C# example :

1. Prerequisite

For this example, you must install an IDE for C#: Microsoft Visual Studio. Download link: https://visualstudio.microsoft.com

2. The creation of a Visual Studio project

In Visual Studio, create a new project and choose “Application console (.NET Core). Click on “Next”, then enter a name and choose a location for your project. Then, click on “Create”.

Afterwards, in « Solution Explorer », right click on your project → Edit the project file and replace the line:

```
<TargetFramework>netcoreapp3.1</TargetFramework>
```

By :

```
<TargetFramework>net48</TargetFramework>
```

3. Include the DLLs

On the right side, in the Solution Explorer, right click on your project → Add → New folder → Name it « libs ».

Copy-paste the DLLs present in the KalimaCSharpExample/libs in yours. Most of the DLLs come from ikvm:

https://github.com/jessielesbian/ikvm

IKVM is an implementation of Java for Microsoft .NET. If you need more Java features, please install ikvm and include the necessary DLLs in your project.

Then, in Visual Studio, right click on your project → Add → Project Reference → Browse → Select all the DLLs that you’ve addded in your libs folder.

4. KalimaClientCallBack

Create a new class under called « KalimaClientCallBack »:

- ⇨ Right click on your project
- ⇨ Add
- ⇨ New element
- ⇨ Select « class » and enter « KalimaClientCallBack.cs » in the field « name »
- ⇨ Add

Find below the complete code of the class (the namespace can change depending on the name of your project)

At this stage, the project includes errors which will be corrected later.

```
using System;
using org.kalima.kalimamq.message;
using org.kalima.kalimamq.nodelib;
using org.kalima.kalimamq.netlib;
using org.kalima.util;
using org.kalima.cache.lib;
using java.util.concurrent.atomic;
using java.nio.channels;

namespace KalimaCSharpExample
{
  public class KalimaClientCallBack : ClientCallback
  {

    private Client client;
    private Logger logger;
    private Node node;
    private SendTimeout sendTimeout;

    public KalimaClientCallBack (Client client)
      {
        this.client = client;
        this.node = client.getNode ();
        this.logger = node.getLogger ();
        sendTimeout = new
SendTimeout(client.getClone().getClonePreferences().getLoadConfig().getS
endTimeout(), logger, node);

      }

    public void putData(SocketChannel ch, KMessage msg) {
      KMsg kMsg = KMsg.setMessage (msg);
      SendTimeout.remove(msg);
      client.getClone ().set (kMsg.getCachePath(), kMsg, true, false);
Console.WriteLine("putData cachePath=" + kMsg.getCachePath() + " key=" +
kMsg.getKey() + " body=" + System.Text.Encoding.Default.GetString(kMsg.getBody()));
    }

    public void onConnectionChanged(int status, NioClient nioClient){
      logger.log_srvMsg ("ExampleClientNode", "KalimaClientCallBack",
Logger.DEBUG, "onConnectionChanged status=" + status);
      client.getClone().onConnectedChange(status==Node.CLIENT_STATUS_CON
NECTED) ? new AtomicBoolean(true) : new AtomicBoolean(false), nioClient,
false);
    }

    public void onCacheDeleted (string cacheSubPath){
      logger.log_srvMsg ("ExampleClientNode", "KalimaClientCallBack",
Logger.DEBUG, "onCacheDeleted cacheSubPath=" + cacheSubPath);
    }

    public void putRequestData(SocketChannel ch, KMessage msg) {}

    public void onNewVersion(int majver, int minver) {}

    public void onNewCache(String cachePath) {}

    public void onCacheSynchronized(String str) {}

    public void onNewCache(KMessage km) {
      sendTimeout.put(Kmsg.setMessage(km));
      }
   }
}
```

5. Client

Lastly, rename your main class (by default Program.cs) by Client.cs → Right click on Program.cs → Rename.

Find below the complete code of the class:

```
using System;
using org.kalima.kalimamq.nodelib;
using org.kalima.kalimamq.message;
using org.kalima.kalimamq.crypto;
using org.kalima.cache.lib;
using ikvm.extensions;

namespace KalimaCSharpExample
{
  public class Client : KalimaNode
  {

    private Node node;
    private Clone clone;
    private Logger logger;
    private KalimaServerCallBack kalimaServerCallback;
    private KalimaClientCallBack kalimaClientCallback;
    private ClonePreferences clonePreferences;
    private byte[] devId;

    public static void Main(string[] args)
    {
      try
      {
        Client client = new Client(args);
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }
     }

     public Client(string[] args)
     {
       clonePreferences = new ClonePreferences(args[0]);
       logger = clonePreferences.getLoadConfig().getLogger();
       initComponents();
       System.Threading.Thread.Sleep(2000);

       for (int i = 0; i < 10; i++)
       {
         String body = "hello" + i;
         KMsg kMsg = new KMsg(0);
         node.sendToNotaryNodes(kMsg.getMessage(devId, KMessage.PUB,
"/sensors", "key" + i, body.getBytes(), new KProps("10")));
         System.Threading.Thread.Sleep(1000);
       }
     }
```

```
    public void initComponents()
    {
      node = new Node(clonePreferences.getLoadConfig());
      clone = new Clone(clonePreferences, node);

      kalimaClientCallback = new KalimaClientCallBack(this);

      node.connect(null, kalimaClientCallback);
     }

     public Node getNode()
     {
      return node;
     }

     public Logger getLogger()
     {
      return logger;
     }

     public Clone getClone()
     {
      return clone;
     }

   }
```

6. Configuration file

Add a cfg folder in your project and place a « node.config » file in it. That configuration file will be passed as a parameter of the application.

See below for an example of a configuration file :

```
LedgerName=KalimaLedger
NODE_NAME=Node Client Example

NotariesList=62.171.131.154:9090,62.171.130.233:9090,62.171.131.157:9090,144.91
.108.243:9090
FILES_PATH=/home/rcs/jit/ClientExample
SerialId=PC1245Tuto
#Watchdog=60000
SEND_TIMEOUT=10000
```

- LedgerName ➔Is not yet used in the current version
- NODE_NAME ➔ You can put something which allows you to recognize your node
- NotariesList ➔ The address and port list of the notary, separated by commas
- FILES_PATH ➔ It is the path where the files useful for Kalima will be stored, as well as the logs
- serialId ➔ It is an ID which will allow the authorization on the blockchain of the first launch of the client node (provided by Kalima Systems in the case of trials on our Notary)

- WATCHDOG ➔ The watchdog here is commented out, it is not useful for our example. The watchdog is useful in case of prolonged connection with the Blockchain. It corresponds to a time (here 60 000 = 10 minutes). In fact, every 10 minutes, we will check that the connection is still active. You can then for example make a SmartContract that will send you an email if the connection is lost. It is thus useful to check the state of the network.
- SEND_TIMEOUT ➔ Corresponds to the maximum time considered for sending a message. If at the end of the time indicated, the message is not well received, we try to send it again (here 10 seconds).

7. Code Execution

To test your project, you can execute the code from Visual Studio, or from an online command console. You need to pass the configuration file as parameter.

Execution from Visual Studio

In Debug ➔ Properties of debug of … ➔ In the Bebug tab ➔ Pass the complete path or relative of the configuration file in "Arguments of the application" (example: ../../../cfg/node.config). Lastly, click on F5 to launch the application.

Command-line execution

First, in Visual Studio, generate the solution by clicking on F6. If the generation works, the executable and the necessary DLLs have been copied in the bin folder (in bin/Debug/net48/ for example).

Then, from the console, go to the executable folder and launch it by passing the configuration file as parameter:

```
cd
Documents\Kalima\git\KalimaTuto\KalimaCSharpExample\KalimaCSharpExample\
bin\Debug\net48
KalimaCSharpExample.exe ..\..\..\etc\cfg\node.config
```

8. Results

The example program connects to the Blockchain, then sends 10 messages (1/second). The TTL (Time To Live) of those messages is of 10 which means that each message will be automatically deleted after 10 seconds (a transaction will happen on the blockchain for each deletion). Thus, if your code is correct, if you have configured the configuration file correctly, and if your device is authorized on the blockchain, you should have something similar to this in your console:

GO

putData cachePath=/sensors key=key0 body=hello0

putData cachePath=/sensors key=key1 body=hello1

putData cachePath=/sensors key=key2 body=hello2

putData cachePath=/sensors key=key3 body=hello3

putData cachePath=/sensors key=key4 body=hello4

putData cachePath=/sensors key=key5 body=hello5

putData cachePath=/sensors key=key6 body=hello6

putData cachePath=/sensors key=key7 body=hello7

putData cachePath=/sensors key=key8 body=hello8

putData cachePath=/sensors key=key9 body=hello9

putData cachePath=/sensors key=key0 body=

putData cachePath=/sensors key=key1 body=

putData cachePath=/sensors key=key2 body=

putData cachePath=/sensors key=key3 body=

putData cachePath=/sensors key=key4 body=

putData cachePath=/sensors key=key5 body=

putData cachePath=/sensors key=key6 body=

putData cachePath=/sensors key=key7 body=

putData cachePath=/sensors key=key8 body=

putData cachePath=/sensors key=key9 body=

Results explanation :

- In the first part the client will send 10 messages in 10 seconds. The messages will be received by all the nodes authorized on the cache path, including yours. Thus, you will have, in the logs, a line for each message sent (lines starting with « `StoreLocal` »).
- Afterwards, the messages will be deleted one by one, since the TLL has been configured on 10 seconds. You will be able to see the transactions in the logs with an empty body.