

ExtendedVisualOtp (EVO) The User's Guide Version 1.5

Ing. Ivan Carmenates García

Ivanco Software Company in association with SPI Team

December 7 2010



Index

Abstract	5
Introduction	6
Content	13
Functionalities of the <i>ExtendedVisualOtp.dll</i> library.....	13
ConvertToCSharp:.....	13
• FromErlangObject(<i>Otp.Erlang.Object _object</i>)	13
• FromErlangReply(<i>Otp.Erlang.Object _replyObject</i>).....	13
ConvertToErlang:	13
• ToErlangObject(<i>object _object</i>)	13
• ToErlangObjectFromString(<i>string _object</i>).....	13
ErlangServerInterface:	13
• AutoTrace	13
• Connect()	13
• ErlangCookie	13
• Disconnect().....	13
• IsConnected	13
• OnDisconnected(<i>IServerReply _reason</i>)	14
• OnReceive(<i>IServerReply _reply</i>).....	14
• PingTimeOut	14
• RemoteAddress	14
• RemoteNodeName	14
• RemoteProcessName	14
• SetOwnerForm(<i>Form _ownerForm</i>)	14
• SentToErlangServerRequest(<i>ErlangServerRequest _erlangServerRequest, IServerReply _reply</i>).....	14
• ServerPort.....	14
• StartEVOTestApplication().....	14
• UseShortNames	14
• AllowReceiveClientImage	14
ErlangServerRequest:	15
• AbortAllRequests().....	15
• AbortAsyncRequests()	15

• AbortNormalRequests().....	15
• AbortRequest(<i>IRequestInfo _reqInfo</i>)	15
• Description.....	15
• Message.....	15
• ErlangServerInterface	15
• OnReceive(<i>IServerReply _reply</i>).....	15
• Request(<i>string _description, object _message</i>)	15
• RequestAsync(<i>string _description, object _message</i>)	15
• Request(<i>object _message</i>)	15
• RequestAsync(<i>object _message</i>)	15
• SetOwnerForm(<i>Form _ownerForm</i>)	16
• PublishClientImage().....	16
IRequestInfo:	16
• AbortRequest()	16
• Description.....	16
• Kind	16
• WaitForReply().....	16
• WaitForReply(<i>int _timeout</i>).....	16
• WasAnswered	16
• WasMade.....	16
IServerReply:.....	16
• CSharpReply.....	16
• Description.....	16
• ErlangReply	16
• RequestInfo	16
RequestKind.....	17
• Async.....	17
• Normal	17
• ServerSent.....	17
The <i>evo_template</i> template.....	18
The <i>evo_template</i> has many files listed at following:.....	18
• 1.install.cmd	18

• 1.start.cmd.....	18
• 2.service_install-start.cmd	18
• 2.service_stop.cmd.....	18
• 2.service_uninstall.cmd	18
• compile.cmd	18
• config.cmd	18
• Folder [<i>beam</i>]	18
• appname.app.....	18
• appname.beam.....	18
• appname_debug_module.beam	18
• appname_main_interface.beam	18
• appname_main_supervisor.beam.....	18
• Folder [<i>sources</i>]	18
• appname_db_module.erl.....	19
• appname_db_server.erl	19
• appname_request_handler.erl.....	19
• user_default.erl	19
The Big Example	22
Annexes.....	26
Conclusions	29

Index of Illustrations

Illustration 1. Example #1, client – server application interaction.....	6
Illustration 2. Visual configuration for the <i>ErlangServerInterface</i> and <i>ErlangServerRequest</i> components.....	7
Illustration 3. Event <i>Request-OnReceive</i>	8
Illustration 4. Erlang server application the <i>evo_template</i>	19
Illustration 5. Client 1.....	26
Illustration 6. Client 2.....	26
Illustration 7. Client 1 and Client 2 interacting through the server application.....	27
Illustration 8. The EVO Test Application	28

Index of Tables

Table 1: Kind of messages that can trigger the <i>Interface-OnReceive</i> event.....	10
--	----

Abstract

The ExtendedVisualOtp (EVO) is a component specialized in client-server communication, implemented in two programming languages, C# of Microsoft and Erlang of Ericsson, oriented to the development of real time applications, where the server applications are written in Erlang and the client applications in .Net. The main concept of this component is the development of client-server applications where the productivity factor is highest and the cost of the production is quite low. Ideal for small companies with few experience in that kind of applications, which want to insert into this so wide world that is the interaction and exchange of information through the network, in a fast and powerful manner.

Introduction

The ExtendedVisualOtp (EVO) is a component specialized in client-server communication, implemented in two programming languages, C# of Microsoft and Erlang of Ericsson, oriented to the development of real time applications, where the server applications are written in Erlang and the client applications in .Net. The main concept of this component is the development of client-server applications where the productivity factor is highest and the cost of the production is quite low. Ideal for small companies with few experience in that kind of applications, which want to insert into this so wide world that is the interaction and exchange of information through the network, in a fast and powerful manner.

It can be illustrated through a simple example:

Example #1: Imagine you want to develop a small client application, so you want to make a request “know my payment” to a server application and to be notified about the answer generated by that request.

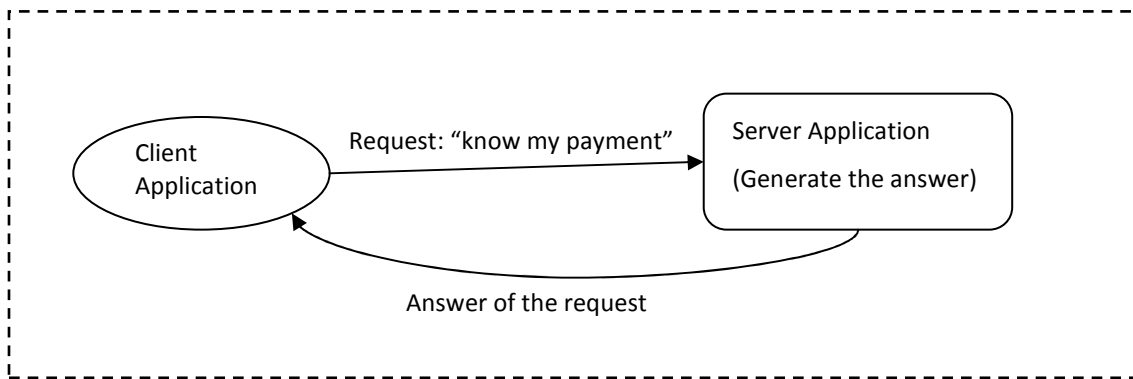


Illustration 1. Example #1, client – server application interaction

Develop this simple example in any programming language without the usage of a helper component, could represent a lot of work. However, using the *ExtendedVisualOtp.dll* library for the development of client applications and the *evo_template* template to write server applications becomes from a large and restless work to a simple and comfortable one.

Answer for the example #1 using EVO

At the client application, after a few and simple visual configurations for the *ErlangServerInterface* and *ErlangServerRequest* components of the *ExtendedVisualOtp.dll* library, you should write:

```
// to connect to the server
erlangServerInterface1.Connect();
```

// to make a request

```
erlangServerRequest1.Request("Description of the request", object _message);
```

or

```
erlangServerRequest1.Request(object _message);
```

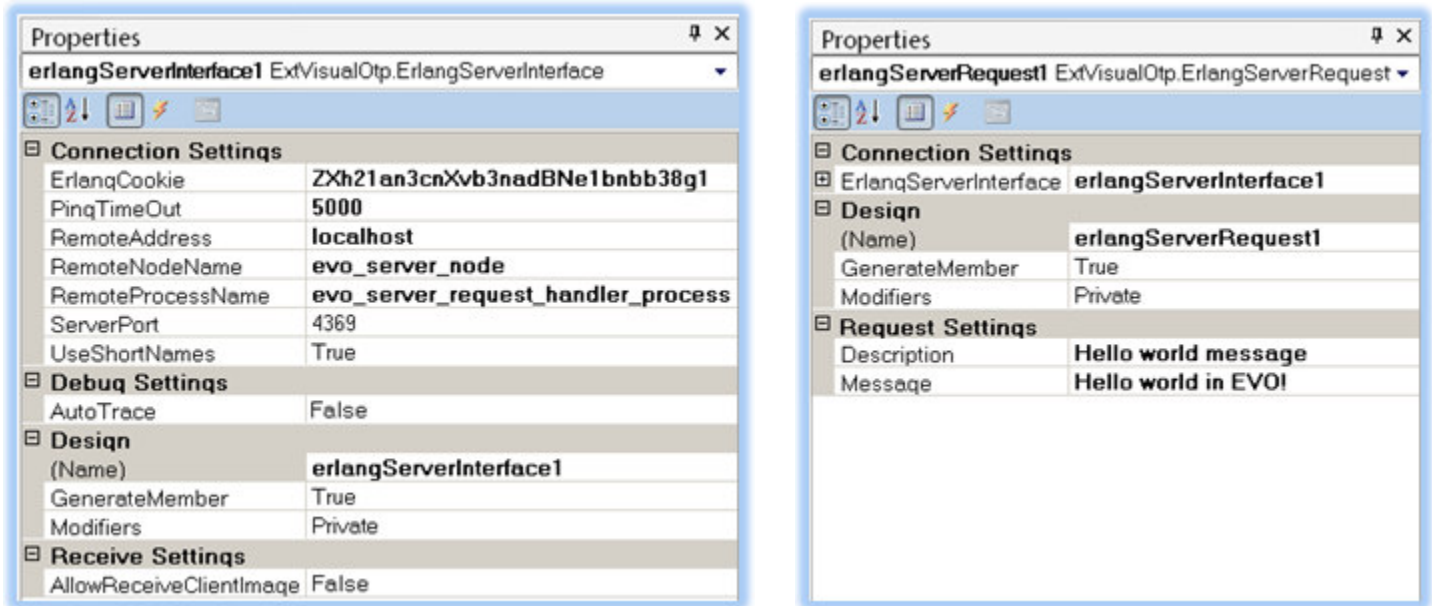


Illustration 2. Visual configuration for the *ErlangServerInterface* and *ErlangServerRequest* components

NOTE:

- ❖ *Request-OnReceive* means the *OnReceive* event of the *ErlangServerRequest* component.
- ❖ *Interface-OnReceive* means the *OnReceive* event of the *ErlangServerInterface* component.

// to receive the answers

The answers of the made requests could be caught of *Async* way through the *Request-OnReceive* event, or could be caught of *Sync* way using the function *WaitForReply()* immediately after the request, example for the *Sync* way:

```
IRequestInfo reqinfo = erlangServerRequest1.Request("knows the salary", new object[] {  
    " 'know_my_payment' ", "Ivan Carmenates García"});  
IServerReply reply = reqinfo.WaitForReply();  
MessageBox.Show(reply.CSharpReply.ToString());
```

reqinfo is an object of the class or interface *IRequestInfo*. It contains the information of the made request, allowing to do some stuffs with it, such as, block the program, to wait for the answer, *reqinfo.WaitForReply()*, or abort the made request, *reqinfo.AbortRequest()*.

reply is an object of the class or interface *IServerReply* that contains the information to treat the answer sent by the server.

In case that you wish to receive the answers using the *Async* way, you must use the *Request-OnReceive* event without the usage of the function *WaitForReply()*, example for the *Async* way:

```
IRequestInfo reqinfo = erlangServerRequest1.Request("knows the salary", new object[] {  
    "know_my_payment", "Ivan Carmenates García"});
```

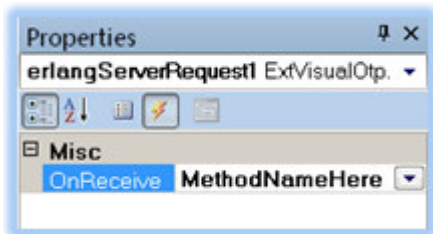


Illustration 3. Event *Request-OnReceive*

And to receive the answer, you should write something like this:

```
OnReceiveErlangServerRequestMethodNameHere(IServerReply _reply)  
{  
    switch(_reply.Description) {  
        case "knows the salary":  
            MessageBox.Show(_reply.CSharpReply.ToString());  
            break;  
    }  
}
```

The same implementation, works for receive all the answers of all requests that you make, you just have to add a new clause to the switch sentence with a new description, a description per request you make.

So, to write server applications, you must edit the server template, *evo_template* and in the file called *evo_request_handler.erl*, located in the *sources* folder, about the line 28, part: *[USER FUNCTIONS TO MODIFY]* write your code. For the case of example #1, could be:

```
{{know_my_payment, Name}, Pid, RequestInfo}->  
    Reply = evo_db_module:get_payment (Name), %% this function is implemented in other module.  
    Pid ! {reply, Reply, RequestInfo};
```

As it seems, is easy to write client-server applications in EVO. If you're wandering, how it is possible? The answer is as simple as it sound, that's all! The EVO component, in its total integrity, both, the template *evo_template* to develop server applications and the *ExtendedVisualOtp.dll* library to develop client

applications, it handles for you all the dirty work. You just have to abstract your mind to the existence of a fast and efficient postman, which you call to deliver your letter or messages to a given destination, in this case the server. The server can send if you wish, the message to all other people in the “world”, or just to one person, if it is programmed of that way, acting as intermediary. An example, if you wish to send “directly” a message to another client, the EVO component has a functionality, where once your client is connected to the server, each time that other clients get connected or disconnected, the server sends to your client a notification message of new client connected, `{new_member, Pid}`, or client disconnected, `{crashed_member, Pid}`, and the ID or address `Pid`. With this ID, you can send to those clients through the server a message, simple as it may seem! Making a request like this:

```
IRequestInfo reqinfo = erlangServerRequest1.Request(new object[] {  
    “ ‘send_to’ ”, ClientPid, Message});
```

Notice here the usage of the other way to make requests to the server, where is not necessary to specify a description for the request, it is the case of the call to the function `Request(object _message)`. The description for this request is obtained by the first element inside the message, if the message is an array or a list of elements and if its first element is an Erlang atom, or if the message is just one element and it is an Erlang atom. For the case before, note that the first element of the array that you send as message is the Erlang atom “ ‘send_to’ ”. This atom is the description for the request and the message at the same time. In case that the first or unique element of the message isn’t an Erlang atom, the description for the request will be set to the Erlang atom “ ‘no_description’ ”.

How to catch the message `{new_member, Pid}` / `{crashed_member, Pid}`?

Through the *Interface-OnReceive* event, example:

```
OnReceiveErlangServerInterfaceMethod(IServerReply _reply)  
{  
    If (_reply.Description == “new_member”) {  
        erlangServerRequest1.Request(new object[] {  
            “ ‘send_to’ ”, _reply.ErlangReply as Otp.Erlang.Pid, “Are you new?”});  
        }  
    }  
}
```

With this, your client sends to each client that got connected to the server, the message: “Are you new?”

`_reply.ErlangReply`, it’s the message, that is, the ID of the client recently connected to the server.

On the server application, in the template *evo_template* to develop server applications, in the file, *evo_request_handler.erl*, you should write something like this:

%% The 'send_to' message.

```
{{send_to, WhoPid, Message}, Pid, RequestInfo = {Description, _, _, _}}->  
    send_to(WhoPid, Description, Message),  
    Pid ! {reply, "The message was successfully sent.", RequestInfo};
```

The function *send_to(ClientPid, Description, Message)* is a function that comes within the component EVO, in the template for server applications, *evo_template*, with which you can send messages to any client that got connected to the server, specifying its id (*Pid*) in the connection.

The *Interface-OnReceive* event is fired each time that any message sent by the server, is received on the client. And the *Request-OnReceive* event is fired only when a message received, is the answer of a request made by it. So if you want to get low level messages, you must use the *Interface-OnReceive* event.

Table 1: Kind of messages that can trigger the *Interface-OnReceive* event

Description	Message	Explanation	Fires the Request-OnReceive event?
<i>new_member</i>	<i>Pid</i>	Means that a new client got connected to the server, <i>Pid</i> is the ID of the new client connected.	No
<i>crashed_member</i>	<i>Pid</i>	Means that a client got disconnected from the server, <i>Pid</i> is the ID of the disconnected client.	No
<i>no_description</i>	Message	When is not possible to retrieve the description for the request made by the client using the function <i>Request(object _mensaje)</i> or <i>RequestAsync(object _mensaje)</i> .	Yes
Description	Message	An answer of a request, where its information can be retrieve successfully.	Yes
Description	Message	All messages sent by the server of voluntary way, (messages of <i>ServerSent</i> kind).	No

So, how is possible to redirect the messages that only trigger the *Interface-OnReceive* event to a specify *Request-OnReceive* event?

Imagine that you already have an implementation for the *Request-OnReceive* event and that you already receive messages of the following manner:

```

void ErlangServerRequestOnReceiveMethod(IServerReply _reply)
{
    swicth (_reply.Description) {
        case "get_image":
            this.paintBox.Image = _reply.CSharpReply as Image;
            break;
    }
}

```

And usually you make requests to the server of the following way:

```
erlangServerRequest1.Request(new object[] { " 'get_image' ", Name});
```

The answer of that request shall trigger the *Request-OnReceive* event implemented before, if in the server application was wrote something like this:

Case 1

```

{{get_image, Name}, Pid, RequestInfo}->
    Reply = appname_db_module:get_picture(Name),
    Pid ! {reply, Reply, RequestInfo}; % This sends the answer to the client who made the request.

```

But, what would happen if we write in the server application, something like this?

Case 2

```

{{get_image, Name}, Pid, RequestInfo}->
    Reply = appname_db_module:get_picture (Name),
    send_to_all(get_image, Reply); % This sends the image to all the clients.

```

Well, in this case, the answer of the request can't fire the *Request-OnReceive* event implemented before, because once you use the function **send_to_all(Description, Message)**, all the information, *RequestInfo*, regards the request, is lost. This information travels from the client to the server and is returned, without any change, to the client, like in the first case, **Pid ! {reply, Reply, RequestInfo}**, where *Pid* is the ID of the client in the connection, *Reply* is the answer for the client and *RequestInfo* is the request's information sent by the client. With this request's information, the client, once receives the answer, knows to which *ErlangServerRequest* handler belongs.

All the messages sent by the server are received on the client in first instance through the *Interface-OnReceive* event. So is possible to re-direct the messages sent by the server to a given *ErlangServerRequest* handler making possible to trigger the *Request-OnReceive* event, using the function of the *ErlangServerInterface*

component, *SendToErlangServerRequest(ErlangServerRequest _erlangServerRequest, IServerReply _reply)*, example:

```
void ErlangServerInterfaceOnReceiveMethod(IServerReply _reply)
{
    if (_reply.Description == "get_image") {
        erlangServerInterface1.SendToErlangServerRequest(erlangServerRequest1, _reply);
    }
}
```

This re-direct the messages sent by the server, using the function *send_to_all(Description, Message)* to a specified *ErlangServerRequest* handler, in this case 'erlangServerRequest1' object, in order to treat that answer as if it was the answer of a request made by the same component.

Sending objects

In EVO you can send objects as easy as you pass an object through parameter, just put your object as a request parameter, example:

```
class MyObject {
    string name;
    int id;
    void get_name() {
        return this.name;
    }
    void set_name(string _name) {
        this.name = _name;
    }
}

MyObject obj = new MyObject();
obj.set_name("Ivan Carmenates García");
erlangServerRequest1.Request(new object[] { " 'send_my_object' ", obj});
```

To receive it, is the same mechanism of always but casting with the class, example:

```
... MessageBox.Show((_reply.CSharpReply as MyObject).get_name()); ...
```

The same happens with any serializable object, example:

```
erlangServerRequest1.Request(new object[] { " 'send_image' ", new Image.FromFile("c:\\test.jpg")});
... if (_reply.Description == "send_image") {pictureBox1.Image = _reply.CSharpReply as Image;} ...
```

Next shall be explained all the class and interfaces of the EVO component.

Content

Functionalities of the *ExtendedVisualOtp.dll* library

ConvertToCSharp: It's a utilitarian class to convert *Otp.Erlang.Object* objects to *CSharp* objects.

- **FromErlangObject(*Otp.Erlang.Object* _object)**

Converts *Otp.Erlang.Object* terms in *CSharp* terms, example:

```
Otp.Erlang.List: [list]      in  CSharp List<object>
Otp.Erlang.Tuple: {tuple}   in  CSharp object[]
Otp.Erlang.Atom: 'atom'     in  CSharp string: " 'atom' "
```

An object *Otp.Erlang.List*: [1, {name, "Ivan"}] is converted to a *CSharp* object: *List<object>* = *new List<object>().AddRange(new object[] {1, new object[] { " 'name' ", "Ivan" }})*.

- **FromErlangReply(*Otp.Erlang.Object* _replyObject)**

Converts a reply or message sent by the server to a *CSharp* object of the class or interface *IServerReply*.

ConvertToErlang: It's a utilitarian class to convert *CSharp* objects to *Otp.Erlang.Object* objects.

- **ToErlangObject(object _object)**

Converts *CSharp* objects to *Otp.Erlang.Object* objects, example:

```
CSharp List<object>      in  Otp.Erlang.List: [list]
CSharp object[]          in  Otp.Erlang.Tuple: {tuple}
CSharp string: " 'atom' " in  Otp.Erlang.Atom: 'atom'
```

- **ToErlangObjectFromString(string _object)**

Creates from a *CSharp* string an *Otp.Erlang.Object* object, example:

"{name, \"Ivan\", 'age', 26}" is converted into an Erlang term {name, "Ivan", age, 26}.

ErlangServerInterface: It's a component to configure the connection parameters in order to work as interface to Erlang server.

- **AutoTrace**

Establishes if the handshake with the server must be displayed in the debug console.

- **Connect()**

Tries to connect to the server.

- **ErlangCookie**

Sets the cookie to connect to server's node.

- **Disconnect()**

Disconnects from the server.

- **IsConnected**

Gets the server connection status.

- **OnDisconnected(*IServerReply _reason*)**

Event that fires when server get disconnected.

- **OnReceive(*IServerReply _reply*)**

Event that fires when a message arrives from the server, the parameter is the message.

- **PingTimeOut**

Sets the timeout to establish the connection with the server.

- **RemoteAddress**

Sets the server address.

- **RemoteNodeName**

Sets the server node name.

- **RemoteProcessName**

Sets the server process name which handles the clients.

- **SetOwnerForm(*Form _ownerForm*)**

Sets the owner form, this makes possible the validation of *thread cross operation error* without the usage of invoke method in any of *OnReceive* and *OnDisconnected* events.

- **SentToErlangServerRequest(*ErlangServerRequest _erlangServerRequest, IServerReply _reply*)**

Allows redirect the global messages sent by the server to a given *ErlangServerRequest* interface.

- **ServerPort**

Sets the daemon port of Erlang server.

- **StartEVOTestApplication()**

Starts the EVO Test Application.

- **UseShortNames**

Establishes if short names format shall be used for the connection. Example of short name format for server address: *pc5*, large name format: *pc5.evo.cu*.

- **AllowReceiveClientImage**

Property that sets if it's possible to receive client's images sent by another client using the function *PublishClientImage()* of the *ErlangServerRequest* component. Disabled by default because security problems, this functionality is only for programming aims.

ErlangServerRequest: It is a component in order to make requests to the server, using the component *ErlangServerInterface* as connection interface.

- **AbortAllRequests()**
Aborts all the requests made to the server.
- **AbortAsyncRequests()**
Aborts all the requests of *Async* kind made to the server.
- **AbortNormalRequests()**
Aborts all the requests of *Normal* kind made to the server.
- **AbortRequest(*IRequestInfo _reqInfo*)**
Aborts a specified request.
- **Description**
Sets the value for request's description when you make a request using *Request()* or *RequestAsync()* functions without parameters.
- **Message**
Sets the value for the message when you make a request using *Request()* or *RequestAsync()* functions without parameters.
- **ErlangServerInterface**
Establishes the connection interface with the server.
- **OnReceive(*IServerReply _reply*)**
Event that fires when any message is received on the client, the parameter is the message.
- **Request(*string _description, object _message*)**
Makes a *Normal* kind request to the server, returns an *IRequestInfo* object.
- **RequestAsync(*string _description, object _message*)**
Makes *Async* kind request to the Erlang server, returns no request's information.
- **Request(*object _message*)**
Makes a *Normal* kind request to the server retrieving the description for the request from the *_message* object, returns an *IRequestInfo* object.
- **RequestAsync(*object _message*)**
Makes *Async* kind request to the Erlang server retrieving the description for the request from the *_message* object, returns no request's information.

- **SetOwnerForm(*Form _ownerForm*)**

Sets the owner form, this makes possible the validation of *thread cross operation error* without the usage of invoke method in any of *OnReceive* and *OnDisconnected* events.

- **PublishClientImage()**

Publishes to all the clients connected to the server the client's image. This is an extra functionality only for programming aims.

IRequestInfo: It's a class or interface in order to contain information about requests made to the server

- **AbortRequest()**

Aborts the request pointed by the reqinfo object.

- **Description**

Property in order to retrieve the description of the made request.

- **Kind**

Property in order to retrieve the kind of the made request (*Normal, Async, ServerSent*).

- **WaitForReply()**

Waits the answer to return the control to the program.

- **WaitForReply(*int _timeout*)**

Waits the answer to return the control to the program using the *_timeout* parameter to abort the waiting.

- **WasAnswered**

Indicates if the request was answered.

- **WasMade**

Indicates if the request was made.

IServerReply: It's a class or interface that contains information about the answer of a made request.

- **CSharpReply**

Contains the answer of the made request in *CSharp* object format.

- **Description**

Contains the description of the made request.

- **ErlangReply**

Contains the answer of the made request in *Otp.Erlang.Object* format.

- **RequestInfo**

Contains the *IRequestInfo* object of the made request.

RequestKind: It's a *CSharp enum* that contains the kind of requests present in EVO.

- **Async**

It's a request that no waits for it to be completed, neither waits for the answer to be received to return the control to the program. Returns no information about the request, the answer can be received through *Request-OnReceive* event of the *Async* way. It's no possible to abort a request made of *Async* way, but it is possible to abort all made requests of that kind using *AbortAsyncRequests()* function of the *ErlangServerRequest* component.

- **Normal**

It's a request that waits for it to be completed to return the control to the program, but not for the answer to be received. Returns an *IRequestInfo* object, it's possible to wait for the answer using the function *WaitForReply()* of the same *IRequestInfo* object, or to abort it using *AbortRequest()* function.

- **ServerSent**

It's a kind of request that was not required by the client, this means that it could be a global message that server can send without be requested by any client. An example of that is the function *send_to_all(Description, Message)* of the template, *evo_template*, for server applications. The *IRequestInfo* object of a message of *ServerSent* kind can be obtained by the property *RequestInfo* of the *_reply* object of the class or interface *IServerReply* once the message is received on the client application through the *Interface-OnReceive* event. That kind of messages can be redirected to a *ErlangServerRequest* handler using the function *SendToErlangServerRequest(ErlangServerRequest _erlangServerRequest, IServerReply _reply)* of *ErlangServerInterface* component.

The *evo_template* template

The *evo_template* has many files listed at following:

- **1.install.cmd**
Installs the server application and its database.
- **1.start.cmd**
Starts the server application.
- **2.service_install-start.cmd**
Installs the server application as a windows service.
- **2.service_stop.cmd**
Stops the windows service execution.
- **2.service_uninstall.cmd**
Uninstalls the windows service.
- **compile.cmd**
Compiles the sources located in the folder [*sources*].
- **config.cmd**
The configuration file.
- **Folder [*beam*]**
Contains the compiled files.
- **appname.app**
The Erlang application file.
- **appname.beam**
The start point of EVO server application.
- **appname_debug_module.beam**
Module to trace the message passing.
- **appname_main_interface.beam**
The main interface with the client.
- **appname_main_supervisor.beam**
The supervisor module, to make possible the interrupted functioning (*Erlang/Otp behavior*).
- **Folder [*sources*]**
Contains the sources.

- **appname_db_module.erl**

The database module, implements a layer for mnesia database. Here you can write the database scripts.

- **appname_db_server.erl**

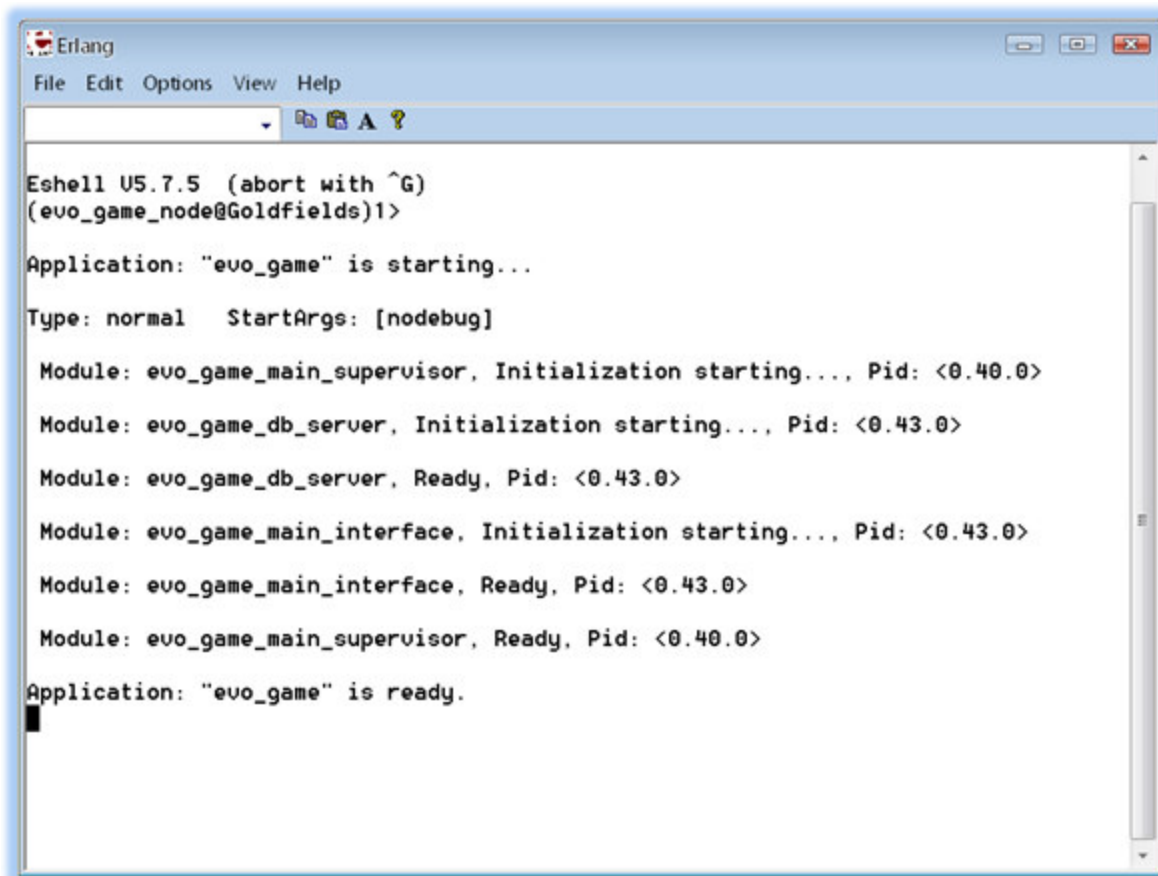
The server module for the database's layer (*gen_server behavior*).

- **appname_request_handler.erl**

The request handler module, here you can write the code to handle the requests made by the clients.

- **user_default.erl**

The EVO shell commands.



```

Eshell U5.7.5 (abort with ^G)
(evo_game_node@Goldfields)1>

Application: "evo_game" is starting...
Type: normal   StartArgs: [nodebug]

Module: evo_game_main_supervisor, Initialization starting..., Pid: <0.40.0>
Module: evo_game_db_server, Initialization starting..., Pid: <0.43.0>
Module: evo_game_db_server, Ready, Pid: <0.43.0>
Module: evo_game_main_interface, Initialization starting..., Pid: <0.43.0>
Module: evo_game_main_interface, Ready, Pid: <0.43.0>
Module: evo_game_main_supervisor, Ready, Pid: <0.40.0>

Application: "evo_game" is ready.

```

Illustration 4. Erlang server application the *evo_template*

Now a brief view of the *evo_request_handler.erl* file

%% DO NOT DELETE ANY OF THESE CLAUSES.

```
-module(evo_request_handler).
```

```
-import(evo_main_interface, [send_to_all/2, send_to_others/3, send_to/3]).
```

```
-export([loopRequest/1, loopOther/1]).
```

%% Request process code.

```
loopRequest(Args)->
```

```
    receive
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% USER FUNCTIONS TO MODIFY
%%
%% Request: {Msg, Pid, RequestInfo}
%% Reply: {reply, Reply, RequestInfo}
%%
%% RequestInfo = {Description, Id, WasMade, Kind, RequestHandlerHash}
%%
%% NOTE:
%% - You can use send_to_all(Description, Msg) function to send a message to all clients.
%% - You can use send_to_others(Description, Msg, ExceptionPid) function to send a message
%% to all clients except ExceptionPid.
%% - You can use send_to(WhoPid, Description, Msg) function to send a message to a client.
%% - Do not use 'loopRequest(Args)' calls in any clause.
%% - You must end every message treat clause with ';'
%% - Do not use process dictionary or any local data store, because this process dies once
%% the request is resolved.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% For send_image example.
{{send_image, Data}, Pid, _RequestInfo}->
    send_to_others(send_image, Data, Pid);

%% For evo_game_client example.
{{update_char, CharName, X, Y}, _Pid, _RequestInfo}->
    send_to_all(update_char, {CharName, X, Y});

{{update_ref, CharName, X, Y}, _Pid, _RequestInfo}->
    send_to_all(update_ref, {CharName, X, Y});

{clear, _Pid, _RequestInfo}->
    send_to_all(clear, all);

{{saludate, Msg}, Pid, RequestInfo}->
    Pid ! {reply, Msg, RequestInfo};

{"hi all", _Pid, _RequestInfo}->
    send_to_all('send to all hi all', "hi all");

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% TEMPLATE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Please do not remove any of these clauses. Do it only if you know what you do.

%% The EVO Test Application clause.
{evo_test, Pid, RequestInfo}->
    timer:sleep(5000),
    Pid ! {reply, "ok", RequestInfo};

%% The send_to message.
{{send_to, WhoPid, Message}, Pid, RequestInfo = {Description, _, _, _}}->
    send_to(WhoPid, Description, Message),
    Pid ! {reply, "The message was successfully sent.", RequestInfo};

```

```

%% Treats the publish_client_image message.
{{publish_client_image, FileName, Bytes, IsLastFile, ExecutableFileName}, Pid, RequestInfo}->
    evo_debug_module:print(Args, "~nSending client_image file: ~P ~P ~P to all the clients.~n", [FileName, 10,
        IsLastFile, 10, ExecutableFileName, 10]),
    send_to_others(published_client_image, {FileName, Bytes, IsLastFile, ExecutableFileName}, Pid),
    Pid ! {reply, "Client file: " "FileName" " published", RequestInfo};

```

```

%% The general message clause.

```

```

{Msg, Pid, RequestInfo}->

```

```

    Reply =

```

```

        %% Code Here.

```

```

        Msg,

```

```

%% Replies to client.

```

```

    Pid ! {reply, Reply, RequestInfo},

```

```

%% If you want to send a message to all clients.

```

```

    send_to_all(no_description, Reply),

```

```

%% If you want to send a message to all clients but this.

```

```

    send_to_others(no_description, Reply, Pid),

```

```

%% Debug message.

```

```

    evo_debug_module:print(Args, reply, {Reply, Pid, RequestInfo})

```

```

end.

```

```

loopOther(_Arg)->

```

```

    receive

```

```

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

        %% OTHER MESSAGES

```

```

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

    Other->

```

```

        %% Here what you want to do with Other message.

```

```

        Other

```

```

end.

```

The Big Example

Now shall be shown the big example in order to understand better the EVO component.

Example #2: Given a source code of a small server application:

```
{{update_char, CharName, X, Y}, Pid, RequestInfo}->  
    send_to_all(update_char, {CharName, X, Y});  
{{update_ref, CharName, X, Y}, Pid, RequestInfo}->  
    send_to_all(update_ref, {CharName, X, Y});  
{clear, Pid, RequestInfo}->  
    send_to_all(clear, all);
```

We can write a little and simple client application in order to connect it to the server application written before, this client application has the functionality of, draw traces in the screen using of course, the mouse, and to draw the same traces in all other applications connected to the server.

```
/*  
 * Created by Ivan Carmenates García.  
 * Using SharpDevelop  
 * Date: 10/09/2010  
 * Time: 8:12  
 */  
using System;  
using System.Drawing;  
using System.Windows.Forms;  
  
namespace evo_game_client  
{  
    public partial class MainForm : Form  
    {  
        private int x = 0;  
        private int y = 0;  
        private string char_name = "No Name" + DateTime.Now.Ticks;  
  
        public MainForm()  
        {  
            InitializeComponent();  
            this.erlangServerInterface1.SetOwnerForm(this);  
            this.erlangServerRequest1.SetOwnerForm(this);  
  
            this.TBOX__server_address1.Text = this.erlangServerInterface1.RemoteAddress;  
            this.TBOX__char_name1.Text = this.char_name;  
        }  
  
        private void BTN__connect1Click(object sender, EventArgs e)  
        {  
            this.LBL__connect_information1.ForeColor = Color.Blue;  
            this.LBL__connect_information1.Text = "Connecting with server.";  
            this.Update();  
            if (TryConnect(this.TBOX__server_address1.Text)) {  
                this.char_name = this.TBOX__char_name1.Text;  
            }  
        }  
    }  
}
```

```

        this.LBL__connect_information1.ForeColor = Color.Green;
        this.LBL__connect_information1.Text = "Connected successfully.";
        this.Update();
        System.Threading.Thread.Sleep(1000);
        this.PNL__login_information1.Visible = false;
        this.BackColor = Color.Black;
        this.pictureBox1.Visible = true;
    } else {
        this.LBL__connect_information1.ForeColor = Color.Red;
        this.LBL__connect_information1.Text = "Couldn't connect.";
    }
}

void CreateGraphicsObject(string name, int x, int y)
{
    Brush br = new SolidBrush(Color.Yellow);
    Pen pen1 = new Pen(br);
    this.pictureBox1.CreateGraphics().DrawLine(pen1, this.x, this.y, x, y);
    this.x = x;
    this.y = y;
}

void PictureBox1MouseDown(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Right) {
        ClearChar();
        this.pictureBox1.CreateGraphics().Clear(Color.Black);
    } else {
        this.x = e.X;
        this.y = e.Y;
        UpdateRef(e.X, e.Y);
    }
}

void PictureBox1MouseMove(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Left) {
        UpdateChar(e.X, e.Y);
        CreateGraphicsObject(this.char_name, e.X, e.Y);
    }
}

// Server Interface.
#region Server Interface.

private bool TryConnect(string server_address)
{
    this.erlangServerInterface1.RemoteAddress = server_address;
    try {
        this.erlangServerInterface1.Connect();
        return true;
    } catch {}
    return false;
}

```

```

private void UpdateChar(int x, int y)
{
    try {
        if (this.erlangServerInterface1.IsConnected) {
            // Makes a request to server to update the char position.
            this.erlangServerRequest1.Request(new object[] {
                "update_char",
                this.char_name, x, y});
        }
    }
    catch { }
}

private void UpdateRef(int x, int y)
{
    try {
        if (this.erlangServerInterface1.IsConnected) {
            // Makes a request to server to update the char position.
            this.erlangServerRequest1.Request(new object[] {
                "update_ref",
                this.char_name, x, y});
        }
    }
    catch { }
}

private void ClearChar()
{
    try {
        if (this.erlangServerInterface1.IsConnected) {
            this.erlangServerRequest1.Request("clear");
        }
    }
    catch { }
}

void ErlangServerRequest1OnReceive(ExtVisualOtp.IServerReply _reply)
{
    switch (_reply.Description) {
        case "update_char":
            string charname = (string)(_reply.CSharpReply as object[][0]);
            int x = (int)((_reply.CSharpReply as object[][1]);
            int y = (int)((_reply.CSharpReply as object[][2]);
            if (this.char_name != charname) {
                CreateGraphicsObject(charname, x, y);
            }
            break;
        case "update_ref":
            string charname2 = (string)(_reply.CSharpReply as object[][0]);
            int x2 = (int)((_reply.CSharpReply as object[][1]);
            int y2 = (int)((_reply.CSharpReply as object[][2]);
            if (this.char_name != charname2) {
                this.x = x2;
                this.y = y2;
            }
            break;
    }
}

```



```

        case "clear":
            this.pictureBox1.CreateGraphics().Clear(Color.Black);
            break;
    }
}

void ErlangServerInterface1OnReceive(ExtVisualOtp.IServerReply _reply)
{
    this.erlangServerInterface1.SentToErlangServerRequest(this.erlangServerRequest1, _reply);
}

void MainFormFormClosing(object sender, FormClosingEventArgs e)
{
    this.erlangServerInterface1.Disconnect();
}

void ErlangServerInterface1OnDisconnected(ExtVisualOtp.IServerReply _reply)
{
    try {
        this.pictureBox1.Visible = false;
        this.BackColor = Color.FromKnownColor(KnownColor.ButtonFace);
        this.LBL__connect_information1.ForeColor = Color.Red;
        this.LBL__connect_information1.Text = "Disconnected from server.";
        this.PNL__login_information1.Visible = true;
    } catch {}
}

#endregion
}
}

```

Annexes

The Big Example, client application's interface

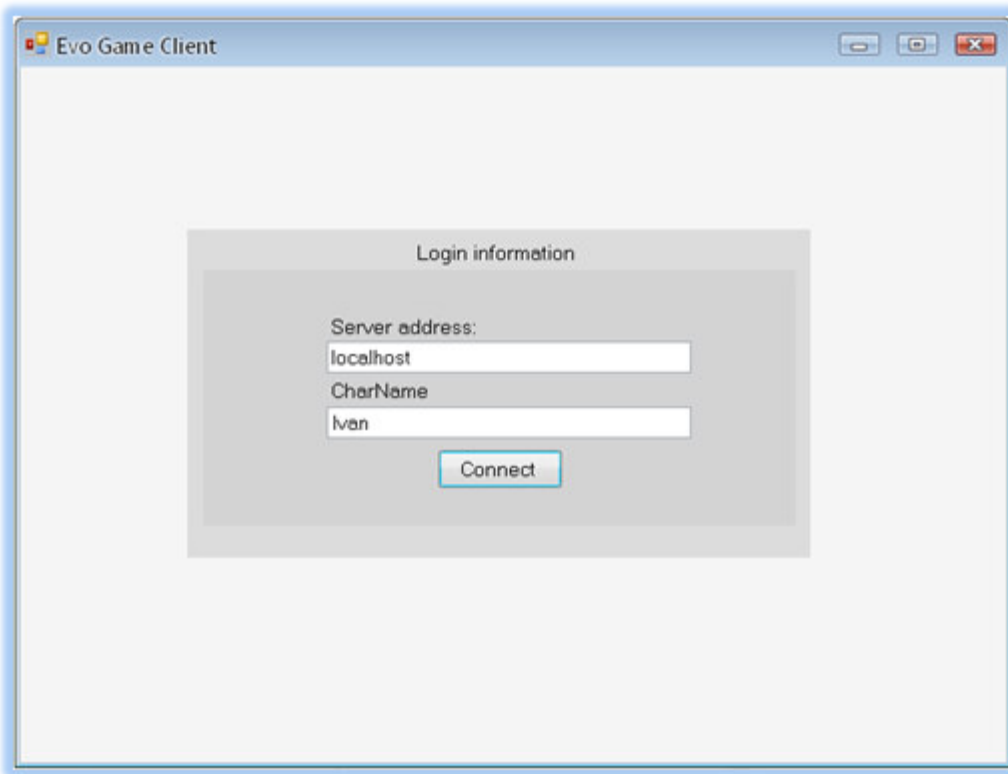


Illustration 5. Client 1

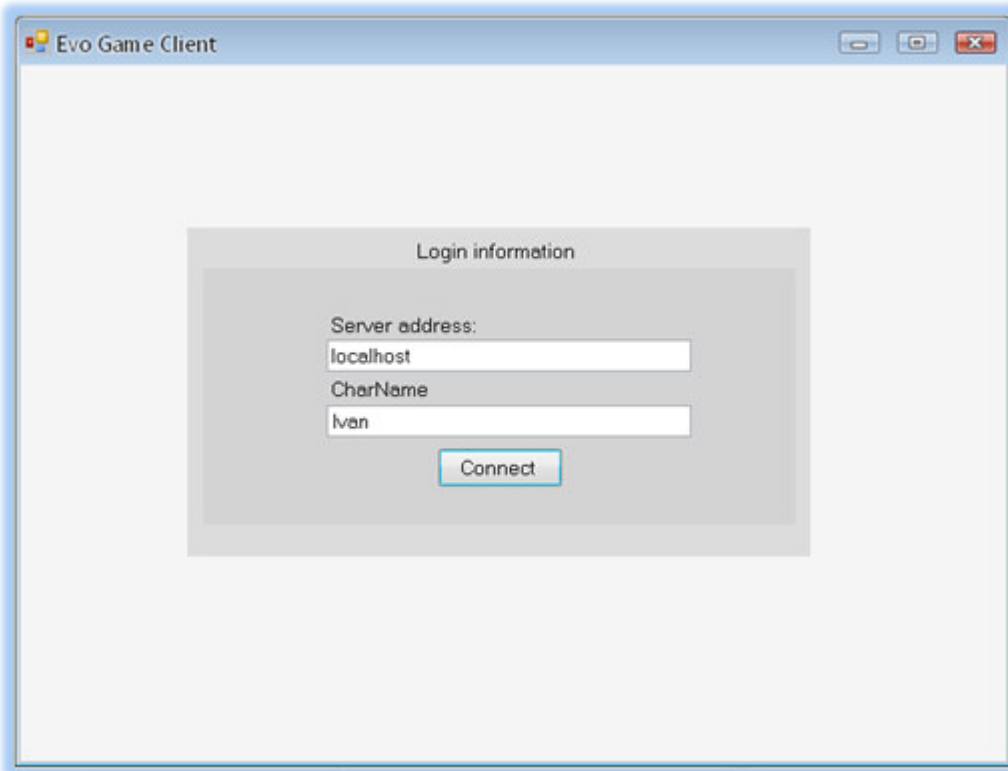


Illustration 6. Client 2

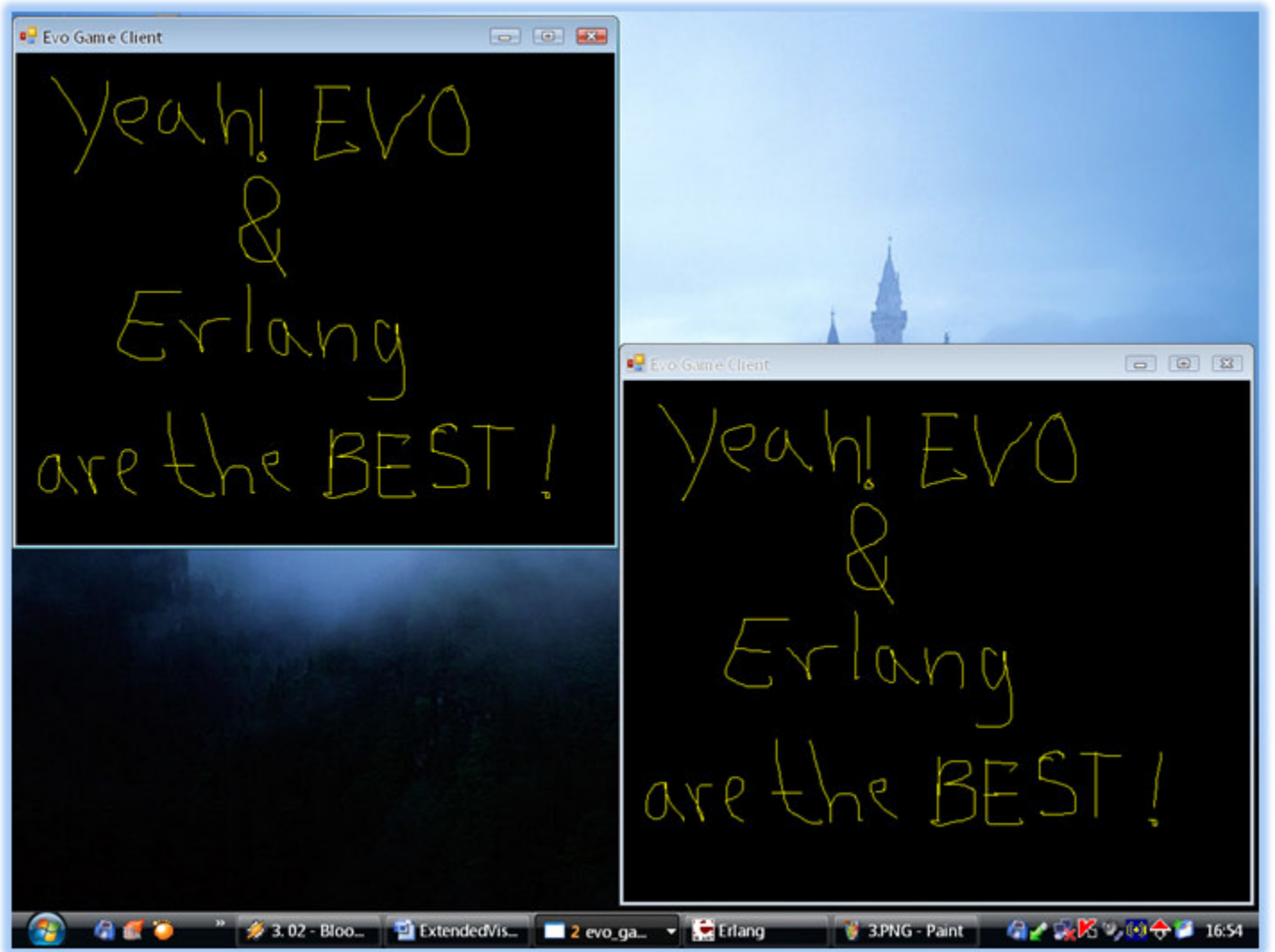


Illustration 7. Client 1 and Client 2 interacting through the server application

The EVO Test Application

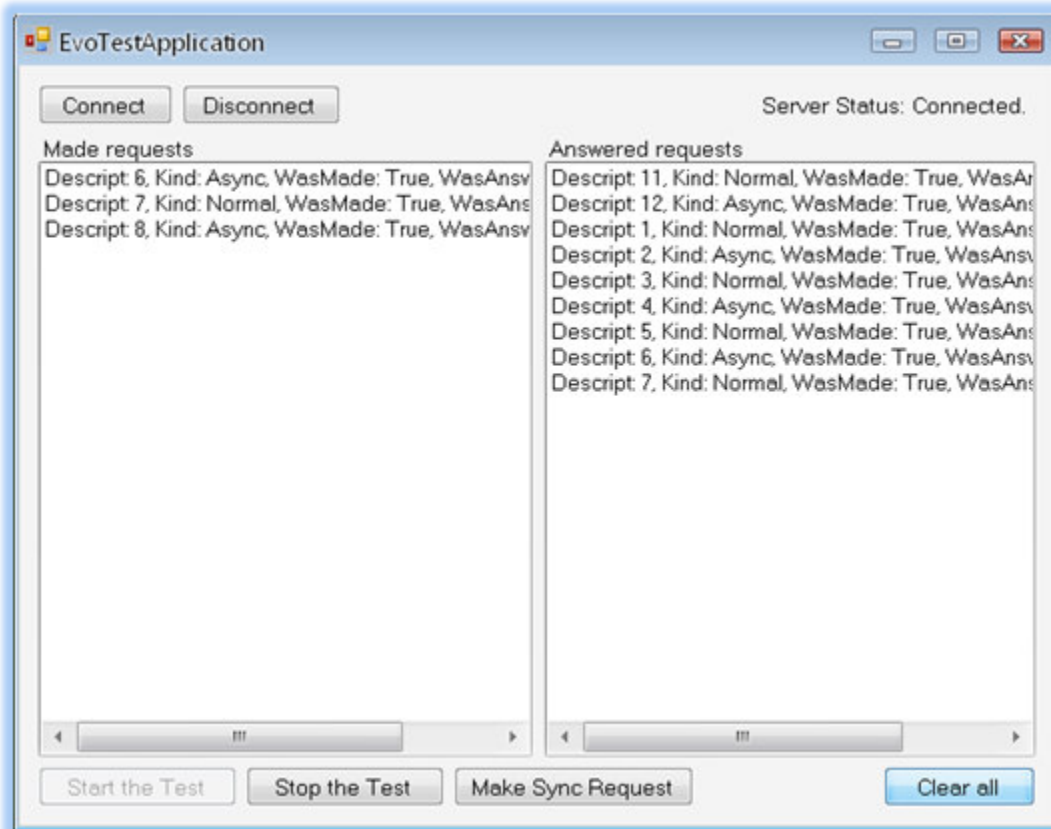


Illustration 8. The EVO Test Application

Conclusions

That's all for now, later it will continue the development of this material, with many more examples and deeper explanations of all classes and interfaces that are presented here.

Thanks for having your attention.