Solomon

cross-platform network framework - version 2.6

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 communicates to and complexity behind real-time server capabilities it provides.
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 text message 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...some of those like upload permission can be changed on the fly in your commands to fix your needs (e.g. a user can pay a monthly fee to have this enabled).
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 starts, it falls listening to port you have specified (set 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 plug-in details). When this happens, 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 handles and requests to update connection time. We will explain it further in the documentation.
Whereas server side can only actually be created on *nix machines (due to some *nix features), the client side of library has 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 name with .so extension and the latter 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>0</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 the 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 during a connection to a 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 be cleaned and closed.
- <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 listening 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 needs 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 changed during the execution tailored on each user connection) and must have a value of 1 to enable permission; 0 otherwise. Default value set to zero is the best way to avoid simple attack performed over telnet o similar

applications.

- <max_bytes_file_upload> the value of this property indicates default size of files a server can accept as upload (it can be changed during the execution 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 clients can download uploaded files. 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 the server has to create. A worker thread has its own thread pool to manage and consume the main queue. Generally the number of worker threads to use is based on number of CPUs.
- <number_pool_threads> indicates how many threads a pool has per worker thread. This value cannot be less then 2 per worker thread otherwise server won't serve requests.
- <trusted_passwd> as the name suggest, this is an internal password you can use just to trust some operations that need a little bit 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 sources (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 need to rename configuration file like: "**lib_odbc.so.config**". Generally we pass path and configuration plug-in file as parameter for the 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 for our new project.
From the server point of view, there is no logical packet or a set of information, but just a *stream* of bytes without a logical use of those. Keeping this in mind, you can imagine how complex it could be to work in such a way from a developer 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.
For Solomon library a packet is always composed by a **header** and a **body.**
This permits developers to have all generic information in the header and detailed request informations in the body. You have just to extend base Solomon objects to have your own protocols and without worry about how information will be sent over the network and reassembled again to be used in your source code.
You have to extend base class AbstractHeaderMessage to create your own header object. This class provides some generic data used internally to manage header information needed by server.
You have to implements these methods to start with it:

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

we discuss meaning of those methods after a little introduction:
for example we need to implement a chat between two people, so our header 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 ***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 reimplement method so it returns the base size of header plus size of our new member.
In the case you need some string objects, such as SolomSmallString or SolomLargeString, you don't have to use `sizeof()` operator but GetSizeOfSolomString() 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 header file `Types.h`

In the same way we implement the other required methods to put content of variables in a *stream* of bytes. Second method looks like this:

```
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 for: in fact we put in it values of our **session_token** and then we call the base method to ensure all base header extra-data are put in the stream as well.
Third method is the opposite of *WriteIn* method: it extracts data from a stream of bytes and put those in variables; 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 since it is a stream of bytes.

You have seen in the base header class, informations about ID message and service; that is the way Solomon is working. Any request, 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 extending the base one. We extend our class using AbstractBodyMessage base class;
You have to implement these 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 be sent:

text	string	Text to send to a person

First method is automatically called by core library and 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->GetSizeOfSolomString());

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

In the example above we are using a ready-to-use string object called SolomLargeString 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 the 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 be sent, you can safely returns NULL.
Memory returned by *initBodyStream* will automatically be discarded by the base object you have just extended.
Purpose of the second method is to create a response for this request. It's implemented in a similar way we did for the request, extending AbstractResponseMessage object. It's implementation looks like this:

```
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 information about ID message and service and then we call the well-known *ParseBinary* method to transform a stream of bytes in data members.

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

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

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

We have so far 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 about the method 3. that 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 be used as 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 see some classes (almost of those are irrelevant right now) useful for the server activities; for the time being we concentrate on IserverBridgeProvider class that can be extended if we want to interact with notifications sent by a server. Methods are very clear so we don't waste time to describe all of them. Just override those for which you intend to receive notifications from the server.
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 established connection called `BaseInfoClient` used to interact with those later on. These slots can also be used by developers to retrieve information about requests sent on a socketId; for example you can extend the expiration time setting the current time using *`extendExpirationTime`* method or you can change some permissions just during the execution. Because the slot is sticking to the connection and is available in the pipeline during the elaboration of a request sent over that, the container can be used to store your own extra information just extending it and override *`createMemorySlot`* of `IserverBridgeProvider` to return an instance of it, so you can use slots to read/write information in the memory speeding up the traditional process (e.g. taking them from database).
A memory slot is available, in the handle plug-in, through `BaseInfoClient`'s infoTcpClient object only if requests are coming from TCP/IP connections.
Keep in mind that information are just local and volatile, so you cannot use those in a cluster environment without a plan for this. When a connection is going to expire, server notifies you through *`serverWillCloseConnection`* callback; in this method you can create a custom request to

take an action on the closing connection (e.g. an entry on a log file, or cleaning up information) and the returned object need to be allocated on the heap (automatically discarded by the server-side garbage collector).
You shouldn't implement long task in those methods or you will face slow elaboration of requests.
You can also force the expiration for a connection by using the method *forceExipirationTime* of BaseInfoClient passing the plug-in as parameter: this method invalidates the connection so it will be freed and closed by the next garbage collector.
Be aware that server automatically forces expiration time when a connection is suddenly broken or an error occurs.
Before we go further to the sample code, we cover how to implement a handle plugin to manage a particular request. For each request you have a handle able to manage that request and send a response if required. These handles are dynamically loaded by server at first time scanning a plug-in path stored in the configuration file. 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 loads in memory all the handle plug-ins putting them in memory chains; when a request has been sent, server goes through the chains searching for an handle able to manage that request (remember the methods *IDMessage* and *IDService*) calling *canManage* method and going on next handle plug-in until one of those returns true. Possible implementation may look like this:

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

In the example above, whenever the server receives a request of service 80 and message 1, the handle with that implementation will elaborate the request calling handler.
ClientContainer is populated with all related information like stream request, connection information 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 mustn't be used 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 instances to interact with servers.
- The object typeOfRequest will always contain the value typeOfEndUserRequest.
- The object infoTcpClient will be NULL if a request comes over UDP/IP, but it will be populated 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;
  buffer_pointer          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);
  delete (login_session); // createUniqueSessionToken() allocates new memory
}
```

You don't have to worry about memory management of parameters, but just worry about all dynamic objects you create inside this method.
Important: after you have compiled all necessary command plug-ins, say service ID 10 and 20 and message ID 1 and 2 (for both 10 and 20), 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 that we have provided all required knowledges, 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 stuff to load handle plug-ins, to create configuration plug-in bridge and to start the server.
The second one 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
    // e.g. argv[1] = "/home/server/libConfigBase.so"
    if (argc != 2) {
        return (EXIT_FAILURE);
    }
    PluginLoader          loader(argv[1]);
    ISpecializedPlugin    *plug = loader.getPlugin();
    MyBridgeProvider      bridge;
    GameServerClass       *server = NULL;
    //
    // checks if plug-in 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 just have to call **StartServerListener**() method.
That's all guys!!! With few line of code you provided a professional, real-time server!

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 represents the bridge used on client side to get notifications from the embedded server running on the client. As usual we have to extend this class to interact with embedded server. Many of these methods are similar to server side ones and other are simple to understand. The only one worth being described 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 server discards that; otherwise you should return **FALSE** and then server will fall in wait for a response (**FALSE** indicates you have sent a request and so you are waiting for a response since both notification and response come from embedded server). So going 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 then we can safely return **TRUE**. If we receive another message (for example we have sent a request to server to fetch some information, we know that it is not a notification sent from someone else thus 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 requests thru 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 information about remote server and local IP and port. Then we can start the embedded server using `StartEmbeddedServer` method and wait until callback `clientConnectedToServer` has been called. When connection has been established, we receive from the server our client ticket (socked ID representing 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 requires a header, a unique identifier (you can use it to build your own vector of sent files, 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 the embedded client, it creates a new request composed by the original header and an instance of `AttachedLinkMessage` as body and pushed in the worker queue. Like other bodies, you create an instance of it and then access to the link where file is available for download.
`AsyncSendFileStream` always starts the procedure on a separate thread and several calls to this method are queued waiting for the end of ongoing transfer. So if we send three or four files, we know they start some threads and then they will be executed one by one:
how do we know which files are waiting for or are sent to?
As we know we have our instance of client bridge 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 information provided calling `AsyncSendFileStream`; in the same way you will be notified when and how a transfer has been completed. When you send an upload file request it 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 it gives 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, *header* and *fileName* are both set to 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 the embedded server and then it will rise this event:

```
void clientDisconnectedFromServer();
```

Above event will even be 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).

Extra Information

This guide is part of the shipped libraries available at this repository
For detailed information about objects please refer to the header files.
This library is not an open-source project. It can be used for personal and commercial use without any limitation.

If you want to see a real professional usage of Solomon library or just contribute with a donation, check out Softair Real Fight.
For suggestions and improvements please contact us using the form in the above link.