Here is proposed the implementation of a WCF service on the server machine and the deployment of client applications which can invoke the service on the server.  
Both client and server are being developed in Microsoft Visual Studio 2013 Ultimate in C# and the .NET framework version is V. 4.5.  
  
**SERVER**The server exposes a service with the default behaviour request-reply where the client issues a request and blocks its calling method until the reply message. A more responsive behaviour in the client can be implemented using async methods and the await keyword in next versions of the projects.  
  
The Service implementation on the server is by means of a Self Hosted WCF Service.  
Self Hosted means that it has his own application domain in an individual application and it is not installed on IIS and the server machine will not require IIS to run the service. This approach favours performance and maintainability and is positively viewed as a common best practice.  
  
The present implementation of the server side service container is in the form of a Windows Forms Server Application. Other scenario of deployment in production could be either a Windows Service or in a WPF application which is even better than the current Windows Forms because WPF is newer and can work better than Windows Forms in all regards (MVVM design Patterns, Use of Data Binding and better GUI Management thanks to the use of dispatchers).  
  
Windows Forms has been chosen for its simplicity and to demonstrate the design; however the conversion to WPF can be very easy to tackle because all the objects that make up the service implementation can be moved to class libraries and referenced in the new WPF container.  
**Important note: The server must run with administrator privileges.**  
  
**CLIENT**  
The supplied client application is like the server in the form of a Widows forms application. Even the client can be easily ported in WPF.  
  
**CORE SERVER ENGINE**  
The Service on the server allows the following protocols to be used:

HTTP wsHttpBinding basicHttpBinding  
TCP netTcpBinding  
  
The Service, once launched on the server is discoverable on the client machine using svcutil.exe as follows:  
Svcutil.exe <http://ServerIP:5341/WCFService/mex>   
Svcutil.exe net.tcp:// ServerIP:7777/WCFService/tcpmex  
Svcutil.exe [http:// ServerIP:8080/WCFService/mex](http://localhost:8080/WCFService/mex)

The outputs, after issuing one of the above commands, are the correspondent C# proxy classes and the configuration file to access the service. These are the core elements to build the client engine.  
The current client implementation uses the Tcp binding.

**Firewall rules** must be applied on the server and the client in order to open the TCP ports that are used for the service (Inbound and Outbound rules).

The security model chosen conveniently for testing purposes in the cloud is NONE.

Client concurrency on the server is handled locking the service per call i.e. a client will update the server and another client will queue to wait until the previous operation has completed. (This is a first degree of implementation in terms of concurrency management that will be considered further on based on requirements.).

**SECURITY**

The security used is transport and the use of Windows Credentials. The client is providing Credentials in code. That means that Username and Password indicated in Client Credentials must exist on the server. The client is Impersonating a specific Username.

Fig 1. The Server application on the Server Machine

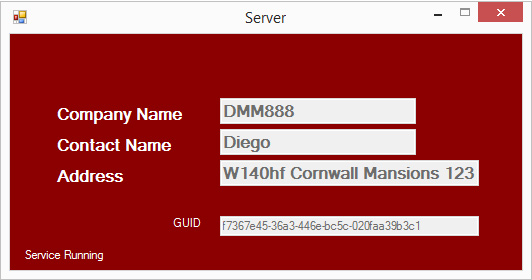
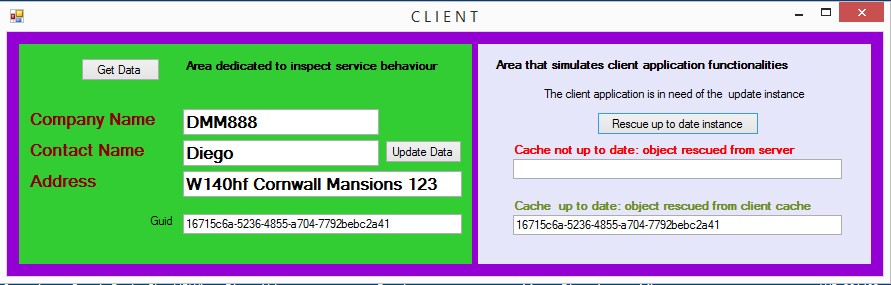
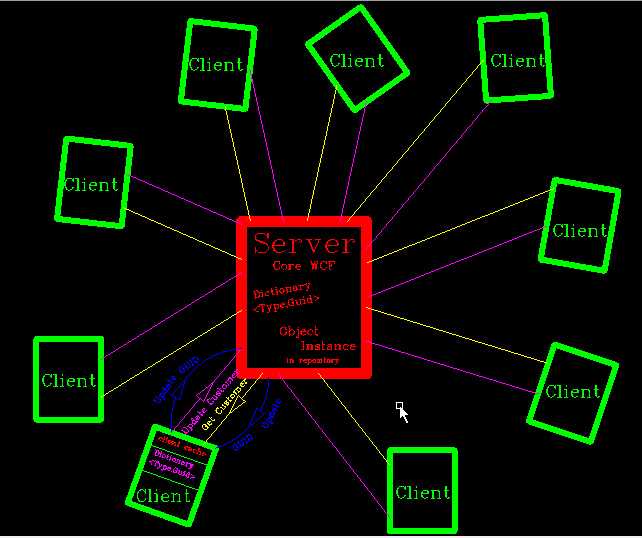


Fig. 2. The Client application on the Client Machine



**CACHING.**  
The instance of the object that initially resides on the server is there stored in a container (simple custom DB) and is retrieved every time a client requires it. The specific instance of the object on the server is related to a key-value pair stored in a Dictionary in a separate container where the key is the Type name of the current instance and the value is a GUID (Globally Unique Identifier).  
The choice between GUID and Timestamp is debatable and Diego is open to discuss  
  
The Dictionary<TypeName,GUI> allows different types of objects to be part of the list and every type will have its corresponding GUI. This observation is made in order to point out that in a real world there is more than one object type to manage in cache. In our simple scenario the Dictionary will contain only one row as follows in Pseudo Code:  
  
Key: “Customer” Value: {C6CD700D-3B03-41C8-B964-54C429F5DC41}  
  
The client has either a local cache where it stores the updated objects retrieved from the server and also the same Dictionary in a separate container. Every time it gets an instance of an object from the server it will also receive the GUID of the up to date instance from the server. If the client updates the object instance to the server, it first will write the updated instance in local cache and then send the update to the server. It will write a new GUID in its local Dictionary as value of the specific Key object type and post to the server such new GUID together with the updated instance and the server receiving them will update either the instance in the container and the row of the dictionary corresponding to the object type to a value equal to the new GUID sent from the client.   
  
When the client needs to use the Instance of the objects for its purposes, it must first check if the local GUID is equal to the Server GUID; if they are equal it means that there’s no need to get the object instance from the server because the one present in the local cache is up to date; instead, if they are different a get request is due.  
   
In a First-Time Client-Server Interaction the client requires to get the updated instance from the server. Since the client has neither the object instance in the local cache nor a key-value pair stored in the local Dictionary for the corresponding object, a mandatory get to the server is triggered which will provide the updated copy of the instance along with the GUID; the client cache will be enriched of the updated instance object and the local client Dictionary will be added with the key-value pair. When the client comes to a point where it needs to use the instance of the object it will check that the local GUID be equal to the server GUID by means of a method exposed by the Server because another client could have updated the server in the meantime. Providing the result of this comparison is true or false the client will either use the local copy of the instance or get an updated copy from the server.

**GRAPHICS**fifg 1





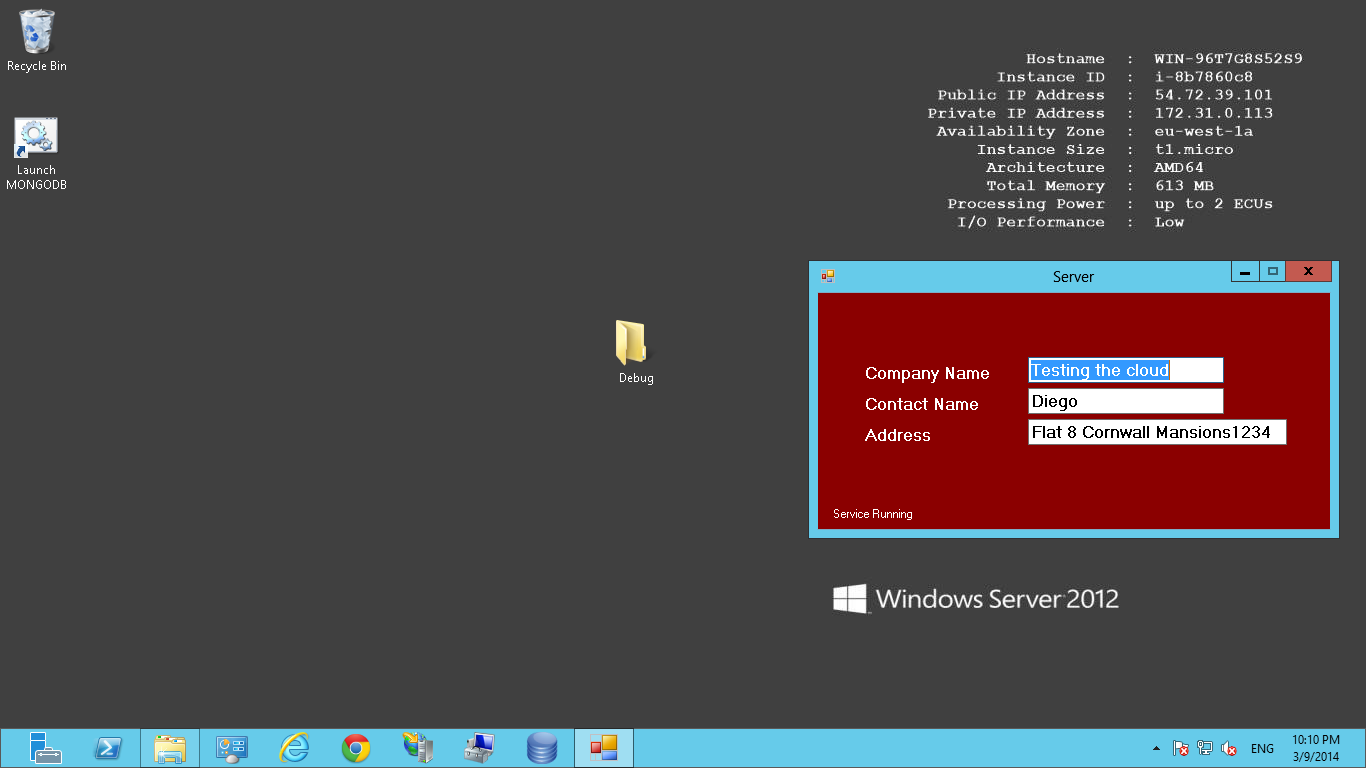
**UNIT TESTS AND INTEGRATION TESTS.**

The application has been tested so far during its assemblage. However in this specific scenario Unit testing is applicable in terms of verifying the correct instantiation of the repository class. And on the way of development other unit tests can be performed.  
The application client server on different machines has been tested using the following scenario:

Server Machine In The cloud using Amazon Micro Instance  
**ec2-54-72-39-101.eu-west-1.compute.amazonaws.com**

**Public IP: 54.72.39.101**





**THREAD SAFETY**  
The local client cache has been made thread safe and further investigation is required to grant full safety since it implements IO on the file system.  
  
USING GENERICS AS A FURTHER STEP TO IMPROVE SCALABILITY.  
There are parts of the server repository that can be written implementing generics in order to improve code standards and scalability. For the sake of the time allowed the present implementation is not yet using generics.

CUSTOM DEPENDENCY INJECTION FRAMEWORK.  
Part of the server development has been implemented using Dependency injection by means of passing an interface in constructor in order to decouple dependencies.

STRESS Tests.

Recently has been added a class library with a test that allows to create up to 1000 client requests.