to Solomon base library and your own custom library you have made (at point 1.)
4. (**optional**) create your **custom configuration plug-in project** with all dependencies you think are necessary.

We recommend to build a custom library on top of Solomon so you can easily link that against your server side projects and against client side ones.
We start to cover all of those one by one and so we start to show you which base objects you need to use to create your custom protocols.

Step 1: create your own custom protocol library

As we early told you, we need to create our custom protocol we will use for our purpose.
From the server's point of view, there is no logical packet or a set of informations, but just a *stream* of bytes without a logical use of those. Keeping this is mind, you can imagine how complex could be to work in such a way from a developer's point of view. Solomon library provides a big support to help you on this and so it permits you to work with a well-known formatted packets.
From Solomon library's point of view a packet is always composed by these: **header** + **body.**
This way permits developers to have all generic informations in the header and detailed request informations in the body. You have just to extend those base classes to have your own protocols and without worry about how those logical informations will be sent over the network and reassembled again to use those for your purpose.
You have to extend base class AbstractHeaderMessage to create your own header object. This class provides some generic data used internally to manage header informations needed by server.
You have to implements this methods to start with it:

```
1. byte     SizeOfHeader()
2. void     WriteIn(DataContainerWriter *writer, DataContainerWriter *body)
3. void     WriteIn(DataContainerWriter *writer, size_t lenBody)
4. void     ParseReader(DataContainerReader *reader)
```

we discuss meaning of those methods after a little introduction:
for example we need to implement a chat between two persons, so our header might looks like this:

session_token	integer	Unique identifier of a person in a chat

The default implementation of first method returns just the length of extra-data we have spoken about, but in our header we need of ***session_token*** too, so we have to reimplement this method to get the right size:

```
byte MyInternetChatHeader::SizeOfHeader()
{
    return (AbstractHeaderMessage::SizeOfHeader() + sizeof(session_token));
}
```

In the example above MyInternetChatHeader is our extended class; we implement method so it returns the base size of header plus size of our member.
In the case you need some string object, such as ConiString or ConiLargeString, you don't have to use `sizeof()` operator but GetSizeOfConiString() method instead.

Tips: you can use primitive objects such as integer and so on or you can use new types within namespace Solomon::Types and include file `Types.h`

In the same way we implement second and third methods required to put our logical members in a *__stream__* of bytes. Third method looks like these:

```
void MyInternetChatHeader::WriteIn(DataContainerWriter *writer, size_t lenBody)
{
    writer->Put(session_token);
    AbstractHeaderMessage::WriteIn(writer, lenBody);// calls after session_token
}
```

we spoke about logical protocol that we have to transform in a stream of bytes, that is what DataContainerWriter is: in fact we put in it value of our session_token and then we call the base method to ensure all base header extra-data are put inside as well.
Second method is just a wrapper on the third one and looks like this:

```
void MyInternetChatHeader::WriteIn(DataContainerWriter *writer
                                    , DataContainerWriter *body)
{
    size_t lenBody = 0;

    if (body != NULL)
        lenBody = body->GetCurLen();
    this->WriteIn(writer, lenBody); // calls the third method
}
```

Fourth method is the opposite of *WriteIn* methods: it extracts data from a stream of bytes and put in logical data; it looks like this:

```
void MyInternetChatHeader::ParseReader(DataContainerReader *reader)
{
    if (reader != NULL) {
        reader->Extract(session_token);
        AbstractHeaderMessage::ParseReader(reader); // call after session_token
    }
}
```

We extract data from the stream of bytes and put it in our member. As usual we do call to method of base class to ensure extra-data are get as well.
__Important note__: it's important to respect the order of base method calls. Because of it's a stream of bytes, changing it can cause to put or get different values.

You have seen in the base header class, informations about ID message and service; that is the way Solomon is working. Every request or handle command and header as well, know about this ID. We speak about this next in the documentation.

Similar to header, we have to create our own body object just extended the base one. We extend our class using AbstractBodyMessage base class;
You have to implements this methods to start with it:

```
1. DataContainerWriter *         GetBodyMessage(void)
2. AbstractResponseMessage *    CreateResponseObject(DataContainerReader *b)
3. byte                         IDService(void)
4. byte                         IDMessage(void)
```

as usual, in our chat implementation, we have a text to send to:

text	string	Text to send to a person

First method is automatically called by core library and it transforms logical protocol in a stream of bytes; an implementation of our method looks like this:

```
DataContainerWriter * MyInternetChatMessage_80_1::GetBodyMessage(void)
{
    DataContainerWriter *body = this->initBodyStream(text->GetSizeOfConiString());

    this->text->WriteIn(this->body);
    return (this->body);
}
```

In the example above we are using a ready-to-use string object called ConiString that has ability to manage string in conjunction with our stream of bytes. The first thing to do is to create a stream of bytes, using base class's *initBodyStream* method, passing the initial size as parameter; then we put content of *text* member in the stream and then we return the populated stream.
If your request doesn't have any data to send, you can safely returns NULL.
Memory returned by *initBodyStream* will automatically discarded by the base object you extended.

Second method is used to create a new object as a response of this request. It's implemented in a similar way of request, extend AbstractResponseMessage object. It's implementation might looks like this code:

```
AbstractResponseMessage *
MyInternetChatMessage_80_1::CreateResponseObject(DataContainerReader *b)
{
    ResponseMessage_80_1 *response = new ResponseMessage_80_1(this->IDService(),
                                                              this->IDMessage());

    if (b != NULL) response->ParseBinary(b);
    return (response);
}
```

We create a response object passing informations about ID message and service and then we call the known *ParseBinary* method to transform a stream of bytes in data members.

Implementation of third and fourth methods are just trivial: they just return a number: they identify ID message and service object is able to manage:

```
byte MyInternetChatMessage_80_1::IDService()
{
    return (80);
}

byte MyInternetChatMessage_80_1::IDMessage()
{
    return (1);
}
```

We have seen the creation of a new instance of `ResponseMessage_80_1`, so we describe this implementation that is very simple:

1. `DataContainerWriter *` `GetBodyMessage()`
2. `void` `ParseBinary(DataContainerReader *reader)`
3. `AbstractResponseMessage *` `CreateResponseObject(DataContainerReader *b)`
4. `byte` `IDService()`
5. `byte` `IDMessage()`

We already have covered all of those methods early in this document; the only consideration here is that method 3. has to return NULL since we are implementing an object that already represents a response.

At this point we are able to create our own logical protocol with all necessary requests, responses and an header to use as a glue. Now we are going to explain how to implement a server bridge to interact with server notifications and then we start with sample code.

If we take a look at `IServerBridgeProvider.h` file, we can see some classes (almost of those are irrelevant right now) we find a base class with same name of header and we have to extend it if we want to interact with notifications send from server. Methods are very clear so we don't waste time to describe all of those. The only thing is just to override those you intend to receive notifications from.

The `createXXX` methods are used to create your extended object instances the server will use internally when needed (implementation of `createHeaderMessage` is mandatory). Internally server creates a container of information for each connection established called `BaseInfoClient` used to interact with those. This slots can be also used by developer to rescue information about request sent on a socketId; for example you can extend the expiration time by setting the current time using `extendExpirationTime` method or you can programmatic change some permissions. This slot capability can of course be extended to fit your own extra info just extending and implementing that class returning an instance of your own new extended object, so you can use slots to store or extract informations in the memory speeding up the traditional processes (e.g. taking them from database).

Memory slot for a connection is available, in the handle plug-in, through `BaseInfoClient`'s infoTcpClient object only if you plan to send requests over TCP/IP protocol.

Keep in mind that informations are just local and volatile, so you cannot use in a cluster environment without get right choices and data are not stored in a persistent support. When a connection is going to expire, server will notify you through `serverWillCloseConnection` callback; in this method you can create a custom request to take action on the closing connection (e.g. a notification on a log, or clean of some connected informations) and the returned object need

to be allocated on the heap (automatically discarded by server-side garbage collector).
<u>You shouldn't implement long task in those methods or you will face slow elaboration of requests</u>.
You can also force the expiration of connection by using the method `forceExipirationTime` of class `BaseInfoClient` passing the plug-in as parameter: this method invalidates connection so it will be cleared and closed by the next garbage collector.
Be aware that server automatically forces expiration time when a connection is suddenly broken or an error occur.
Before we go further inside sample code, we start to cover how to implement an handle plugin to manage a particular request. For each request you have an handle able to manage that request and send a response if required. This handles are dynimically loaded by server at first time scanning a plugin path stored in the configuration plug-in. As usual we have to extend a Solomon base object called `IchainCommand` and your header will be similar to this:

```
class CommandRequest_XX_X : public IChainCommand {
    public:
      CommandRequest_XX_X();

      boolean              canManage(byte service, byte message);
      IChainCommand *      handler(AbstractHeaderMessage *h, ClientContainer *req);
  };
```

When the server has been started it loaded in memory all handle plug-ins putting them in memory chains; when a request will be sent, server starts to go through chains searching for an handle able to manage that request (remember the methods IDMessage and IDService) calling `canManage` method and going on next handle plugin until one of those returns true. Possible implementation might looks like this one:

```
boolean CommandRequest_XX_X::canManage(byte service, byte message)
{
   return (service == 80 && message == 1);
}
```

In the example above, whenever server receives a request of service 80 and message 1, the handle with that implementation will be selected to elaborate the request calling method `canManage and` passing the header and other informations extracted by server.
`ClientContainer` is populated with all related informations like stream request, connection informations and so on:

```
typedef struct ClientContainer {
    TInfoConn                   *client;
    DataContainerReader         *buffer;
    AbstractServerGame          *udpServer;
    AbstractTcpServerGame       *tcpServer;
    TTypeOfRequest              typeOfRequest;
    BaseInfoClient              *infoTcpClient;
    ClientContainer();
} ClientContainer;
```

Important note: to identify your service you can only use a range from 1 to 254. Service valued of 255 is reserved for internal use and you shouldn't have to use it for your purpose.
A quick explanation about class members:

- The object client contains information about IP and port used by a client.
- The object buffer represents the stream of bytes (just the body message).
- Objects udpServer and tcpServer represent the instance for interact with servers.
- The object typeOfRequest will always contain the value `typeOfEndUserRequest`.
- The object infoTcpClient will be NULL if request comes over UDP/IP, but will be populated with with the well-known `BaseInfoClient` object if request comes over TCP/IP.

A possible implementation of handler could be:

```
IChainCommand * CommandRequest_XX_X::handler (AbstractHeaderMessage *header,
                                             ClientContainer *client)
{
  MyInternetChatHeader     *chatHeader = (MyInternetChatHeader *) header;
  ISpecializedPlugin       *plugin = client->udpServer->getPlugin () ;
  BodyMessage_80_1         *bodyRequest = NULL ;
  ResponseMessage_80_1     *answer = NULL;
  string                   login_session = client->tcpServer->createUniqueSessionToken () ;

  bodyRequest = new BodyMessage_80_1(client->buffer);
  answer = new ResponseMessage_80_1(header->GetIDService (), header->GetIDMessage () ) ;
  //
  // setting other stuff inside response or header
  if  (client->infoTcpClient != NULL)  // request over TCP/IP
       client->tcpServer->sendMessage(client->infoTcpClient->getSocketId ()
                                      , chatHeader, answer) ;
  delete  (bodyRequest) ;
  delete  (answer) ;
}
```

You don't have to worry about memory management of parameter members, but just worry about all dynamic objects you create inside this method.
Important: when you have compiled all necessary command plug-ins, say service ID 10 and 20 and message ID 1 and 2 (for all service Ids), you now need to create a structured directory in path where server will extract and load your own plug-ins, as follow:

- /home/myserver/plugins/ (path specified in configuration plug-in)
  - 10 (directory)
    - libCommand_10_1.so (compiled handle)
    - libCommand_10_2.so (compiled handle)
  - 20 (directory)
    - libCommand_20_1.so (compiled handle)
    - libCommand_20_2.so (compiled handle)

Now we have covered all necessary objects, we can start implementing a real server using some examples. Let's start with those...

Example 1: how to create a concrete server

It's now easy to create a real server with few instructions!
Let me show you this code (of course you need to link Solomon and your own libraries to project).

Necessary include files:

```
#include "GameServerClass.h"
#include "PluginLoader.h"
#include "mybridgeprovider.h"
```

The first include represents a base object that provide all necessary to load handle plug-ins, to create configuration plug-in bridge and start server.
The second represents a base object to manage all information in the configuration plug-in.
The third one is our custom provider object used to interact with some notifications from our server.

```
int main(int argc, char *argv[])
{
    // it requires one parameter: path and .so filename
    if (argc != 2) {
        return (EXIT_FAILURE);
    }
    PluginLoader          loader(argv[1]);
    ISpecializedPlugin    *plug = loader.getPlugin();
    MyBridgeProvider      bridge;
    GameServerClass       *server = NULL;
    //
    // check if plugin has been successfully loaded:
    if (plug != NULL) {
        bridge.setPlugin(plug);
        server = new GameServerClass(&bridge, 0, 0);
        server->StartServerListener();
        printf("Error starting server: %d", errno);
    }
    else
        printf("\nError loading plugin: %s\n", argv[1]);
    return (EXIT_SUCCESS);
}
```

First we ensure about the presence of parameter, then we create a PluginLoader passing such plugin path and we try to get a plug-in instance using **getPlugin**().
If all went well we create a new GameServerClass instance. Notice second and third parameters can be zero or equal to a service and message number to create a direct short-cut to a most often request. To keep server up and running we have just to call **StartServerListener**() method.
That's all guys!!!

Now we continue to explain the use of libraries on the server side's point of view.

Step 2: client side implementation

We have created our own library on top of Solomon, but that implementation has just the server side objects and our own protocol implementation. We have to do an extra step to create support for client side communications.
As we have seen for server side, we have `IGameClientBridgeProvider.h` that represent the bridge used on client side to get notifications from the embedded client. As usual we have to extend this class to interact with embedded client. Many of these methods are similar to server side and other are simple to understand. The only one worth to describe here is:

```
boolean clientDidHandleServerRequest(const AbstractHeaderMessage *header,
                                     DataContainerReader *body)
```

This method is automatically called by client when it receives something from server: if the notification is handled by you in this function (because information represents a notification from someone) you should return TRUE so client discard that; otherwise you should return FALSE and so the server falls in wait for a response (FALSE indicates you have sent a request and so you are waiting for a response since both notifications and responses come from embedded client). So go back to our chat, a possible implementation looks like this:

```
boolean MyClientBridge::clientDidHandleServerRequest(
            const AbstractHeaderMessage *header, DataContainerReader *binaryBody)
{
  bool   handled = true;

  if (header->GetIDService() == 80 && header->GetIDMessage() == 1) {
     ResponseMessage_80_1 mes(0, 0); //text sent by a chatter: able to manage it
  }
  else {
     handled = false;
  }
  return (handled);
}
```

In our chat, when client receives a message of 80_1 (text from other chatter) we know it is a notification, so we are able to manage that and so we can safely return TRUE. If we receive another message (for example we have sent a request to server to fetch some informations, we know it is not a notification sent from someone else and so we cannot handle that (this is the way Solomon manages synchronous requests with an expiration time-out).
We are half way from using our client implementation to communicate to server...next step is just to create an instance of `GameClientClass` that permits to establish a connection with server on both UDP/IP and TCP/IP protocols. **For each instance of above class, you need to periodically update expiration time of connection (using *`extendExpirationTime`* `within your dedicated handle`) to avoid connection being closed for inactivity by server**. It also provides methods to easily send through those protocols with synchronous and asynchronous capabilities and send files as well. In our example we know request 80_1 will be always sent using asynchronous methods and so we have to manage response as described above.
If you plan to create requests meant to be sent in synchronous ways you always have to send them

using synchronous methods and don't handle in the above implementation (returning FALSE). When we create an instance of `GameClientClass`, we construct it passing a real instance of our client bridge extended class plus informations about remote server and local IP and port. Then we can start server using StartEmbeddedServer() method and wait until callback *`clientConnectedToServer()`* has been called. When connection has been established, we receive from server our client ticket (socked id that represent our connection on server) ; this is important if we plan to send requests over UDP/IP (remember server responds to client always through TCP/IP so we need to know socket id; this limitation is not present when we send through TCP/IP protocol). At this point, when *`clientConnectedToServer()`* has been called, we are able to send and receive requests or files to/from server. Send a file is always performed by AsyncSendFileStream() method and it requires an header, a unique identifier (you can use it to build your own vector of files sent, so you can always identify a file in the vector) and of course a path to a file.
When a server has successfully received a file from client, it creates a new request using the original header, passed as parameter, and an istance of `AttachedLinkMessage` class as a body and send it to a handle able to manage this request. As for other bodies, you create an instance of it and then access to link where file is available for download.
AsyncSendFileStream() always start the procedure on separate thread and multiple call to this method are queued waiting for the end of current transfer. So if we send three or four files, we know they start some threads and then they will be executed one by one in a not specified order: how do we know which files are waiting for or sent to?
As we know we have our instance of client bridge provider that also provides notifications for file:

```
void clientWillStartTransfer(const AbstractHeaderMessage *header, u_integer id
                                    , const buffer_pointer fileName);

void clientDidFinishTransfer(const AbstractHeaderMessage *header, u_integer id
                             , const buffer_pointer fileName, TFileUploadResult sent);
```

when a file is going to be sent, the first callback will be risen with all informations provided calling AsyncSendFileStream(); in the same way you will be notified when and how a transfer file has been completed. When you send a transfer file request is possible that server can be busy and so it is not able to receive file in that moment (a queued request has an expiration time of 5 seconds before giving up) or you don't have permission to send it: so you always have to check value of `TFileUploadResult` to know if a transfer has been completed successfully or not.
When *sent* is equal to fileUploadSuccess, you can rely just on *id* of file, since the success is returned by server side and so *header* and *fileName* are both NULL.
When the garbage collector on server-side decides to close a connection (because inactivity time has been reached) before doing this, a notification of close will be sent by server to client and then it will rise this event:

```
void clientDisconnectedFromServer();
```

Above event will be even called if someone calls StopEmbeddedServer() method somewhere.
Important note: sending file doesn't provide a way to avoid inactivity of connection on server-side (we have spoken about this early speaking of memory slots), so you have to provide your own dedicated handle and periodically send request to server that will be managed by that handle (inside you have just to find memory slot related to that connection and update current time).